Genomic Computing with SciDB, a Data Management System for Scientific Applications

Bioinformatics Group
Politecnico di Milano

Advisor:
Prof. Stefano Ceri

Master thesis of:
Simone Cattani - 816450

Academic Year 2015-2016
Acknowledgment

I would like to thank my supervisor prof. Stefano Ceri, for the great opportunity he has given me. Working in such a challenging and instructive environment has allowed me to make an important experience that has certainly contributed to my professional training. Another thanks goes to my friends and colleagues of the Bioinformatics group at Politecnico, Stefano, Arif, Vahid and especially Pietro and Abdo for their contribution to this work.

Desidero ringraziare tutti coloro che mi hanno supportato durante questo progetto e nel corso di tutta la mia carriera accademica.

Innanzi tutto un affettuoso grazie alla mia sorellina e ai miei genitori: grazie per il vostro costante supporto e per l’incondizionata fiducia, grazie per ogni piccolo o grande sacrificio che questo lungo percorso ha richiesto e grazie anche per tutte le pagine di quaderno scritte e riscritte, alla fine a qualcosa sono servite. Desidero ringraziare tutta la mia famiglia, in particolare nonno Giuseppe, nonna Anna, zia Carla e la mia cuginetta Sara.

Un caloroso abbraccio a Bruno, Anna e soprattutto Noelia, per essermi stati accanto con pazienza ed affetto durante tutto questo periodo.

Ringrazio tutti i miei amici, con cui ho condiviso momenti di crescita professionale e soprattutto personale. Un abbraccio ai miei coscritti Samuel, Rossella, Stefano, Andrea e Sabrina, che porto sempre vicino anche quando molto lontani. A Bert e a tutti i “Lizzardi”: grazie di cuore per i bellissimi anni passati assieme a sudare sui libri condividendo gioie e dolori.

Il mio ultimo pensiero va a nonno Alberto, che per me ha sempre rappresentato un esempio e una fonte di ispirazione. Grazie di tutto nonno.
Abstract

A new technology for reading the DNA, called Next Generation Sequencing (NGS), is changing biological research and medical practice, thanks to the low-cost availability of millions of whole genome sequences of a variety of species, and most important of humans. So far, the bio-informatics research community has been mostly challenged by primary and secondary analysis (data alignment and feature calling) but the emerging problem today is the so-called tertiary analysis, concerned with multi-sample processing, annotation and filtering of variants, and genome browser-driven exploratory analysis. The amount of data for tertiary analysis requires Big Data management.

The GenData 2020 project is focused on this problem. The project developed new abstractions for querying heterogeneous genomic datasets, centered on the notion of Genomic Data Model (GDM) and Genometric Query Language (GQL). GDM describes genomic datasets produced by NGS experiments, and includes both regions and metadata associated to DNA experiments; GMQL is a high-level, algebraic language which operates upon GDM and provides both conventional and domain-specific operations.

In this thesis we developed a new implementation of GMQL. We used SciDB, a data management system for scientific applications developed by Paradigm4, a startup company located in Cambridge. SciDB provides abstraction from storage management, data distribution, and optimized parallel execution. In the thesis, we provide a general framework for translating GDM and GMQL into SciDB; in particular, the translation of GMQL requires a general description of queries as DAGs of operations and then a specific translation of each GMQL operation into low-level DAG nodes.

Other implementations of GMQL use cloud computing frameworks (Pig, Flink and Spark) to execute operations directly on files; our work covers all the GMQL language and provides a complete alternative baseline implementation, based on the SciDB database engine. In comparison, we obtained better performance on various operators which can exploit the data model
of SciDB, but worse performance on massive operations (such as map and join), where the implementation based on Spark is faster. However, a lot of options exists for improving our baseline SciDB implementation.

Along this objective, we focused on a new abstractions for parallelism; we realized that parallelism of massive operations on the genome requires binning, i.e. the partitioning of the genome into portions so that operations are performed in parallel at each bin. We then studied the mono-dimensional binning supported by the current SciDB implementation, and we designed bi-dimensional binning, an alternative strategy also applicable to the SciDB data model. We focused on range intersection, where the challenge is to build a binning strategy comparing as few regions as possible, by limiting the comparison just to the regions that can intersect. In comparison, bi-dimensional binning obtains better performance on several datasets and scales better with data of increasing size.
Sommario

Una nuova tecnologia per leggere il DNA, chiamata Next Generation Sequencing (NGS), sta cambiando la ricerca biologica e le pratiche mediche, grazie alla disponibilità a basso costo di milioni di sequenze di DNA di una vasta varietà di specie, tra cui l'uomo. Finora la comunità di ricerca bioinformatica si è concentrata perlopiù sull'analisi primaria e secondaria (allineamento e correlazione), ma il problema recente è quello dell'analisi terziaria, che riguarda il trattamento di molti sample sperimentali e l'esplorazione attraverso browser visivi. La quantità di dati processata per l'analisi terziaria richiede un sistema Big Data.

GenData 2020 è un progetto di ricerca nato per affrontare questo problema. Lo sviluppo di questo progetto ha portato alla definizione di un nuovo sistema generico per l'interrogazione di sorgenti di dati eterogenee, basato su un modello dati chiamato GDM e un linguaggio di interrogazione del GDM, chiamato GMQL. GDM permette di descrivere i dati provenienti da esperimenti NGS, includendo dati relativi sia alle regioni che ai metadati derivati dagli esperimenti sul DNA; GMQL è un linguaggio algebrico di alto livello che opera su GDM e fornisce sia operatori convenzionali che specifici al dominio genomico.

In questa tesi abbiamo sviluppato una nuova implementazione di GMQL. Abbiamo deciso di utilizzare SciDB, un sistema per la gestione dati orientato alle applicazioni scientifiche e sviluppato da Paradigm4, una startup localizzata a Cambridge. SciDB permette di astrarre il sistema di gestione della persistenza, della distribuzione dei dati e della ottimizzazione delle operazioni tramite parallelismo. In questa tesi proponiamo un framework per la conversione di GDM e GMQL su SciDB; nello specifico, verranno descritte le traduzioni richieste per trasformare ogni singolo nodo del DAG in una operazione a basso livello di GMQL.

Le implementazioni di GMQL esistenti fanno uso di framework basati sul cloud computing (Pig, Flink e Spark) per eseguire le operazioni direttamente su file; il nostro lavoro copre tutte le operazioni GMQL e fornisce
una alternativa stabile e completa basata su SciDB. Comparando le implementazioni, abbiamo ottenuto migliori prestazioni su molti operatori in grado di sfruttare il particolare modello dati di SciDB, mentre risultati peggiori sono stati registrati su operazioni più pesanti (quali la map e la join), dove l’implementazione basata su Spark risulta vincente. In ogni caso, abbiamo definito una serie di valide opzioni per migliorare l’implementazione su SciDB.

In tal senso, ci siamo concentrati su nuove metodologie di parallelismo; abbiamo realizzato che per eseguire parallelismo sulle maggiori operazioni è necessario effettuare del binning, per esempio partizionando l’intero genoma in varie sezioni, eseguendo quindi le varie operazioni su ogni singola sezione. Abbiamo studiato il binning mono-dimensionale, ad oggi implementato seguendo le metodologie proposte in letteratura, e abbiamo progettato il binning bidimensionale, una strategia alternativa applicabile al modello dati di SciDB. Ci siamo concentrati sulla intersezione di intervalli, dove l’obiettivo è costruire una strategia che sia in grado di comparare il minor numero di regioni, limitando la selezione a quelle regioni che possono effettivamente intersecarsi. Comparando le due metodologie, il binning bi-dimensionale ottiene prestazioni migliori su varie tipologie di dataset e scala in maniera più efficiente all’aumentare del numero di dati.
Acronyms

**AFL**     Array Functional Language
**AQL**     Array Query Language
**API**     Application Programming Interface
**DAG**     Directed Acyclic Graph
**DBMS**    Database Management System
**EBM**     Empty Bitmap
**GDM**     Genomic Data Model
**GMQL**    Genometric Query Language
**HDFS**    Hadoop Distributed File System
**MAC**     Multidimensional Array Clustering
**NGS**     Next Generation Sequencing
**RDBMS**   Relational Database Management System
**SSH**     Secure Shell
Contents

1 Introduction 1

2 Model and Language 5
   2.1 Genomic Data Model ................................. 5
      2.1.1 Motivation ...................................... 5
      2.1.2 Definition ........................................ 6
      2.1.3 Examples ......................................... 7
   2.2 Genometric Query Language ...................... 8
      2.2.1 General properties ............................. 8
      2.2.2 Predicates evaluation ......................... 9
      2.2.3 Relational GMQL operations ................ 10
      2.2.4 Domain-specific GMQL operations ........... 14

3 Architecture of Gendata 2020 23
   3.1 Software layers ...................................... 23
   3.2 Internal representation ............................. 24

4 SciDB 29
   4.1 Background ........................................... 29
   4.2 Array data model ..................................... 30
   4.3 Multidimensional Array Clustering (MAC) ........ 32
   4.4 Array Functional Language (AFL) .................. 35
      4.4.1 iQuery client ................................. 36
      4.4.2 Operators ....................................... 36
   4.5 Genomic addition ..................................... 39
      4.5.1 Binning strategy ................................ 40

5 Scala APIs for SciDB 43
   5.1 Array representation ................................ 44
   5.2 Operator methods .................................... 45
   5.3 Stored arrays ........................................ 46
6 GMQL Implementation on SciDB

6.1 Architecture ............................................. 51
  6.1.1 Internal architecture .................................. 52
  6.1.2 Data flow ............................................. 53

6.2 Data model .................................................. 54
  6.2.1 Meta data ............................................... 55
  6.2.2 Region data ............................................. 56
  6.2.3 Chunk sizes .............................................. 57
  6.2.4 Example ................................................ 58

6.3 Query Processing ............................................ 59
  6.3.1 General procedure ...................................... 59
  6.3.2 Materialization ......................................... 61
  6.3.3 Translation tree ........................................ 61
  6.3.4 Translation node ........................................ 62

6.4 Translation of operators ................................... 63
  6.4.1 Select ................................................ 63
  6.4.2 Extend ................................................ 67
  6.4.3 Map ................................................ 69

7 Bidimensional Binning ........................................ 75

7.1 Motivations ................................................ 75

7.2 Representation ............................................. 77

7.3 Search space window ...................................... 78

7.4 Implementation ............................................ 81

8 Experiments .................................................... 83

8.1 Execution environment .................................... 83

8.2 Technology comparison .................................... 84
  8.2.1 Datasets ............................................... 84
  8.2.2 Region filtering ....................................... 84
  8.2.3 Aggregation ............................................ 86
  8.2.4 Range intersection .................................... 87

8.3 Binning strategy comparison ................................ 88
  8.3.1 Datasets ............................................... 88
  8.3.2 Promoters .............................................. 89
  8.3.3 Genes ................................................ 91
## List of Figures

2.1 Regions and metadata of a dataset consisting of two samples. 7
2.2 Examples of schema with one instance for two different types of processed data; coordinates and features are enclosed within two records. 8
2.3 Accumulation index and \textsc{cover} results with three different \textit{minAcc} and \textit{maxAcc} values. 15
2.4 Accumulation index and \textsc{cover} variants results. 16
2.5 Example of map using one sample as reference and three samples as experiment, using the \texttt{count} aggregate function. 18
2.6 Different semantics of genometric clauses due to the ordering of distal conditions; excluded regions are gray. 21

3.1 Logical Architecture of GenData 2020 24
3.2 Software Architecture of GenData 2020 25
3.3 DAG for a GMQL Query 26
3.4 Recursive descent of the DAG 27

4.1 Example of data coming from a sensor network 31
4.2 Data access operations on an array, from \textit{Paradigm4} documentation. 33
4.3 Chunking and distribution mechanism, from \textit{Paradigm4} documentation. 34
4.4 Binning strategy adopted by \textit{Paradigm4} in the genomic add-on. 41

5.1 Code example of array representation with Scala API 44
5.2 Code example of operation method implementation 45
5.3 Code example of operation method usage 46
5.4 UML diagram of SciDB arrays representation in Scala. 47
5.5 Code example of script execution 48

6.1 SciDB implementation respect to the project architecture 52
6.2 Execution scheme of the implementation module 53
Chapter 1

Introduction

A new technology for reading the DNA, called Next Generation Sequencing (NGS), is changing biological research and will change medical practice, thanks to the low-cost availability of millions of whole genome sequences of a variety of species, and most important of humans. Huge repositories of sequence information are being collected by large consortia of research laboratories by using NGS; among them, ENCODE [9], TCGA [38], the 1000 Genomes Project [8] and the 100,000 Genomes Project1. These sequences can be assembled with specific experimental data produced at the various research or clinical centers, opening new opportunities for biological discovery and for personalized medicine.

Several organizations are considering genomics at a global level. Global Alliance for genomics and Health2 is a large consortium of over 200 research institutions with the goal of supporting voluntary and secure sharing of genomic and clinical data; their work on data interoperability is producing a data conversion technology3. Google recently provided an API to store, process, explore, and share DNA sequence reads, alignments, and variant calls, using Google’s cloud infrastructure4. Parallel frameworks are used to support genomic computing, including Vertica (used by Broad Institute and NY Genome Center) and SciDB (used by NCBI for storing the data of the 1000 Genome project [8]).

So far, the bio-informatics research community has been mostly challenged by primary analysis (production of sequences in the form of shorts DNA segments, or ”reads”) and secondary analysis (alignment of reads to a reference genome and search for specific features, such as variants/muta-
tions and peaks of expression); but the most important emerging problem is the so-called *tertiary analysis*, concerned with multi-sample processing, annotation and filtering of variants, and genome browser-driven exploratory analysis [18]. While secondary analysis targets *raw data* in output from NGS processors by using specialized methods, tertiary analysis targets *processed data* in output from secondary analysis and is responsible of *sense making*, e.g., discovering how heterogeneous regions interact with each other.

This thesis has been developed inside the *GenData 2020* project, a research project founded at Politecnico di Milano in joint with European Institute for Oncology and Italian Institute for Technology under the guidance of Prof. Stefano Ceri. The GenData 2020 research project was conceived to address tertiary analysis challenge, by enabling queries and analysis of processed genomic data\(^5\). The project’s main results so far are a *Genomic Data Model* (GDM), which encodes processed data in terms of their regions and metadata, and a *GenoMetric Query Language* (GMQL) for extracting regions of interest from experiments and for computing their properties, with high-level operations for manipulating regions and for measuring their distances [24].

The most relevant and, to the best of our knowledge, original aspects of *GenData 2020* is the targeting of the projet towards heterogeneous processed data rather than raw data. World-wide genomic repositories already contain huge amounts of processed data, and actually their value stems from the certification of high-quality processing. While processed data are much smaller than raw data, they can be considered as "big data", because each processed file can contain thousands or even millions of genomic regions. In the *GenData 2020* repository structure, we show that thousands of files can be extracted from the repositories and organized within one dataset, that can be simply referenced by name in GMQL.

Another unique aspect of *GenData 2020* is the inclusion of metadata in the GDM model and of metadata management in the GMQL query language. Each dataset includes in its metadata all known information about each sample (from the sample’s preparation up to the patient’s phenotype.) GMQL progressively computes both the regions and the metadata of resulting samples. Thus, result samples of every query carry their meta-data, linking them to their contributing input samples, as an indication of data provenance; this is very powerful, e.g. for building genotype-phenotype associations.

The *GenData 2020* implementation uses cloud computing. Specifically,

\(^5\)http://www.bioinformatics.deib.polimi.it/gendata/
Version 1 of the system, developed between 2013 and 2014, translates GMQL to Pig\textsuperscript{6} in the context of Hadoop 1 HDFS, while Version 2 of the system, developed in 2015, uses the Spark\textsuperscript{7} and Flink\textsuperscript{8} frameworks as supported within Hadoop Yarn\textsuperscript{9}; the need of cloud architectures for genomics is advocated by \cite{32}. The expressive power and flexibility of GenData 2020’s data model (GDM) and query language (GMQL) are demonstrated in \cite{24}, where we show four very different genomic use cases\textsuperscript{10}.

*GenData 2020* focuses on tertiary data analysis; a similar approach is advocated by Paradigm4, a company\textsuperscript{11} whose products include genomic add-on to SciDB \cite{18}. Their toolkit offers a set of analysis instruments designed for specific datasets. The core of the thesis is design, and realize, a new implementation of GMQL v2 using SciDB. *GenData 2020* is designed to work on different kind of formats, for this reason it’s required to adapt, and extend, Paradigm4 algorithms and strategies. The implementation of the domain specific operations, like range intersection, has required the study of binning techniques. In particular we designed a new 2\textsuperscript{D} binning strategy, able to compare just the regions that have an high probability of intersection.

For the sake of clarity and understandability of the work, it is necessary to give a brief introduction of *GenData 2020*; Section 2.2 presents GDM, the data model, and Section 2.2 presents GMQL, the Genometric Query Language; then, Chapter 3 explains the general architecture of the system, including the parser, compiler, optimizer and executor.

Then, we focus on SciDB, the target engine used for implementing GDM and GMQL. Chapter 4 presents the data model and language supported by SciDB; Chapter 5 is dedicated to the presentation of an API that we have designed in order to interact with SciDB from Scala, enabling the reuse of a general framework that already supports the translation of GMQL to cloud computing platforms (Flink and Spark).

Chapter 6 is focused on the SciDB implementation, describing the architecture, the data model and the execution of a GMQL query on SciDB, and specifically describes the translation of three GMQL operations into SciDB queries (the other 7 operations of SciDB are similarly translated and their description is omitted). Then, Chapter 7 introduces the new bidimensional

\textsuperscript{6}http://pig.apache.org/
\textsuperscript{7}http://spark.apache.org/
\textsuperscript{8}http://link.apache.org/
\textsuperscript{9}http://hadoop.apache.org/docs/stable
\textsuperscript{10}Finding ChIP-seq peaks in promoter regions; finding distal bindings in transcription regulatory regions; associating transcriptomics and epigenomics; finding somatic mutations in exons.
\textsuperscript{11}Founded by last year’s Turing Award Michael Stonebraker.
binning strategy that we have designed, explaining how it can exploit the SciDB characteristics.

Chapter 8 presents several experiments; we first compare the performance of the SciDB implementation with the Spark implementation, then we compare performance of the classical monodimensional binning strategy against the new bidimensional one. Finally, Chapter 9 briefly reports the related work and Chapter 10 reports the conclusions and the future developments.
Chapter 2

Model and Language

2.1 Genomic Data Model

The Genomic Data Model (GDM) is based on the notions of datasets and samples; datasets are collections of samples, and each sample consists of two parts, the region data, which describe portions of the DNA and their features, and the metadata, which describe general properties of the sample.

2.1.1 Motivation

Processed data have a variety of file formats, and typically lack an attribute-based organization. GDM provides a schema to regions, thus it makes data self-describing, as advocated by Jim Gray [16]; however, we don’t include data into a database, so as to preserve the possibility for biologists to work with their usual file-based tools; loaders are used to copy data files to the Gendata 2020 distributed file system on the cloud at their first use.

The schema has a fixed part that guarantees the comparability of regions produced by different kinds of processings, and then a variable part describing the features produced by the various kinds of processing. Although DNA regions are strings of nucleotides\(^1\), we instead associate them with a list of one or more features, where each feature is produced by secondary data analysis.

Due to the lack of agreed standards for metadata, we model them as free attribute-value pairs; we expect metadata to include at least the experiment type, the sequencing and analysis method used for data production, the cell

\(^1\)DNA can be abstracted as a string of billions of nucleotides (represented by the letters A, C, G, T) enclosed within chromosomes (23 in humans), which are disconnected intervals of the string.
line, tissue, experimental condition (e.g., antibody target) and organism sequenced; in case of clinical studies, individual’s descriptions including phenotypes. Attributes may have multiple values (e.g., the "Disease" attribute can have both values "Cancer" and "Diabetes"). Hundreds of datasets and thousands of samples\(^2\) can be queried thanks to the GDM model.

### 2.1.2 Definition

A genomic region \( r \) is a portion of the genome defined by the quadruple of values \( \langle \text{chr}, \text{left}, \text{right}, \text{strand} \rangle \), called **region coordinates**, where \( \text{chr} \) is the chromosome, \( \text{left} \) and \( \text{right} \) are the two ends of the region along the DNA coordinates\(^3\); \( \text{strand} \) represents the direction of DNA reading\(^4\), encoded as either + or −, and can be missing (encoded as *\(^5\)). Formally, a sample \( s \) is a triple \( \langle \text{id}, \text{R}, \text{M} \rangle \) where:

- \( \text{id} \) is the sample identifier of type \text{long}.

- \( \text{R} \) is the set of regions of the sample, built as pairs \( \langle c, \text{f} \rangle \) of coordinates \( c \) and features \( \text{f} \); coordinates are arrays of four fixed attributes \( \text{chr}, \text{left}, \text{right}, \text{strand} \) which are respectively typed \text{string}, \text{long}, \text{long}, \text{string}; features are arrays of typed attributes; we assume attribute names of features to be different, and their types to be any of \text{string}, \text{int}, \text{long}, \text{double}, \text{boolean} (GDM types are available both in Java, Scala, and in the Flink, Spark, SciDB and Pig frameworks). The **region schema** of \( s \) is the list of attribute names used for the identifier, the coordinates and the features.

- \( \text{M} \) is the set of metadata of the sample, built as attribute-value pairs \( \langle \text{a}, \text{v} \rangle \), where we assume the type of each value \( \text{v} \) to be \text{string}. The same attribute name \( \text{a} \) can appear in multiple pairs of the same sample (in which case, we say that \( \text{a} \) is multi-valued).

A **dataset** is a collection of samples with the same region schema and with features having the same types; sample identifiers are unique within each

---

\(^2\)We currently store in *GenData 2020* most of ENCODE [9] and TCGA [38] processed data.

\(^3\)Species are associated with their reference genome; DNA samples are aligned to these references, hence referred to the same system of coordinates; for humans, several references were progressively defined, the latest reference is \( \text{h19} \).

\(^4\)DNA is made of two strands rolled-up together in anti-parallel directions, i.e., they are read in opposite directions by the biomolecular machinery of the cell.

\(^5\)According to the UCSC notation, we use 0-based, half-open inter-base coordinates, i.e., the considered genomic sequence is \( [\text{left}, \text{right}] \). Left and right ends can be identical (e.g., when the region represents a single nucleotide polymorphism).
dataset. Each dataset is typically produced within the same project (either at a genomic research center or within an international consortium) by using the same technology and tools, but with different experimental conditions, described by metadata.

### 2.1.3 Examples

Each dataset is stored within *GenData 2020* using two tables, one for regions and one for metadata; an example of the two tables for representing a particular experiment, called *ChIP-seq*, is shown in Fig.2.1. Note that the region value has an attribute `P_VALUE` of type `float` (representing how significant is the calling of the peak of expression in that genomic region); note also that the `ID` attribute is present in both tables; it provides a many-to-many connection between regions and metadata of a sample; e.g., sample 1 has 3 regions and 4 metadata attributes, sample 2 has 2 regions and 3 metadata attributes\(^6\). The regions of the two samples are within chromosomes 1 and 2 of the DNA, and both are not stranded.

While the above example is simple, GDM supports the schema encoding of any processed data type, e.g., files for mutations, ChIP-seq, DNA-seq, RNA-seq, ChIA-PET, VCF, and SAM/BAM formats. We use GDM also

---

\(^6\)Note that the quadruple `(id, chr, left, right)` is not a key of the region table (because a sample can have multiple regions with the same coordinates), and similarly the pair `(id, attribute)` is not a key of the metadata table (because metadata attributes can be multi-valued).
for modeling annotations, i.e. regions of the genome with known properties (such as genes, with their exons and introns). Schema encodings and one exemplar instance of mutations and RNA-seq data samples are described in Fig.2.2.

### 2.2 Genometric Query Language

A GMQL query (or program) is expressed as a sequence of GMQL operations with the following structure:

\[
\text{<variable>} = \text{OPERATION(<parameters>)} \text{<variables>}
\]

where each variable stands for a GDM dataset. Operations are either unary (with one input variable), or binary (with two input variables), and construct one result variable.

#### 2.2.1 General properties

GMQL operations form a closed algebra: results are expressed as new datasets derived from their operands. All operations produce a result dataset consisting of several samples, whose identifiers are either inherited by the operands or generated by the operation. Each operation separately applies to metadata and to regions; the region-based part of an operation computes the result regions, the metadata part of the operation computes the associated metadata so as to trace the provenance of each resulting sample; identifiers preserve the many-to-many mapping of regions and metadata as discussed in section 2.1.3.

Most GMQL operations, although defined upon two connected data structures, are extension of classic relational algebra operations, twisted...
to the needs of genomics; they are denoted as relational. Three domain-specific operations, called \texttt{COVER}, (distal) \texttt{JOIN} and \texttt{MAP}, significantly extend the expressive power of classic relational algebra.

The main design principles of GMQL are \textit{relational completeness} and \textit{orthogonality}. Completeness is guaranteed by the fact that classical algebraic manipulations are all supported, suitably extended and adapted to comply with region-based calculus. Orthogonality is achieved because no operator can be defined as a suitable expression of all other operators; note that the classic abstractions of \textit{grouping} is supported, with the same semantics, in the unary operations \texttt{GROUP} and \texttt{COVER}, and similarly \textit{joining} is supported, with the same semantics, in the binary operations \texttt{JOIN}, \texttt{MAP} and \texttt{DIFFERENCE}.

Compared with languages which are currently in use by the bio-informatic community, GMQL is \textit{declarative} (it specifies the structure of the results, leaving its computation to each operation’s implementation) and \textit{high-level} (one GMQL query typically substitutes for a long program which embeds calls to region manipulation libraries); the progressive computation of variables resembles other algebraic languages (e.g. Pig Latin\textsuperscript{7}). For all these features, GMQL may inspire a change of paradigm in genomics, along a direction that was indicated long ago by Edward T. Codd’s seminal papers.

### 2.2.2 Predicates evaluation

Parameters of several operations include predicates, used to select and join samples; predicates are built by arbitrary boolean expressions of simple predicates, as it is customary in relational algebra. The region attributes can refer positionally to the schema, i.e., \$0$ denotes the first attribute \$1$ to the second, and so on. Predicates are either evaluated in the context of regions or of metadata, as follows:

- Predicates on regions have a classic interpretation: they select the regions where the predicate is true. Legal predicates must use the attributes in the region’s schema; when a predicate is illegal, the query is also illegal, and compilation fails\textsuperscript{8}. The evaluation of predicates involving two or more regions (essentially join predicates) is defined only when regions have compatible strands; positive and negative strands are incompatible, but they are both compatible with a missing strand.

\textsuperscript{7}http://pig.apache.org/

\textsuperscript{8}Region predicates may include metadata attributes, but in such case they are legal iff the metadata attribute is single-valued and not null, and invalid otherwise; in such case, for a given sample, metadata attributes are equivalent to constant values.
• Predicates on metadata have an existential interpretation over samples: they select the entire sample if it contains some metadata attributes such that the predicate evaluation on their values is true. Formally, for each sample, a simple predicate \( p \) expressed as \((A \text{ comp } V)\) on metadata \( M \) is defined as:

\[
p \iff \exists (a_i, v_i) \in M : (a_i = A) \land (v_i \text{ comp } V)
\]

When a predicate on metadata uses an attribute which is missing, the predicate is unknown; we use three-value (i.e. true, false, unknown) logic for metadata predicates \( p \), and we select samples \( s \) for which \( p(s) \) is true given the above interpretation. The special predicate \( \text{missing}(A) \) is true if the attribute \( A \) is not present in \( M \).

2.2.3 Relational GMQL operations

We next describe relational operations; they include six unary operations (\texttt{SELECT}, \texttt{PROJECT}, \texttt{EXTEND}, \texttt{MERGE}, \texttt{GROUP} and \texttt{SORT}) and two binary operations (\texttt{UNION} and \texttt{DIFFERENCE}).

Select

\[
\langle S_2 \rangle = \text{SELECT}([\text{SJ}_\text{clause}];[<\text{pm}>];[<\text{pr}>]) \langle S_1 \rangle
\]

It keeps in the result all the samples which existentially satisfy the metadata predicate \(<\text{pm}>\) and then selects those regions of selected samples which satisfy the region predicate \(<\text{pr}>\); a sample is legal also when it contains no regions as result of a selection. Identifiers of selected samples of the operand \( S_1 \) are assigned to the result.

Semi-join clauses are used to further select samples as effect of simple metadata predicates, defined as: \( p: \langle A \rangle \text{ AS } \langle \text{extA} \rangle \text{ IN } \langle \text{extV} \rangle \). Each clause corresponds to a predicate \( p(a_i, a_j) \) (with \( a_i \) mapped to \( A \), \( a_j \) to \( \text{extA} \)); the predicate is true for a given sample \( s_i \) with attribute \( a_i \) if there exists a sample in the variable denoted as \( \text{ExtV} \) with an attribute \( a_j \) having at least one common value. Formally, if \( M_E \) denotes the metadata of samples of \( \text{ExtV} \), then:

\[
p \iff \exists (a_i, v_i) \in M_i, (a_j, v_j) \in M_E : (a_i = a_j) \land (v_i = v_j)
\]

A semi-join clause can be constructed as the conjunction of the above simple metadata predicates that refer to the same variable \( \text{ExtV} \). Semi-joins are used to connect variables, e.g. in the example below:
OUT = SELECT(Antibody IN EXP2.Antibody) EXP1

samples of EXP1 are selected only if they have the same Antibody value as some samples of EXP2.

**Project**

\[
<S2> = \text{PROJECT}(<A_{m1}> [\text{AS } f_1], \ldots, <A_{mn}> [\text{AS } f_n] ; \\
<a_{r1}> [\text{AS } f_1], \ldots, <a_{rn}> [\text{AS } f_n]) <S1>
\]

It keeps in the result the metadata (Am) and region (Ar) attributes expressed as parameters\(^9\). It can also be used to build new attributes as scalar expressions fi (e.g., for metadata the age from the birthday; for regions, the length of a region as the difference between its right and left ends). If the name of existing attributes are used, the operation updates region attributes to new values and assigns to metadata attributes new values. Identifiers of the operand \(S1\) are assigned to the result.

**Extend**

\[
<S2> = \text{EXTEND} ( <A_{m1} > \text{ AS } g_1, \ldots, <A_{mn} > \text{ AS } g_n ) <S1>
\]

It generates new metadata attributes Am as result of aggregate functions \(g\) applied to region attributes; identifiers of the operand \(S1\) are assigned to the result. The supported aggregate functions include COUNT, BAG, MIN, MAX (applicable to any type) and SUM, AVG, MEDIAN, STD (applicable to numeric types). E.g. in the example below:

OUT = EXTEND (RegionCount AS COUNT, MinP AS MIN(p\_value)) EXP

for each sample of EXP two new metadata attributes are computed, RegionCount as the number of regions, and MinP as the minimum \(p\_value\).

**Merge**

\[
<S2> = \text{MERGE} <S1>
\]

It builds a dataset consisting of a single sample having as regions all the regions of the input samples and as metadata the union of all the attribute-values of the input samples.

\(^9\)A syntactic variant (using the keywords ALL BUT) allows to specify only the attributes that are kept in the result; this variant is very useful with datasets having hundreds of metadata.
Group

\(<S2> = \text{GROUP}([<\text{Am1}>..<\text{Amn}>;\\<\text{Gm1}> \text{ AS } <\text{g1}>, .., <\text{Gmnn}> \text{ AS } <\text{gn}>] [;]\\[<\text{Ar1}>..<\text{Arn}>;\\<\text{Gr1}> \text{ AS } <\text{g1}>, .., <\text{Grn}> \text{ AS } <\text{gn}>]) <S1>;\)

It is used for grouping both regions and metadata according to distinct values of the grouping attributes. For what concerns metadata, each distinct value of the grouping attributes is associated with an output sample, with a new identifier explicitly created for that sample; samples having missing values for any of the grouping attributes are discarded. The metadata of output samples, each corresponding to a given group, are constructed as the union of metadata of all the samples contributing to that group; in this way, metadata include attributes storing the grouping values, that are common to each sample in the group. New grouping attributes \(\text{Gm}\) are added to output samples, storing the results of aggregate function evaluations over each group. Examples of typical metadata grouping attributes are the classification of patients (e.g., as cases or controls) or their disease values.

When the grouping attribute is multi-valued, samples are partitioned by each subset of their distinct values (e.g., samples with a "Disease" attribute set both "Cancer" and "Diabetes" are within a group which is distinct from the groups of the samples with only one value, either "Cancer" or "Diabetes"). Formally, two samples \(s_i\) and \(s_j\) belong to the same group, denoted as \(s_i \gamma_{A}s_j\), if and only if they have exactly the same set of values for every grouping attribute \(A\), i.e.

\[ s_i \gamma_{A}s_j \iff \{v : \exists (A, v) \in M_i\} = \{v : \exists (A, v) \in M_j\} \]

Given this definition, grouping has important properties:

- reflexive: \(s_i \gamma_{A}s_i\)
- commutative: \(s_i \gamma_{A}s_j \iff s_j \gamma_{A}s_i\)
- transitive: \(s_i \gamma_{A}s_j \land s_k \gamma_{A}s_i \iff s_k \gamma_{A}s_j\)

When grouping applies to regions, it includes by default grouping attributes \(\text{chr, left, right}\); this choice corresponds to the biological application of removing duplicate regions, i.e. regions with the same coordinates, possibly resulting from other operations, and ensures that the result is a legal GDM instance (with missing \(\text{strand}\).) Other attributes may be added to grouping
attributes (e.g., \textit{strand} or \textit{RegionType}); aggregate functions can then be applied to each group. The resulting schema includes the attributes used for grouping and possibly new attributes used for the aggregate functions. The following example is used for calculating the minimum \texttt{p.value} of duplicate regions:

\texttt{OUT = GROUP (p.value AS MIN(p.value)) EXP}

**Order**

\texttt{<S2> = ORDER([DESC]<Am1>, \ldots, [DESC]<Amn>}
\texttt{[; TOP <k> | TOPG <k>][;]}
\texttt{[DESC]<Ar1>, \ldots, [DESC]<Arn>}
\texttt{[; TOP <k> | TOPG <k>]}) <S1>;

It orders either samples, or regions, or both of them; order is \textit{ascending} as default, and can be turned to \textit{descending} by an explicit indication. Sorted samples or regions have a new attribute \texttt{Order}, added to either metadata, or regions, or both of them; the value of \texttt{Order} reflects the result of the sorting. Identifiers of the samples of the operand \texttt{S1} are assigned to the result. The clause \texttt{TOP <k>} extracts the first \(k\) samples, the clause \texttt{TOPG <k>} implicitly considers the grouping by identical values of the first \(n-1\) ordering attributes and then selects the first \(k\) samples of each group. The operation:

\texttt{OUT = SORT (RegionCount, TOP 5; MutationCount, TOP 7) EXP}

extracts the first 5 samples on the basis of their region counter and then, for each of them, 7 regions on the basis of the mutation counter.

**Union**

\texttt{<S3> = UNION[_LEFT] <S1> <S2>}

It is used to integrate possibly heterogeneous samples of two datasets within a single dataset; each sample of both input datasets contributes to one sample of the result with identical metadata and merged region schema. The merging regards just the region features and is done by merging the feature attributes of the first dataset with the feature attributes of the second
dataset when they have identical name and type; instances are built accordingly\(^\text{10}\). New identifiers are assigned to each sample, and missing attribute values are set to NULL.

With the LEFT option, the operation assigns to the result the schema of the left operand \(S_1\); the values of the attributes of \(S_2\) that have the same name and type as some attribute of \(S_1\) are preserved, while all attributes of \(S_2\) that do not appear in the schema of \(S_1\) are eliminated.

**Difference**

\[
<S3> = \text{DIFFERENCE} \left[ ([\text{JOINBY} <Am11> == <Am21>, \ldots, <Am1n> == <Am2n>]) \right] <S1> <S2>;
\]

This operation produces a sample in the result for each sample of the first operand \(S_1\), with identical metadata. It considers all the regions of the second operand, that we denote as *negative regions*; for each sample \(s_1\) of \(S_1\), it includes in the result sample those regions which do not intersect with any negative region. Identifiers of the operand \(S_1\) are assigned to the result.

When the **JOINBY** clause is present, for each sample \(s_1\) of the first dataset \(S_1\) we consider as negative regions only the the regions of the samples \(s_2\) of \(S_2\) that satisfy the join condition (i.e., such that \(p(s_1, s_2)\) is true)\(^\text{11}\). We formally define a simple equi-join predicate \(p : a_i = a_j\), but the generalization to conjunctions of simple predicates is straightforward. The predicate \(p\) is true for given samples \(s_i\) and \(s_j\) with attributes \(a_i\) and \(a_j\) iff the two attributes share at least one value, e.g.:

\[
p \iff \exists \langle a_i, v_i \rangle \in M_i, \langle a_j, v_j \rangle \in M_j : (a_i = a_j) \land (v_i = v_j)
\]

### 2.2.4 Domain-specific GMQL operations

We next focus on *domain-specific* operations, which are more specifically responding to genomic management requirements: the unary operation **COVER** and the binary operations **MAP** and **JOIN**.

\(^{10}\)If two attributes have the same name and different types, they are considered as different and disambiguated in the schema by using a *dot notation*, where the dataset name is used as prefix to the attribute name to indicate their provenance from the first or second operand, respectively.

\(^{11}\)When the join condition is between homonym attributes, syntactic disambiguation is based on the dot notation, using the dataset name for disambiguation.
Figure 2.3: Accumulation index and COVER results with three different minAcc and maxAcc values.

Cover

\[ \text{COVER} = \text{COVER}[\_FLAT|\_SUMMIT|\_HISTOGRAM] \]
\[ ( [ \text{<minAcc>}, \text{<maxAcc>} ; ] \]
\[ [ \text{<Ar1>} \text{ AS } \text{<g1>}, \ldots, \text{<Arn>} \text{ AS } \text{<gn>} ; ] \]
\[ [ \text{GROUPBY} \text{<Am1>}, \ldots, \text{<Amn>} ] ) \text{<S1>}; \]

The COVER operation responds to the need of computing properties that reflect region’s intersections, for example to compute a single sample from several samples which are replicas of the same experiment or for dealing with overlapping regions (as, by construction, resulting regions are not overlapping.)

Let us initially consider the COVER operation with no grouping; in such case, the operation produces a single output sample, and all the metadata attributes of the contributing input samples \( s_1 \) are assigned to the resulting sample \( s \). Regions of the result sample are built from the regions of samples in \( S_1 \) according to the following condition:

- Each resulting region \( r \) in \( S_2 \) is the contiguous intersection of at least \( \text{minAcc} \) and at most \( \text{maxAcc} \) contributing regions \( r_i \) in the samples of \( S_1 \); \( \text{minAcc} \) and \( \text{maxAcc} \) are called accumulation indexes\(^{12}\).

\(^{12}\)The keyword ANY can be used as \( \text{maxAcc} \), and in this case no maximum is set (it is equivalent to omitting the \( \text{maxAcc} \) option); the keyword ALL stands for the number of samples of the operand, and can be used both for \( \text{minAcc} \) and \( \text{maxAcc} \); these can also be expressed as arithmetic expressions built by using ALL (e.g., ALL-3, ALL+2, ALL/2); cases when \( \text{maxAcc} \) is greater than ALL are relevant when the input samples include overlapping regions.
Resulting regions may have new attributes $Ar$, calculated by means of aggregate expressions over the attributes of the contributing regions. Jaccard Indexes\textsuperscript{13} are standard measures of similarity of the contributing regions $r_i$, added as default attributes. When a GROUP\_BY clause is present, the samples are partitioned by groups, each with distinct values of grouping metadata attributes, and the cover operation is separately applied to each group, yielding to one sample in the result for each group, as discussed in section 2.2.3.

For what concerns variants:

- The _HISTOGRAM variant returns the nonoverlapping regions contributing to the cover, each with its accumulation index value, which is assigned to the AccIndex region attribute.

- The _FLAT variant returns the union of all the regions which contribute to the COVER (more precisely, it returns the contiguous region that starts from the first end and stops at the last end of the regions which would contribute to each region of the COVER).

- The _SUMMIT variant returns only those portions of the result regions of the COVER where the maximum number of regions intersect (more

\textsuperscript{13}The JaccardIntersect index is calculated as the ratio between the lengths of the intersection and of the union of the contributing regions; the JaccardResult index is calculated as the ratio between the lengths of the result and of the union of the contributing regions.
precisely, it returns regions that start from a position where the number of intersecting regions is not increasing afterwards and stops at a position where either the number of intersecting regions decreases, or it violates the max accumulation index).

Example  Fig.2.3 shows three applications of the COVER operation on three samples, represented on a small portion of the genome; the figure shows the values of accumulation index and then the regions resulting from setting the minAcc and maxAcc parameters respectively to (2, 2), (1, 2), and (2, 3).

The following COVER operation produces output regions where at least 2 and at most 3 regions of EXP overlap, having as resulting region attributes the min p_value of the overlapping regions and their Jaccard indexes; the result has one sample for each input "CellLine".

\[
\text{RES} = \text{COVER}(2, 3; \text{p_value AS MIN(p_value)}
\text{GROUP_BY CellLine}) \text{ EXP}
\]

Map

\[
<S3> = \text{MAP } [(\text{JOINBY } <\text{Am}_1>, \ldots, <\text{Am}_n>)]
\text{(}<\text{Ar}_1> \text{ AS } <\text{g}_1>, \ldots, <\text{Ar}_n> \text{ AS } <\text{g}_n>) ] <\text{S}_1> <\text{S}_2>;
\]

MAP is a binary operation over two datasets, respectively called reference and experiment. Let us consider one reference sample, with a set of reference regions; the operation computes, for each sample in the experiment, aggregates over the values of the experiment regions that intersect with each reference region; we say that experiment regions are mapped to reference regions. The operation produces a matrix structure, called genomic space, where each experiment sample is associated with a row, each reference region with a column, and the matrix entries is a vector of numbers\(^{14}\). Thus, a MAP operation allows a quantitative reading of experiments with respect to the reference regions; when the biological function of the reference regions is not known, the MAP helps in extracting the most interesting regions out of many candidates.

We first consider the basic MAP operation, without JOINBY clause. For a given reference sample \(s_1\), let \(R_1\) be the set of its regions; for each sample \(s_2\) of the second operand, with \(s_2 = \langle i_{d_2}, R_2, M_2 \rangle\) (according to the GDM notation), the new sample \(s_3 = \langle i_{d_3}, R_3, M_3 \rangle\) is constructed; \(i_{d_3}\) is generated.

\(^{14}\)Biologists typically consider the transposed matrix, because there are fewer experiments (on columns) than regions (on rows). Such matrix can be observed using heat maps, and its rows and/or columns can be clustered to show patterns.
Figure 2.5: Example of map using one sample as reference and three samples as experiment, using the count aggregate function.

from id1 and id2, the metadata M3 are obtained by merging metadata M1 and M2, and the regions \( R_3 = \{ \langle c_3, f_3 \rangle \} \) are created such that, for each region \( r_1 \in R_1 \), there is exactly one region \( r_3 \in R_3 \), having the same coordinates (i.e., \( c_3 = c_1 \)) and having as features \( f_3 \) obtained as the concatenation of the features \( f_1 \) and the new attributes computed by the aggregate functions \( g \) specified in the operation; such aggregate functions are applied to the attributes of all the regions \( r_2 \in R_2 \) having a non-empty intersection with \( r_1 \). A default aggregate count counts the number of regions \( r_2 \in R_2 \) having a non-empty intersection with \( r_1 \). The operation is iterated for each reference sample, and generates a sample-specific genomic space at each iteration.

When the JOINBY clause is present, for each sample \( s_1 \) of the first dataset \( S_1 \) we consider the regions of the samples \( s_2 \) of \( S_2 \) that satisfy the join condition. Syntactically, the clause consists of a list of attribute names, which are homonyms from the schemas of \( S_1 \) and of \( S_2 \); the strings "left" or "right" that may be present as prefixes of attribute names as result of binary operators are not considered for detecting homonyms.

**Example** Fig.2.5 shows the effect of this MAP operation on a small portion of the genome; the input consists of one reference sample and three mutation experiment samples, the output consists of three samples with the same regions as the reference sample, whose features corresponds to the number of mutations which intersect with those regions. The result can be interpreted as a \((3 \times 3)\) genome space.

In the example below, the MAP operation counts how many mutations
occur in known genes, where the dataset \( \text{EXP} \) contains DNA mutation regions and \( \text{GENES} \) contains the genes.

\[
\text{RES} = \text{MAP(count)} \text{ GENES EXP};
\]

**Join**

\[
\text{<S3>} = \text{JOIN} ([\text{JOINBY} \text{ <Am1>}, \ldots, \text{<Amn>} \}; \\
[\text{<genometric-pred>} \}; \\
[\text{<coord-gen>}]) \text{ <S1> <S2>};
\]

The **JOIN** operation applies to two datasets, respectively called *anchor* (the first one) and *experiment* (the second one), and acts in two phases (each of them can be missing). In the first phase, pairs of samples which satisfy the **JOINBY** predicate (also called meta-join predicate) are identified; in the second phase, regions that satisfy the **genometric-pred** are selected. The meta-join predicate allows selecting sample pairs with appropriate biological conditions (e.g., regarding the same cell line or antibody); syntactically, it is expressed as a list of homonym attributes from the schemes of \( S_1 \) and \( S_2 \), as previously. The genometric join predicate allows expressing a variety of distal conditions, needed by biologists. The anchor is used as startpoint in evaluating genometric predicates (which are not symmetric). The join result is constructed as follows:

- The meta-join predicates initially selects pairs \( s_1 \) of \( S_1 \) and \( s_2 \) of \( S_2 \) that satisfy the join condition. If the clause is omitted, then the Cartesian product of all pairs \( s_1 \) of \( S_1 \) and \( s_2 \) of \( S_2 \) are selected. For each such pair, a new sample \( s_{12} \) is generated in the result, having an identifier \( id_{12} \), generated from \( id_1 \) and \( id_2 \), and metadata given by the union of metadata of \( s_1 \) and \( s_2 \).

- Then, the genometric predicate is tested for all the pairs \( \langle r_i, r_j \rangle \) of regions, with \( r_i \in s_1 \) and \( r_j \in s_2 \), by assigning the role of anchor region, in turn, to all the regions of \( s_1 \), and then evaluating the join condition with all the regions of \( s_2 \). From every pair \( \langle r_i, r_j \rangle \) that satisfies the join condition, a new region is generated in \( s_{12} \).

From this description, it follows that the join operation yields to results that can grow quadratically both in the number of samples and of regions; hence, it is the most critical GMQL operation from a computational point of view.

Genometric predicates are based on the *genomic distance*, defined as the number of bases (i.e., nucleotides) between the closest opposite ends of
two regions, measured from the right end of the region with left end lower coordinate.\textsuperscript{16} A genometric predicate is a sequence of distal conditions, defined as follows:

- **UP/DOWN** denotes the *upstream* and *downstream* directions of the genome. They are interpreted as predicates that must hold on the region \( s_2 \) of the experiment; **UP** is true when \( s_2 \) is in the *upstream genome* of the anchor region\textsuperscript{17}. When this clause is not present, distal conditions apply to both the directions of the genome.

- **MD(K)** denotes the *minimum distance* clause; it selects the \( K \) regions of the experiment at minimal distance from the anchor region. When there are ties (i.e., regions at the same distance from the anchor region), regions of the experiment are kept in the result even if they exceed the \( K \) limit.

- **DLE(N)** denotes the *less-equal distance* clause; it selects all the regions of the experiment such that their distance from the anchor region is less than or equal to \( N \) bases\textsuperscript{18}.

- **DGE(N)** denotes the *greater-equal distance* clause; it selects all the regions of the experiment such that their distance from the anchor region is greater than or equal to \( N \) bases.

Genometric clauses are composed by strings of distal conditions; we say that a genometric clause is *well-formed* iff it includes the *less-equal distance* clause; we expect all clauses to be well formed, possibly because the clause **DLE(max)** is automatically added at the end of the string, where \( \text{max} \) is a problem-specific maximum distance.

\textsuperscript{16}Note that with our choice of interbase coordinates, intersecting regions have distance less than 0 and adjacent regions have distance equal to 0; if two regions belong to different chromosomes, their distance is undefined (and predicates based on distance fail).

\textsuperscript{17}*Upstream* and *downstream* are technical terms in genomics, and they are applied to regions on the basis of their *strand*. For regions of the *positive strand* (or for *unstranded regions*), **UP** is true for those regions of the experiment whose right end is lower than the left end of the anchor, and **DOWN** is true for those regions of the experiment whose left end is higher than the right end of the anchor. For the *negative strand*, ends and disequations are exchanged.

\textsuperscript{18}**DLE(-1)** is true when the region of the experiment overlaps with the anchor region; **DLE(0)** is true when the region of the experiment is adjacent to or overlapping with the anchor region.
Example  The following strings are legal genometric predicates:

DGE(500), UP, DLE(1000), MD(1)
DGE(500000), UP, DLE(1000000), (S1.left - S2.left > 600)
DLE(2000), MD(1), DOWN
MD(100), DLE(3000)

Note that different orderings of the same distal clauses may produce different results; this aspect has been designed in order to provide all the required biological meanings.

Examples In fig.2.6 we show an evaluation of the following two clauses relative to an anchor region: A: MD(1), DGE(100); B: DGE(100), MD(1).
In case A, the MD(1) clause is computed first, producing one region which is next excluded by computing the DGE(100) clause; therefore, no region is produced. In case B, the DGE(100) clause is computed first, producing two regions, and then the MD(1) clause is computed, producing as result one region.19

Similarly, the clauses A: MD(1), UP and B: UP, MD(1) may produce different results, as in case A the minimum distance region is selected regardless of streams and then retained iff it belongs to the upstream of the anchor, while in case B only upstream regions are considered, and the one at minimum distance is selected.

Next, we discuss the structure of resulting samples. Assume that regions $r_i$ of $s_i$ and $r_j$ of $s_j$ satisfy the genometric predicate, then a new region $r_{ij}$ is created, having merged features obtained by concatenating the feature attributes of the first dataset with the feature attributes of the second dataset.

19The two queries can be expressed as: produce the minimum distance region iff its distance is less than 100 bases and produce the minimum distance region after 100 bases.
as discussed in section 2.2.3. The coordinates $c_{ij}$ are generated according to the `coord-gen` clause, which has four options:

1. **LEFT** assigns to $r_{ij}$ the coordinates $c_i$ of the anchor region.
2. **RIGHT** assigns to $r_{ij}$ the coordinates $c_j$ of the experiment region.
3. **INT** assigns to $r_{ij}$ the coordinates of the intersection of $r_i$ and $r_j$; if the intersection is empty then no region is produced.
4. **CAT** (also: **CONTIG**) assigns to $r_{ij}$ the coordinates of the concatenation of $r_i$ and $r_j$ (i.e., the region from the lower left end between those of $r_i$ and $r_j$ to the upper right end between those of $r_i$ and $r_j$).

**Example** The following join searches for those regions of particular ChIP-seq experiments, called histone modifications (HM), that are at a minimal distance from the transcription start sites of genes (TSS), provided that such distance is greater than 120K bases. Note that the result uses the coordinates of the experiment.

```
RES = JOIN((MD(1), DLE(12000); RIGHT) TSS HM;
```

---

20 If the operation applies to regions with the same strand, the result is also stranded in the same way; if it applies to regions with different strands, the result is not stranded.

21 This query is used in the search of *enhancers*, i.e., parts of the genome which have an important role in gene activation.
Chapter 3

Architecture of Gendata 2020

The translation and execution of GMQL queries is very complex, as it adds domain-specific features to a relationally complete language. In the first version of GenData 2020, we translate GMQL into Pig, and then execute a Pig program, in which the domain-specific aspects of GMQL are encoded into user-defined functions. Pig programs, in turn, are translated into map-reduce programs, whereas user-defined functions are executed by dedicated reducer nodes. This architecture has produced a stable but rather unflexible implementation; in particular, the syntax-directed translation of GMQL operations into Pig does not allow for optimizations.

The second version of GenData 2020, here described, is centered on an abstract representation of queries, called operator DAG for genomics; as illustrated in fig.3.1, the GMQL compiler produces an operator DAG from a GMQL program, and then the execution consists of a recursive traversal of the DAG. This architecture allows for powerful optimizations, that can be either performed at the logical level (by equivalence transformations of the DAG) or at the physical level; physical optimizations depend on the deployment platform used for the implementation and also on the physical properties of the datasets.

3.1 Software layers

The logical architecture of fig.3.1 is used in the context of the software architecture of fig.3.2. The most important difference between the two architectures is the introduction of a GMQL API, which can be invoked from a variety of systems, including the GMQL parser which processes GMQL
queries line-by-line; in the fig.3.2, we show that the API can be also invoked from systems which are typically used by bio-informaticians (such as R or Galaxy) and can be directly invoked from applications (e.g., Scala programs).

The API provides an interface to each GMQL operation; in addition, it also provides operations for \textit{loading} the datasets and for \textit{storing} them. Loading occurs during the execution of the first operation that mentions a dataset in the repository, and it is optimized so as to load the data which is strictly needed by the query (because data loading is very critical in cloud computing systems). Data storing is explicitly requested by \texttt{STORE} operations that refer to specific GMQL variables.

The architecture supports as well a flexible mapping to execution environments, currently we have deployed \textit{GenData 2020} on both Spark, and Flink, two emerging frameworks for big data management in cloud-based architectures; as proposed with this work, the architecture allows new implementations based on different technologies.

Fig.3.2 schematically illustrates the three levels of software interfaces; the API supports calls to GMQL operations (\texttt{SELECT, PROJECT, MAP}). The DAG is a collection of classes for DAG operators, which are next discussed. Finally, the execution introduces calls to the framework operators (in the fig., Flink operators).

3.2 Internal representation

DAG operators apply separately to metadata and to regions, hence each GMQL operator is mapped to two (one for region and one for metadata) or more DAG operators, as illustrated in Table 3.1; due to the language orthogonality, most GMQL operators require the introduction of specific DAG operators; however, the \texttt{MetaJoin} and \texttt{MetaGroup} clauses of GMQL are highly reused by many different operators as much as some metadata operators like CombineMD. The most relevant feature of DAGs is that they illustrate the dependencies between DAG operators; every DAG node

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure3.png}
\caption{Logical Architecture of GenData 2020}
\end{figure}
includes as parameters the pointers to the DAG nodes that it depends from.

The entire translation of GMQL operations into the DAG nodes requires 28 nodes in total. Both the Flink and Spark implementation includes all of them and are able to execute a complete GMQL query. The aim of the work documented in this thesis was to develop all the 28 nodes on SciDB.

Fig. 3.3 shows the DAG constructed for the following query, which includes five SELECTs, two JOINs, and one DIFFERENCE; in this example, all samples are extracted from global datasets, named PEAKS and ANNOTATIONS.

```
AC = SELECT(Antibody == 'AcK27') PEAKS;
ME1 = SELECT(Antibody == 'me1K4') PEAKS;
ME3 = SELECT(Antibody == 'me3K4') PEAKS;
GENES = SELECT(Feature == 'genes' AND Prov == 'UCSC') ANNOTATIONS;
PE = JOIN(DLE(0); CONTIG) AC ME1;
E = DIFFERENCE() PE ME3;
AX = JOIN(MD,DLE(100000); LEFT) E GENES;
R = SELECT(LogFCgene > 1.5 AND LogFCen >1.5) AX;
STORE R;
```

Note that the query variables are either extracted from the file system (in this case the PEAK and ANNOTATION variables) or defined by operations before being used by other operations, and such precedence relationship determines
Figure 3.3: DAG for a GMQL Query
the edges of DAGs. Note also that region loaders are invoked after the loading of the corresponding metadata, so that they load just the regions of selected samples.

The DAG execution is triggered by the GMQL operation \textbf{STORE}. Consider queries with a single \texttt{STORE} operation\footnote{When a DAGs has multiple \texttt{STORE} operations, at least one of them does not depend on any stored variable; this induces a partial order of materializations.}; we denote the set of nodes which are reachable from \texttt{StoreMD} as "meta-DAG". The DAG has two roots, called \texttt{StoreMD} and \texttt{StoreRD}; the translator adds a \textit{dummy root} node which has \texttt{StoreMD} and \texttt{StoreRD} as direct precedences, and then invokes the execution on such dummy root. The execution of any node cannot occur

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{GMQL Operation} & \textbf{DAG Operators} \\
\hline
SELECT & SelectMD, SelectRD, SemiJoinMD \\
\hline
PROJECT & ProjecMD, ProjectRD \\
\hline
EXTEND & ExtendMD, AggregateMD \\
\hline
MERGE & MergeMD, MergeRD \\
\hline
GROUP & GroupMD, GroupRD \\
\hline
ORDER & OrderMD, OrderRD, PurgeRD \\
\hline
UNION & UnionMD, UnionRD \\
\hline
DIFFERENCE & JoinMD, DifferenceRD \\
\hline
COVER & GroupMD, MergeMD, GenometricCoverRD \\
\hline
MAP & JoinMD, CombineMD, GenometricMapRD \\
\hline
JOIN & JoinMD, CombineMD, GenometricJoinRD \\
\hline
STORE & StoreMD, StoreRD \\
\hline
\end{tabular}
\caption{DAG Operators used for each GMQL operations.}
\end{table}
until all the nodes from which it depends are executed, and this induces a partial ordering of node executions. Note that the DAG creates only one node for each query variable, and therefore the operation that computes a variable is executed once, even if a variable is used multiple times in a query. A simple case is illustrated in Fig. 3.4: the execution of node $N_4$ recursively invokes the execution of $N_3$ and then of $N_1$ and $N_2$; such nodes directly invoke loaders of two datasets $DS_1$ and $DS_2$. Then, the execution of $N_3$ can be concluded, and finally the execution of $N_4$ can be concluded.
Chapter 4

SciDB

SciDB [10, 34] is a new open-source data management system intended primarily for use in application domains that involve very large scale array data; for example scientific applications such as astronomy, remote sensing and climate modelling, bio-science information management, as well as commercial applications such as risk management systems in the financial services sector, and the analysis of web log data.

This chapter will describe in detail this technology. At first we give a brief introduction of the background and the motivations that have led to the realization of SciDB, then we describe the array data model and the storage management adopted. Section 4.4 is about the query languages provided to communicate and use the system. Finally we present the genomic add-on developed by Paradigm4, describing the data structures used to represent genomic data into SciDB and their optimization strategies.

Some of the following sections comes from official SciDB documentation provided by Paradigm4, the sources will be cited in the text.

4.1 Background

The XLDB-1 (Extremely Large Data Base) workshop [4] in October 2007 brought together a collection of big data science users with extreme data base requirements.

The users complained about the inadequacy of current commercial DBMS offerings. RDBMSs would never meet the scientific community requirements because they have the wrong data model, the wrong operators and are missing the required capabilities [33]. The RDBMS vendors appear not to be focused on the science market, because the business enterprise market is perceived to be larger; hence, there was skepticism that these shortcomings
would ever be addressed.

A representative from the database research community (Michael Stonebraker) said that, if the workshop participants could define functional requirements for a broadly applicable science DBMS, then they would try to build it. What followed was a sequence of small workshops to define what ultimately become SciDB. Today, the development of a commercial version of SciDB, with improved performance, complete documentation and commercial assistance, has been undertaken by Paradigm4\(^1\).

### 4.2 Array data model

SciDB has a native array data model. Instead of classical tables, the logical object accessed by the user is a N-dimensional array. This choice was taken because arrays are the natural data object for much of the science, and because most of the complex analytics that the science community uses are based on core linear algebra operations.

The database can contain an arbitrary number of arrays, each one with a unique name and an independent schema. The array schema is defined by a list of dimensions and a list of attributes.

\[
\text{array\_name} = < \text{attributes} > [ \text{dimensions} ]
\]

SciDB allows any number of dimensions for an array[33]. Originally, non-integer dimensions were allowed, because, in many contexts, the information is more naturally defined by float or string domain specific values (ex. latitude, longitude, labels, codes). These kind of dimensions were managed through specific mapping tables automatically created and maintained by the system. Today non-integer dimensions are no longer supported, every coordinate of the array has to be an unsigned 64 bit integer. If codes and labels are required as dimensions, the user has to explicitly manage the indexing operation using the \texttt{uniq} and \texttt{index\_lookup} operators. Each dimension is defined using the following syntax:

\[
\text{dim\_name} = \text{dim\_lo} : \text{dim\_hi} , \text{chunk\_length} , \text{overlap}
\]

The \texttt{dim\_lo} value is an expression for the dimension start value. \texttt{dim\_hi}, the dimension end value, could be either an expression or an asterisk (*). Asterisks indicate the dimension has no limit (referred to as an unbounded dimension). Together, the starting and ending values define the range of

\(^1\)The current version can be downloaded from http://forum.paradigm4.com/c/downloads

30
possible values that the dimension coordinate can take. This range includes both the starting and ending values themselves. For example, \([1,1000]\) defines a dimension size of 1000 [17].

The last two values, chunk_length and overlap, are the number of dimension values between consecutive chunk boundaries and the number of overlapping dimension values for adjacent chunks. In section 4.3 will be presented the storage management of the arrays, and will be better explained the meaning of these parameters.

Each combination of dimension values defines a cell of the array, which can hold an arbitrary number of attributes of any user-defined data type. Attributes are defined using the following syntax:

\[\text{attribute\_name : attribute\_type [ nullable ] [ default ]}\]

The attribute type can be either a native SciDB type (float, double, int, uint, string, char, datetime) or a user-defined data type. SciDB supports Postgres-style user-defined types and functions. The nullable parameter indicates whether null values are allowed for the attribute. The default value specifies the value to automatically substitute when the user not explicitly supplies a value.

The choice of whether data should be an attribute or a dimension is a logical database design problem that should be based on the expected workload to be processed. Usually, if a data is frequently used to address the information vector, it is a good candidate to be a dimension. Moreover, SciDB supports schema migration, so attributes can be promoted, if possible, to dimensions and dimensions deprecated to attributes. Moreover, attributes and dimensions can be added and removed.

To clarify the SciDB data model, we propose an example for a sensor network[34]. Figure 4.1 shows data coming from the sensors: each element provides information about wind speed, temperature and general weather.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Time 100</th>
<th>Time 500</th>
<th>Time 900</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.5 75.1 clear</td>
<td>11.5 69.5 cloud</td>
<td>9.5 65.2 rain</td>
</tr>
<tr>
<td>2</td>
<td>1.0 55.2 cloud</td>
<td>0.5 55.2 clear</td>
<td>2.0 55.2 clear</td>
</tr>
<tr>
<td>3</td>
<td>10.0 35.1 snow</td>
<td>12.5 34.2 snow</td>
<td>12.5 33.8 snow</td>
</tr>
<tr>
<td>4</td>
<td>0.5 85.1 clear</td>
<td>1.0 85.5 clear</td>
<td>0.5 85.7 clear</td>
</tr>
<tr>
<td>5</td>
<td>15.2 66.2 rain</td>
<td>7.9 66.5 clear</td>
<td>12.7 66.9 clear</td>
</tr>
</tbody>
</table>

*Figure 4.1: Example of data coming from a sensor network*
conditions with a certain frequency. According to this kind of data, a possible schema for a SciDB array is the following:

```plaintext
SENSOR_DATA
  <wind:double, temp:double, cond:string>
  [sensor=1:15, time=0:*]
```

Sensor ids and timesteps are natural dimensions of data, because they are a spatial and a temporal coordinate for the tuples. The sensor dimension can be bounded if we previously know the number of involved sensors. Using the proposed schema, the data in figure will be imported into SciDB as follows:

```plaintext
{sensor,time} wind,temp,cond
{1,100} 12.5,75.1,'clear'
{1,500} 11.5,69.5,'cloud'
{1,900} 9.5,65.2,'rain'
{2,100} 1.55.2,'cloud'
{2,500} 0.5,65.2,'cloud'
{2,900} 2.55.2,'clear'
{3,100} 10,35.1,'snow'
{3,500} 12.5,34.2,'snow'
{3,900} 12.5,33.8,'snow'
{4,100} 0.5,85.1,'clear'
{4,500} 1,85.5,'clear'
{4,900} 0.5,85.7,'clear'
{5,100} 15.2,66.2,'rain'
{5,500} 7.9,66.5,'clear'
{5,900} 12.7,66.9,'clear'
```

### 4.3 Multidimensional Array Clustering (MAC)

The following section is completely extracted from [19], a document provided by Paradigm4 that explains in details the MAC system.

SciDB’s Multidimensional Array Clustering storage subsystem is built to efficiently store multi-attribute multidimensional arrays, exceeding tens of terabytes in size, in a distributed DBMS, while facilitating array-style slicing and lookup operations as fast as possible.

The most common operations that SciDB should be efficiently support are: slices, returning all relations for some values of a certain dimension, multiple horizontal slices, with projections that only returns a subset of the attributes, and sub-region selections, rectilinear regions defined by dimensions. Figure 4.2 shows these three operations in order to understand what they mean on an array data structure. In order to meet the above objectives, MAC was designed using the following features.

- Columnar storage with respect to attribute. Even if an array has hundreds of attributes, only the attributes requested by the query are read off on the disk.
• Algebraic indexing. SciDB can very quickly figure out the on-disk physical location of a cell or a requested block of cells. The indexing method utilizes a combination of hashing as well as lookup structures that are automatically maintained as the data sizes grow.

• Clustering. SciDB clusters data in it’s chunks so that co-local regions of the logical array are co-located in the physical data. This ensures that, for a slice or between query, the number of disk locations that need to be visited to retrieve the required data is minimized.

In the following paragraphs will be presented the chunking mechanism of SciDB. A deep exploitation of this sophisticated feature could improve the performance significantly.

**Chunking**  SciDB runs as a network of processes, or instances, each responsible for a subset of the overall data, usually on a cluster of multiple physical computers, or nodes. Each instance keeps data in its own file system directory, which is usually located on an independent storage device, but may also be part of a large, shared storage subsystem.

MAC works by defining a grid of fixed-size rectilinear chunks that partition the multidimensional space of the array. Each chunk is then assigned to a particular instance using a hash function over the chunk’s coordinates in the array space. Chunks for different attributes are stored separately.

Figure 4.3 shows an example. The attribute a1 and a2 are stored separately. The chunk size is set to $3 \times 3$ and there is a total of 9 chunks’ worth of data for each attribute. The diagram shows the chunks at position $\{0,0\}$ are assigned to instance 1, whereas the chunks at $\{3,3\}$ are assigned to instance 2 by virtue of hashing. Prior to being written to disk, chunks are run-length encoded, compressing out frequently repeated values. There is also a hidden empty bitmap (EBM) attribute stored as a run-length encoded bitmask that encodes the positions of non-empty cells inside each chunks.
The chunking aspect of the architecture ensures that only a few chunks are required to satisfy a particular dimensional query. For example, to project $a_1$ from a slice along $\text{dimension1}=4$, we would only need to scan 3 of the chunks: $a_1\{0,3\}$, $a_1\{3,3\}$, and $a_1\{6,3\}$. Attribute $a_2$ is not touched since it is not requested. Only 3 chunks out of 18 total would have to be read off the disk. Moreover, the hashing ensures that the three chunks are likely on separate SciDB instances, so that the disk reads can happen in parallel.

In DBMS-theoretic terms, SciDB is said to automatically index and cluster data on dimensions. Indexing means that, given particular dimension coordinates, the system can retrieve data at those coordinates without having search through most of the data; it is just a quick matter of locating the right chunks. Clustering means that data which are close to each other in the array coordinate system are likely stored in the same region on disk.

**Chunk size selection** The user has full control over the array chunk sizes, a powerful tuning knob that can often make a difference of over 100x in query performance. Picking an optimal chunk size can be a complex problem, particularly if the data are sparse and skewed. As general rule, some skew is tolerated and the average chunk should be between 5MB and 50MB in size. This ensures a good ratio between the number of chunk map entries and data payloads. The chunk is the unit of disk read, so if some
chunks are large as multiple GB, reading a single chunk may cause a spike in memory usage and lead to instability. This also diminishes the selectivity advantage when a query only needs to lookup a few values. On the other hand, if the chunks are too small, then the number of chunk map entries grows, leading to chronic increased memory usage and ineffective storage utilization.

Assuming few repeated values, a chunk with a million 8 byte numbers, or 7-characters strings would take up roughly 8MB. Recall that rather than storing dimension coordinates, SciDB encodes the positions of cells using the special EBM attribute. For dense arrays, the EBM is highly compressed and occupies just a few bytes per chunk. For sparse arrays, the EBM takes up to 24 bytes per value, regardless of the number of dimensions. So a million non-empty cells per chunk is often a recommended ratio.

Overlap Optionally, the user can instruct SciDB to store a set of neighboring cells around the boundary of each chunk. Those extra cells are stored redundantly on multiple instances, increasing the storage footprint but, in turn, making window aggregates embarrassingly parallel, thus avoiding extra chunk reads for windows that straddle chunks so long as the window size fits inside the overlap region.

4.4 Array Functional Language (AFL)

SciDB provides two different query languages in order to access and manipulate arrays inside the system: AQL and AFL.

The Array Query Language (AQL) has a SQL-like syntax and was designed in order to be intuitive for programmers already experienced with commercial RDBMS solutions. AQL is compiled into an AFL query, and for this reason it’s possible to use AFL operators inside the from clause of an AQL query.

The Array Functional Language (AFL) comes from the need to have a list of primitives that could be combined and nested to produce the result. This language was strongly inspired by APL. Follows a simple pattern to understand the syntax of AFL.

\[
\text{operator}_1(\text{operator}_2(\text{array}, \text{args}_2), \text{args}_1);
\]

Basically each operator is a function returning a new array structure. These functions can receive one or two arrays as arguments, plus a number of parameters that depends on the specific operation. Excluding some particular
operators (like store), the result of one operator can be used as input for another one, nesting the two queries.

According to [14], SciDB is able to optimize the queries avoiding, when possible, the materialization of the intermediate results. The optimization strategy is currently under development, and it’s important to remember that some classical DBMS improvements are not still completely supported. An example is the exploitation of the same intermediate result required by two different subqueries [15].

4.4.1 iQuery client

To communicate with SciDB and issue both AFL and AQL queries, the system provides the iQuery client. SciDB can be deployed on a cluster of machines in order to improve the computational power of the system. Regardless of the cluster size and the number of involved nodes, each SciDB installation has a single coordinator machine. The coordinator acts as access point for the system, that is presented to the final user as a single black box. The management of the nodes interactions and the data distribution is completely up to SciDB.

The following command represents an exemple of iQuery usage. In the example the query is directly passed as argument of the iquery executable.

```
iquery -aq "store(filter(NARROW_PEAKS, chr=12), CH12);"
```

For big queries, or list of queries, it’s possible to save them inside a file and then pass the file name as argument.

```
iquery -af pipeline_021.afl
```

The a flag means that the following query is written using AFL, because AQL is set as default.

4.4.2 Operators

SciDB provides a set of native operators for array management and manipulation, in addition to the classic DBMS operations like selection, projection, join and so on. This section presents some of the most important operators, that are frequently used for our work and that will be recalled in the next chapters.\(^2\)

\(^2\)For a complete documentation of SciDB operators see https://paradigm4.atlassian.net/wiki/display/ESD/SciDB+Operators
In order to meet the specific requirements coming from science, SciDB is extensible through Postgres-like plugins where can be implemented user defined operators.

**Filter**

The *filter* operator returns an array containing just the elements that satisfy the boolean expression.

\[
\text{filter}(\text{array, expression});
\]

The expression condition can be defined as conjunction and/or disjunction of atomic clauses on attribute or dimension values. The condition is evaluated cell-wise without optimizations on the filtering conditions. Considering the example proposed in section 4.2, let’s see an application of the filter operator.

\[
\text{filter(SENSOR\_DATA, wind>10)};
\]

\[
\text{\{sensor, time\} wind, temp, cond}
\]

\[
\{1,100\} 12.5,75.1,'clear'
\]

\[
\{1,500\} 11.5,69.5,'cloud'
\]

\[
\{3,500\} 12.5,34.2,'snow'
\]

\[
\{3,900\} 12.5,32.8,'snow'
\]

\[
\{5,100\} 15.2,66.2,'rain'
\]

\[
\{5,900\} 12.7,66.9,'clear'
\]

**Between**

Frequently, a user requires a sub-part of the entire structure in order to analyse a specific aspect of the data. The *between* operator is a filter that masks a rectilinear region of the input array.

\[
\text{between(array, low\_coord1 [, low\_coord2 ...]},
\]

\[
\text{high\_coord1 [, high\_coord2 ...]);}
\]

The result is an array with the same schema of the input array and containing all and only the cells within the specified dimensions. The operator requires first the lower bounds lists (one for each array dimension) and then the upper bounds one. It’s possible to use the keyword `null` in order to select all the points on a certain dimension.

Each *between* operation can be realized using the *filter* operator, too, specifying the dimension limits as conjunctions of conditions. Using *between* we can considerably improve the query performance, because it exploits the MAC system, selecting the chunks (or portion of chunks) that compose the results without loading the data. Just to understand the difference, according to [19], a between selection on a 7GB array can require 0.152s
against the 25.742s required by the corresponding filter condition. In the example below, we want to select just the data from every sensor with timestamp between 200 and 600.

between(SENSOR_DATA, null, 200, null, 600);
{sensor, time} wind, temp, cond
{1,500} 11.5, 69.5, 'cloud'
{2,500} 0.5, 55.2, 'clear'
{3,500} 12.5, 34.2, 'snow'
{4,500} 1.85, 5.2, 'clear'
{5,500} 7.9, 66.5, 'clear'

Cross Join

The cross_join operation is the equivalent of the classic join in SciDB. Instead of performing the complete cross product between the two arrays, the user can specify a list of dimension equality condition in order to pair just the cell having the same values on those dimensions.

cross_join(left_array [ as left],
right_array [ as right],
[left.]left_dim1, [right.]right_dim1, ...);

The equality predicates can be resolved without the loading of the data, just requiring the cross product between subsets of chunks. In order to join two arrays using non equality conditions, it’s required to perform a filter after a complete cross join.

Let’s consider, for example, having stored into SciDB a second array, called SENSOR_INFO, containing all the information about the sensors (like the installation year).

{sensor} name, installation
(1) 'sensor_1', 2013
(2) 'sensor_2', 2012
(3) 'sensor_3', 2015
(4) 'sensor_4', 2013
(5) 'sensor_5', 2015

If we want, for any reasons, all the data coming from sensors installed after 2014, we have to join the two structures to filter the information. In this case the filter can be pushed inside the join operation filtering the SENSOR_INFO table and then use this result in the cross join.

cross_join(SENSOR_DATA as D,
filter(SENSOR_INFO, installation>2014) as I,
D.sensor, I.sensor);

{sensor, time} wind, temp, cond, name, installation
{3,100} 10, 35.1, 'snow', 'sensor_3', 2015
{3,500} 12.5, 34.2, 'snow', 'sensor_3', 2015
{3,900} 12.5, 33.8, 'snow', 'sensor_3', 2015
{5,100} 15.2, 66.2, 'rain', 'sensor_5', 2015
{5,500} 7.9, 66.5, 'clear', 'sensor_5', 2015
{5,900} 12.7, 66.9, 'clear', 'sensor_5', 2015
Redimension

As mentioned at section 4.2, an attribute can be promoted, if possible, to dimension and a dimension deprecated to attribute. The redimension operator allows changes in the array schema. The resulting schema has to be defined using exactly the same attribute or dimension name. Of course just integer attribute can be promoted to dimension.

\[
\text{redimension}(\text{source\_array}, \\
\quad \text{schema\_definition} \\
\quad [, \text{isStrict}] \\
\quad [, \text{aggregate(attribute)} \ [\text{as label}]] \ldots );
\]

Redimensioning the array means that potentially two or more cells of the source can fall into the same resulting cell. If isStrict flag is selected, collisions are interpreted as errors, and an exception is thrown. Another option is the insertion into the new schema of one synthetic dimension used to ”list” the colliding cells, if no synthetic dimension is declared, the system will keep one of the colliding cells.

Finally, the last option is the aggregation. If the user specifies a list of aggregation function, the system will generate new attributes in the resulting array computing the functions on the colliding values. In that case, the new aggregated attributes have to be already specified in the target schema definition, too.

Suppose you want to know the minimum and maximum temperature recorded by each sensor. It’s possible to redimension the SENSOR\_DATA array keeping just the sensor id dimension and aggregating the max and min value for temperature of the colliding cells.

\[
\text{redimension}(\text{SENSOR\_DATA}, \\
\quad <\text{max\_temp:double, min\_temp:double}>[\text{sensor}=0:15,16,0], \\
\quad \text{false}, \text{max}(\text{temp}) \text{ as max\_temp, min}(\text{temp}) \text{ as min\_temp});
\]

<table>
<thead>
<tr>
<th>{sensor}</th>
<th>max_temp,min_temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75.1,65.2</td>
</tr>
<tr>
<td>2</td>
<td>55.2,55.2</td>
</tr>
<tr>
<td>3</td>
<td>35.1,33.8</td>
</tr>
<tr>
<td>4</td>
<td>85.7,85.1</td>
</tr>
<tr>
<td>5</td>
<td>66.9,66.2</td>
</tr>
</tbody>
</table>

4.5 Genomic addition

SciDB supports an add-on\(^3\) developed by Paradigm4; the add-on is a toolkit, written in R, designed to perform specific operations on specific datasets

\(^3\)https://github.com/Paradigm4/variant_warehouse
and not, as in GMQL, a generic language designed for application to arbitrary sources. For this reason, inside the add-on, there are several different schemes used to map genomic data in SciDB, depending on the dataset kind. In general, regions are stored using indexes like chromosome id, sample id and tumor id as dimensions. Sometimes start and stop ends are used, too.

In this document we are not interested in an exhaustive presentation of the various adopted schema, but we have analyzed them in order to design our schema considering the different requirements.

4.5.1 Binning strategy

One of the most critical operation using a region based data model is the range intersection. This procedure is frequently used in genomic analysis and its computation represent a complex problem working on big data.

The region intersection, for example used by the GMQL Map operator, will be the core of chapter 7. For this reason this section describes the optimization adopted by Paradigm4 in order of parallelize and improve this procedure.

Obviously, the easiest way to perform the range intersection is compare each possible couple of regions from the reference and experiment datasets (the used names refer to the map definition at section 2.2.4), selecting the ones that satisfy the intersection predicate. In SciDB this is possible executing a cross join on the two corresponding arrays and then applying a filter on the intermediate result. This method requires the cross product of the arrays with a computational complexity that depends on the product between the reference size and the experiment size, not feasible on really big data.

Figure 4.4 shows how Paradigm4 improves the region intersection procedure applying a binning strategy (they call the bins ”buckets”). Without presenting in deep the used code, we will summarize the strategy.

- **Step 1** - Compute the maximum length of the regions. This value will define the lower bound for the bins size. In this way we can be sure that each region falls at maximum into two different bins.

- **Step 2** - Each region is then duplicated joining the datasets with a synthetic 2 cells array. Then bin ids are assigned. To the first copy is selected the bin where falls the left end of the region, and for the second copy the bin where falls the right end. If the two values are the same, the second copy is dropped. At the end each region, duplicated if required, is marked with a single bin id.
Figure 4.4: Binning strategy adopted by Paradigm4 in the genomic add-on.

- **Step 3** - Execute a cross join between the two prepared datasets on same chromosome and same bin id. This will reduce the original cross product checking just the regions that fall in the same bin.

- **Step 4** - The intersection condition is then applied on the cross join result, selecting just the intersecting regions for each bin.

- **Step 5** - It’s required a clean up step to avoid duplicates in the result. Looking at figure 4.4, the couple (R2, E2) is evaluated both for bin 1 and bin 2. To remove duplicates the procedure drops the pairs composed by two regions not starting in the current bin.

- **Step 6** - Finally, the result is produced simply merging the partial ones.
Using the presented procedure, really distant regions are not evaluated, reducing the computation complexity of the operator. Nevertheless this method has two drawbacks: some data have to be replicated, and it is sensitive to the presence of really big regions that, however few, increase the bin size reducing the clustering power.

In the rest of the document we refer to this binning strategy as mono-dimensional binning or 1D-binning.
Chapter 5

Scala APIs for SciDB

SciDB was designed and built to meet the requirements coming from the science community. In order to provide a comfortable environment for science people, the developers implemented APIs for both R\(^1\) and Python\(^2\), the most used programming languages in science, to simplify the integration of SciDB with the already developed codes.

Basically, the two APIs provide a representation of the SciDB arrays inside the language environment, this objects are just pointers to the structures inside the DBMS. The AFL operations are performed using a set of functions that use these pointers to generate and issue the corresponding queries. In addition to the native operators, the APIs contain some methods to simplify the input/output procedures between the developing environment and SciDB, usually just through mono or bi-dimensional arrays.

The usage of the APIs doesn’t increase the expressive power of the system: every operation can be done communicating with SciDB through AFL queries directly issued to the iQuery terminal.

GenData 2020 was completely implemented using Scala, hence, in order to facilitate the implementation of GMQL on SciDB, we have decided to build a new general purpose library that provides an abstraction of the AFL syntax and a set of tools designed to build and run AFL queries in an agile way from the Scala environment.

This chapter will briefly present the built library. At first, section 5.1 introduces the adopted array representation, then section 5.2 describes the calling of the operators, section 5.3 the storing procedures and section 5.4 the management of an AFL script and the execution of the resulting query.

\(^1\)http://paradigm4.github.io/SciDBR/
5.1 Array representation

As presented in section 4.2, in SciDB an array is described by a list of dimensions and a list of attributes; if already stored, by a name, too. Following the native definition of the arrays, we have implemented the Scala representation as follows:

```scala
val WINDY = new SciArray(
    List[Dimension]("sensor", 1, Some(15), 100, 0),
    List[Attribute]("wind", DOUBLE),
    "filter(SENSOR_DATA, wind>10)")
```

```scala
val TEMPS = new SciArray(
    List[Dimension]("sensor", 1, Some(15), 100, 0),
    List[Attribute]("temp", DOUBLE),
    "project(SENSOR_DATA, temp)")
```

Figure 5.1: Code example of array representation with Scala API

The `Dimension` class is a simple container for dimension name, limits (None option stays for asterisk), chunk length and overlap size. In the same way `Attribute` contains the name, type and the nullable flag (the `DataType` class is just the enumeration of the allowed types).

```scala
Dimension(name:String,
    dim_lo:Int, dim_hi:Option[Int],
    chunk_length:Int, overlap:Int)
```

```scala
Attribute(name:String,
    datatype:DataType,
    nullable:Boolean = true)
```

In addition, each array object in Scala stores the query that produces the corresponding array if executed in SciDB. This is the core of our library: each array object maintains it’s own translation of the high level representation inside Scala environment into the AFL query that will build it inside SciDB.
Figure 5.1 shows two examples of SciArray objects. The first one represents the array that should be produced filtering the sensor data on the wind attribute. The second represents the array obtained projecting the data on the temperature attribute.

5.2 Operator methods

According to the explained representation, AFL operations have to be implemented as Scala functions that manipulate SciArray objects. The SciArray instances are immutable objects, and all the operators are defined as methods of the SciArray class that returns new instances as result. Here we can see three examples of prototypes.

```scala
def filter(expression:Expr) : SciArray =
{
  val context = (dimensions, attributes)
  // expression evaluation ----------------------------
  val expressionString = expression.eval(context) match {
    case (string, dataType, BOOL) => string
    case (string, dataType) => throw new IllegalOperationException()
  }
  // query construction ----------------------------
  var resultQuery = "filter(" + query.tab() + ") + expressionString + ")"
  // output -------------------------------
  new SciArray(dimensions, attributes, resultQuery)
}
```

Each method takes as input the conditions or the values required by the specific operation and, if required, the second operand array.

The first step of a method is check if the required operation is valid, verifying the input values and evaluating them respect to the current schema of the array. For example in the filter method is verified if the condition expression, defined as a syntax tree, has value for the present array.

Then the object dimensions and attributes lists are copied to the result and manipulated according to the operator definition. For example, the filter methods simply copies the input schema without alterations, while the project keeps the same dimension list and selects the attributes that are required in the projection list passed as parameter.
Finally, the query string is manipulated. The resulting string is the syntax of the AFL operator where the nested subquery for the source array is the string of the input object query string. Figure 5.2 shows the implementation of the filter method, while figure 5.3 shows the method calls required to obtain the same queries proposed as examples in section 4.4.2.

5.3 Stored arrays

As already mentioned, stored array have a name, too. Figure 5.4 shows the UML diagram of the complete arrays representation adopted by our library. We have decided to distinguish between live arrays and stored arrays. The first ones is actually the object presented until now, having all the AFL operators as methods. The second ones have a name, the anchor, and only one method called reference.

Once we have manipulated the live arrays, applying the sequence of required operations, to persist the result it’s possible to call the store method, that requires as parameter the name that we want assign to the persisted array. The store operation return a SciStoredArray instance.

```java
val SENSOR_DATA = new SciArray{
    List.Dimension("sensor",1,Some(15),100,0),
    List.Dimension("time",2,None,100,0)),
    List.Attribute("wind",DOUBLE),
    Attribute("temp",DOUBLE),
    Attribute("cond",STRING),
    "SENSOR_DATA"
}

val SENSOR_INFO = new SciArray{
    List.Dimension("sensor",1,Some(15),100,0)),
    List.Attribute("name",STRING),
    Attribute("installation",INT64)),
    "SENSOR_INFO"
}

val WINDY = SENSOR_DATA.filter( OP(A("wind"), "">", V(10)) )

val TIMED = SENSOR_DATA.between((null,null),(200,600))

val JOINED = SENSOR_DATA.cross_join
    (SENSOR_INFO, filter( OP(A("installation"), "">", V(2014)) ), "D", "I")
    ((D("sensor"), D("sensor")))

val JOINED = SciArray
    (schema:<wind:double, temp:double, cond:string>:{sensor=0:15,100,0, time=0:*100,0}
    query: "filter(SENSOR_DATA, wind>10)"
    
    val JOINED = SENSOR_DATA.cross_join
    {schema:<wind:double, temp:double, cond:string, name:string, installation:int64>
    filter(SENSOR_INFO, installation>V(2014)) as I,
    D("sensor"), I("sensor")
}
```

*Figure 5.3: Code example of operation method usage*
According to the official definition, the store AFL operation can not be used as input for another operator. This is why we have defined two different objects, and why it’s possible to apply AFL methods just on the SciArray instances.

Nevertheless, stored arrays can be used by other operators just calling the array name. For that reason, the reference method returns a SciArray instance with the same schema of the persisted array, but with just the name as query. In that way, the reference can be use as live array itself, or passed as second operand in another method. Figure 5.5, at line 11, shows an example of the reference usage.

5.4 Script management

In section 4.4.1 was presented the iQuery terminal. One of the usage method is calling the executable passing the file name of an AFL script. An AFL script is a list of queries and commands, generally called statements.

The SciScript class in our library represents a script manager that provides the method addStatement to add a new query or command to the current script. Each SciArray or SciStoredArray instance is a valid statement, and when issued, the corresponding query is added to the script content (queries corresponding to not stored array are read-only operation that return the fetched cells as string in the terminal without permanent modification of the system).

Commands are statements for the configuration of iQuery options\(^3\). For example the user can specify \texttt{set [no] fetch} to enable or not the printing of the results on the console, or \texttt{set [no] timer} to require the execution times of each query.

\(^3\)https://paradigm4.atlassian.net/wiki/display/ESD/The+iquery+Client
In addition, another valid statement is the special command `create`. This operator allows the definition of new empty arrays. The Scala object to define a create statement is the following:

```java
SciCreate(temp : Boolean, name : String, dimensions : List[Dimension], attributes : List[Attribute])
```

The `temp` flag allows the definition of temporary arrays that are materialized structures, without persistence guarantees on system errors. This structure is used, because faster, when users need to define intermediate results that will not be persisted at the end of the procedure. This command is translated with the following AFL statement:

```
create [temp] array name < attributes >[ dimensions ];
```

After the issuing of all the required statements, the `SciScript` class provides the `run` method to execute the script on the desired SciDB instance. If
no values are passed as parameters, the method assume that the iQuery terminal is available on the local machine. Otherwise, the user can specify the IP address and the login credential to open a SSH connection with a remove machine and run the script on the desired system. Figure 5.5 shows how an AFL script can be defined and executed.

```python
def run(server_ip:String = null,
        server_user:String = null,
        server_pass:String = null)
```

### 5.5 Queues

In order to increase the flexibility of the script management, we have introduced the queues system. Using queues, the user can issue a list of statements that will be saved inside the SciScript object, but that will be added to the statements list just when required. The following primitives are the methods provided to use the queue:

```python
def addQueueStatement(stmt:SciStatement)
def flushQueue()
```

Some times one single queue is not enough, and the user needs a stack of queues. This is the case of recursive programming, where for each call it’s required to open a new context and the current one has to be saved and restored at the end of the call. For this reason the queue system provided by SciScript provides the primitives to open, and close, new queues; in this case the previous commands (addQueueStatement and flushQueue) work on the last opened queue.

```python
def openQueue()
def closeQueue()
```
Chapter 6

GMQL Implementation on SciDB

In the introduction we have anticipated that the core of the thesis is design, and realize, a new implementation of GMQL v2 using SciDB. The two main problems that we have to address to archive that goal is the design of a module compatible with the already implemented system, in order to maintain the great flexibility of GenData 2020, and the complete re-thinking of the algorithms that implement the operators, due to the complete different nature of the used technology respect to the ones used for the already existing implementations.

Section 6.1 will present the integration of the new module inside the existing architecture and the details about internal architecture and behaviour. Section 6.2 is dedicated to the explication of the data model adopted to map genomic data into SciDB arrays. Finally, section 6.3 describes the translation procedure that generate the AFL script for the issued query starting from the passed DAG and section 6.4 the detailed implementation of some operators.

6.1 Architecture

In chapter 3 we have explained the architecture of GenData 2020. The system was designed to be compatible with various implementations of GMQL, realized on different technologies. To do that it was necessary to define an abstract interface for the implementations. Each module that extends this interface can be used by the rest of the system without worrying about the specific used technology. Here we report the methods of the interface:
The `getDataset` method is used by the compiler to check if a certain dataset is already present in the system and if yes, which is its schema. When the compiler has generated the DAGs representing the materialized results, it passes the graphs to the implementation through the `addDAG` method and, when it is ready, calls the computation procedure using the `go` method.

Extending the presented interface, we have realized a new implementation of GMQL on SciDB. Figure 6.1 shows the general project architecture and how our module will be integrated with the existing modules.

### 6.1.1 Internal architecture

Any GMQL implementation receives the queries that have to be processed as DAG objects. Then the task is to translate the commands represented by nodes into specific procedures that realize the query for the underlying technology.

In our case, a query has to be transformed into an AFL script that produces the arrays containing the materialized results (in section 6.2 will be presented in details the dataset representation using SciDB arrays, for the moment, just to better understand the general architecture of the module, it’s possible to think that each dataset is represented with an array).

Therefore, our implementation is basically a translator, that has to interpret the issued graph, understand how translate each node to a sequence of AFL operations, combine them, and run the generated script on SciDB.

Just to recall, in section 4.4.1, we explained that SciDB has one single
coordinator able to provide a monolithic abstraction of the system. For this reason, after the translation phase, it’s sufficient to run the produced script on the desired iQuery terminal. Figure 6.2 shows the execution scheme of the implementation module.

6.1.2 Data flow

This section want show how data are loaded, manipulated and exported by our implementation. Figure 6.3 contains a complete data flow schema.

In order to use a dataset inside a query, it’s first of all required to import the data inside SciDB. All the importation (and exportation) functions are provided by the class \texttt{GmqlSciRepositoryManager}; these set of primitives will be called by the application user interface, responsible for the repository management. It’s possible require a dataset importation using the following method:

\begin{verbatim}
importation(name:String, 
path:String, 
columns:List[([String, PARSING_TYPE, Int]), 
format:String = "tsv")
\end{verbatim}

The function requires the name of the imported array, the local path of the
source files, the columns of the file that will be imported and the file format (usually TSV, but can be used the CSV format, too).

The importation step is required only once. After the importation of the dataset, all the data will be persisted inside SciDB and every future query that requires that dataset as input will directly accesses the imported array.

Each GMQL query is then translated into an AFL script. The query can use as input every array stored inside SciDB (possibly just the ones that the user can use according to its permissions). Execute a query means perform some operations on the input, manipulate the fetched data and store the result as a new materialized array. This new structure represents the dataset materialized by the query.

Once generated the query result, the user can export the produced dataset in a typical file format through the exportation primitive.

```scala
fetch(location:String) : Option[IRDataSet]

exportation(dataset:IRDataSet,
             path:String,
             format:String = "tsv")
```

The `fetch` function returns the dataset information for the required name, while the `exportation` method exports the dataset to the specified local directory using the model. As for the importation, the exportation calls will be done by the repository manager user interface.

### 6.2 Data model

In section 2.2 was already presented the data model adopted by GQML in order to provide a general and flexible representation of genomic data. To
realize a new implementation on SciDB, it’s required to import the datasets, provided as pairs of files, inside the database, using an array base data model compatible with GMQL operations and characteristics.

Each dataset, having a specific region schema, will be mapped with a pair of SciDB arrays, respectively containing metadata and region data. Using this representation, it’s possible translate the GMQL operations, working on datasets, into SciDB operations, working on arrays.

6.2.1 Meta data

For each sample, metadata are defined as a set of pairs attribute:value, where each attribute can be repeated more than once with different values. In order to design the best model to map metadata on a SciDB array, we have to consider some GMQL characteristics:

- most of the operations performed on metadata are selections of sample ids, and or conjunctions are interpreted as intersection and union in algebra of sets;
- the selection operations usually are based on equality conditions, on attribute name and/or value, range conditions are allowed but not frequently used;
- selection conditions are evaluated using the existential interpretations, that means: a sample is selected if exists at least one pair attribute - value that satisfies the condition;
- metajoins and metagroups used by domain specific operators, require respectively the coupling and the grouping of samples based on attribute values.

The previous requirements suggest that we should choose a schema where sample id lists can be reached just slicing the structure on a certain attribute name and/or value and where it should be possible join two metadata arrays by attribute name and value in order to obtain the metajoin couples.

The solution is store the metadata into a cube where the three dimensions are: attribute name, value and sample id. As reported in section 4.2, SciDB doesn’t permit string as dimension type. To avoid that problem it’s possible store the data using some hash values of name and value as dimensions. The hashing of strings into 64-bit integers (standard dimension type in SciDB), introduces possible collision errors, for example two colliding attribute names means, in addition to a possible information loss, that
a selection based on a slice of the array returns sample having the required attribute or the colliding one. Using a double hashing, two values coming from two different hashing functions, both for attribute name and value, we can significantly reduce the collision probability, or even completely avoid it using two orthogonal hash functions, according to [12] definition.

According to the previous considerations, for each dataset DS, metadata are stored into a single 5-dimensional array.

\[
\text{DS MD} = \text{name: String, value: String} \\
[\text{nid}_1, \text{nid}_2, \text{vid}_1, \text{vid}_2, \text{sid}]
\]

where \((\text{nid}_1, \text{nid}_2)\) and \((\text{vid}_1, \text{vid}_2)\) are respectively the double hash values for attribute name and value and \(\text{sid}\) is the sample id. The schema of metadata arrays are identical for all the datasets imported into SciDB.

6.2.2 Region data

Similarly to the metadata, for each dataset DS, regions are stored into a single array. The regions are organized according to the relative sample id and genomic coordinate. Conceptually, each not empty cell inside the array represents a region.

In order to use region coordinates as dimensions for the SciDB data model, it’s required to cast chromosome and strand values, natively represented by strings, to integers.

For chromosomes, it’s possible define a global codification map table that provides chromosome ids shared among all the datasets. This indexing operation is natively supported by SciDB through `uniq` and `index_lookup` operators\(^1\). For strand, due to the limited accepted values, it’s possible apply a static conversion\(^2\).

Using these transformations, regions data are mapped to a 6-dimensional array, where attribute fields are based on the specific dataset feature schema provided by the user.

\[
\text{DS RD} = \text{< feature schema >} \\
[\text{sid, chr, left, right, strand, x}]
\]

---

\(^1\)Paradigm4’s official forum discussion: http://forum.paradigm4.com/t/prototype-workaround-for-non-integer-dimensions-in-13-6/388

\(^2\)Static conversion applied: 'x'=2, '+'=1, '-'=0
The \( x \) dimension is an enumeration value required because for GDM (section 2.2), each sample could have more than one region with the same coordinates and at least one different feature. This value is a unique identifier inside one single sample.

The feature schema can be composed by any list of attributes according to GDM data model; string, int, long, double, boolean are natively supported by SciDB.

### 6.2.3 Chunk sizes

In the previous section, we presented the logical schema of the arrays used to map genomic data. This paragraph explains in details the dimensions properties for the region array, as presented in section 4.2.

```plaintext
name = dim, chunk, overlap
sid = 0:* , 1, 0
chr = 0:* , 1, 0
left = 0:* , 1000000, 0
right = 0:* , 1000000, 0
strand = 0:2 , 3, 0
x = 0:* , 1000000, 1
```

Sample ids are defined during the importation procedure, that enumerates the files passed as input and assigns an incremental value for each sample, this value can not be previously bounded because a dataset could be composed by an arbitrary number of samples. Each chunk will contain information of one single sample, because the sample enumeration doesn’t induce a proximity correlation between sample with close values.

Chromosome ids, as for the sample ones, can be arbitrary high and are assigned using the map table presented in the previous section. Regions on different chromosomes, even if in the same sample, are always independent; domain specific operators treat separately the chromosomes, evaluating regions just with the same \( \text{chr} \) id.

Left and right coordinates are big as the genome, usually about 300 million bases, but we can assume arbitrary high values. The usage of regions end as coordinates allows their storage based on real regions proximity, a fundamental property in order to speed up domain specific operations that use range intersection or range selection. The physical representation of these two coordinates will be better explained in chapter 7. The chunk length of 1 million comes from experimental evaluations on the best chunk size.
Strand is the only limited dimension, because it can assume just three values: positive, negative or missing. Due to the definition of the missing strand behaviour, regions with different strands can be used and aggregated by the same operation. For that reason, the regions can not be stored into different chunks, according to the strand values.

As already mentioned in the previous section, the enumeration value $x$ is unique within a certain sample. Therefore the number of enumeration values for a chunk is the number of non-empty cells inside that chunk. According to [19], the optimal size for a chunk should be between 5MB and 50MB. Considering the possible attribute types, and that string attributes are usually shorts name or labels, a probable single attribute size is about 8-Bytes.Chunks with a million of regions has size about 8-10 MB.

### 6.2.4 Example

In this section, we propose a simple importation example in order to better understand the used data model inside SciDB. Let’s consider a small dataset with just two samples. Figure 6.4 shows the GDM representation of the input files.

After the importation into SciDB, the dataset $DS$ is stored into the arrays $DS_{MD}$ and $DS_{RD}$. Figure 6.5 shows the representation of the two arrays.

---

3Hashes values were truncated for a better visualization of the structures
6.3 Query Processing

Section 6.1.1 presented the module architecture. We explained that the implementation basically performs a translation from the DAG representation of the GMQL query to an AFL query. In this section will be explained in detail the translation procedure, showing how the DAG nodes are mapped into AFL operators.

6.3.1 General procedure

A DAG represents a query as graph, where each node is an atomic operation. Each node uses the others as input and provides the result as new region or meta data. Figure 6.6 shows an example of the graph generated for the following query:

\[
\begin{align*}
\text{REF}_1 &= \text{SELECT}(\text{chr}='\text{chr1}') \text{ REF}; \\
\text{RES} &= \text{MAP}(\text{max}($1)) \text{ REF}_1 \text{ EXP}; \\
\text{MATERNALIZE} \text{ RES};
\end{align*}
\]

The DAG nodes are divided into categories depending on the output type: MetaOperators (the blue ones in the figure), returning metadata, RegionOperators (the red ones), returning region data, MetaJoin and MetaGroup, returning special structures that provides the joining sample pairs or the group mapping table. According to this classification, each node requires, as input, other nodes with specific types; for example a SelectRD node, subclass of RegionOperator, needs one MetaOperator and one RegionOperator as input.
Each AFL operator produces as output an array that can be nested inside other operations. This means that we can translate each single DAG node to a corresponding AFL query, providing the result using standard schemes for the various node types and expecting standard schemes from the other nodes arrays that will be used as nested queries for the current operation. Figure 6.7 shows an example of DAG node translation.

In addition to the region and meta data schemes presented in section 6.2, we need to define standard schemes even for meta join and meta group.

\[
\text{META\_JOIN} = \langle \text{RESULT}\$\text{sid}\$\text{int64} \rangle \\
[\text{ANCHOR}\$\text{sid}, \text{EXPERIMENT}\$\text{sid}] \\
\text{META\_GROUP} = \langle \text{null}\$\text{int64} \rangle[\text{sid}, \text{gid}] \\
\]

The \text{MetaJoin} is described with the sample id pairs as dimensions and the new sample id assigned to the result as attribute. The \text{MetaGroup} simply
defines a group id \((\text{gid})\) for each sample, the \texttt{null} attribute is used just because it’s required at least one attribute.

### 6.3.2 Materialization

A node in DAG can be used by several other nodes; hence, the intermediate result can be read more than once. The translation produces for each node a corresponding SciDB array that in general is directly nested into other operators. When an array has to be read more times, a materialization of the intermediate results could speed up the execution. This is a well known problem for the database community, called \textit{Multi Query Optimization}. Roy \textit{et al.} [28] propose some heuristic methods to understand which nodes should be materialized and which not.

A complete optimization of the materialization problem will be a future goal for our work. However, from experimental results we have seen that some complex nodes always require a storage of the intermediate results. For this reason we have decided to design the translation procedure predisposing the possibility, for each node, to be materialized in a temporary structure that will deleted when the final result is ready.

### 6.3.3 Translation tree

In order to realize the translation, each node in the DAG is extended with a \texttt{GmqlOperator} object, that provides the \texttt{compute} method (it requires as input the script where will be stored all the generated statements). Calling this method we can obtain the array corresponding to the result of the current node.

```python
def compute(script:SciScript) : SciArray
```

\texttt{GmqlOperator} is an abstract class that represents a general translation node; then it is subclasses by other 4 abstract classes, one for each node type: \texttt{GmqlMetaOperator}, \texttt{GmqlRegionOperator}, \texttt{GmqlMetaJoinOperator} and \texttt{GmqlMetaGroupOperator}.

For each single DAG operator (see table 3.1 at section 3.2) we have defined one specific object, subclass of one of the presented types. In section 6.4 will be presented in deep some of the operators, but here we present as example the translation node for the \texttt{SelectRD}.

```python
class GmqlSelectRD(source : GmqlRegionOperator, 
    metafilter : Option[GmqlMetaOperator],
    conditions : Option[RegionCondition])
```
The constructor of the translation node object requires, in addition to the specific conditions and parameters, the input nodes. These nodes are declared using the abstract classes, allowing any operator that produces a result compatible with the required input.

The definition of all these translation nodes generates a twin graph connected point to point with the DAG. This graph is basically a multi-root tree, where each root node is a result that has to be materialized. We call this structure translation tree. To perform the complete translation of the issued DAGs, the implementation calls the compute method on each root passing the same script. After this call, the passed script will contain the complete AFL query that realizes the GMQL query.

Figure 6.8 shows the procedure called to build the translation tree. When addDag receives a new DAG, it runs the prepare function on the two roots (the region and meta data store operations).

### 6.3.4 Translation node

In the previous section we have introduced the GmqlOperator as the node item for the translation tree. The following is the list of attributes and methods of this abstract object:

```scala
val array : Option[SciAbstractArray]
val stored : Boolean = false
```
var usages : Int = 1
var accesses : Int = 0

def apply(script:SciScript) : SciArray
def compute(script:SciScript) : SciArray
def use()

The array attribute stores the produced array, if already computed. stored is a flag that is true if the intermediate result has to be stored. The usages and accesses variables are counters: the first one counts the number of nodes that depend on the current node, the second is the number of times that someone has required the intermediate result. The node is live until the accesses counter is less then the usages one.

The use method is called to increase the number of usages, in figure 6.8 is showed its usage. When the prepare procedure finds a node already initialized, it simply calls the use method to add its ”reference” to that node.

As introduced in section 6.3.3, the compute method returns the array representing the result of the corresponding translation node. Actually, the result is not directly computed by this method. Indeed, compute is implemented by the abstract object and it contains the logic for the generation, materialization and management of the intermediate result. Figure 6.9 presents the flowchart of this method.

When the compute requires the real generation of the array, it calls the internal method apply. It is implemented by the final class and encapsulates the logic of the specific operator. The apply method uses the results of the compute functions called on the input nodes to obtain the arrays that has to used as subqueries.

6.4 Translation of operators

This section will present three of the eleven operators of GQML: SELECT, EXTEND and MAP. We have chosen these three because are examples of the most common operations performed on genomic data, respectively filtering, aggregation and region intersection.

6.4.1 Select

The SELECT operator can either filter samples based on metadata values, or filter regions based on them features. In general a selection statement in the GQML query generates two nodes in the DAG: one for the meta selection and
one for the region selection. Figure 6.10 shows an example of the selection nodes in a DAG. The dependency between the region selection and the meta selection is optional: if the user doesn’t require a meta selection, its is not generated and the region selection is applied on the entire source.

SelectMD

The meta selection filters the source samples keeping just the ones that satisfy a certain list of conditions. The conditions are expressed as a tree of conjunction and disjunction where the leaf nodes are atomic conditions on an attribute. Each atomic predicate is an existential condition: the sample is selected if exists at least one attribute that satisfies the predicate.
Let’s see an example using the dataset introduced at section 6.2.4. Consider a query that requires all the samples having type=chipseq and containing the attribute cell. This selection will be compiled producing the following meta condition:

\[
\text{AND( Predicate("type", EQ, "chipseq"), \\
\hspace{1em} ContainAttribute("cell") )}
\]

The following is the procedure executed by the implementation in order to apply the meta selection on the source metadata.

- The conditions tree is read recursively till the atomic conditions. The evaluation of one single atomic condition produces as result the list of selected samples.

  - The Predicate conditions, with equality operator, are evaluated filtering all the metadata having the required name and value.

    \[
    \begin{align*}
    \{\text{sid}\} & \text{name, value} \\
    \{1\} & \text{‘type’, ‘chipseq’} \\
    \{2\} & \text{‘type’, ‘chipseq’}
    \end{align*}
    \]

  - The ContainAttribute conditions, are similar to the previous ones but filtering just on the attribute values.

    \[
    \begin{align*}
    \{\text{sid}\} & \text{name, value} \\
    \{2\} & \text{‘cell’, ‘hela-s3’}
    \end{align*}
    \]

- Then the atomic conditions are composed: the and and or predicates have to be interpreted as intersection and union of the partial results.
With AFL these two operations are realized with the cross_join and the merge functions.

\{sid\} selected
\{2\} true

• The final step is generate the result. The output has to be composed by all the metadata of the selected samples. To use the previous result as "filtering list" for the source array, we have to join them by sample id. Then the result is redimensioned according to the standard schema.

\{nid_1,nid_2,vid_1,vid_2,sid\} name,value
\{10200,65266,88441,34749,2\} 'average', '0.0'
\{51397,25895,88642,62213,2\} 'type', 'chipseq'
\{55956,45049,40723,11284,2\} 'antibody', 'BRD4'
\{55956,45049,10988,47850,2\} 'antibody', 'GTF2i'
\{60523,56972,64604,81920,2\} 'cell', 'hela-s3'
\{68488,62526,51166,23206,2\} 'organism', 'mm9'
\{68488,62526,58003,97420,2\} 'organism', 'hg19'

**SelectRD**

The meta selection, if defined, works filtering entire samples, this means that meta conditions have to be applied on regions, too: just the regions of the meta-filtered samples have to be considered for the result. Region conditions are then applied on the remaining regions. Let’s consider again the previous example. Starting from the samples filtered in the previous example, we want to select just the regions having a score greater than 0.4 and chromosome 4.

• To apply the meta filter on the regions, too, the metadata array is redimensioned using just the sample id as dimension and joined with the source region array to filter it and obtain all the regions contained in the selected samples.

\{sid,chr,left,right,strand,x\} score
\{2,4,1129,1195,1,8\} 0.585989
\{2,3,2153,2512,1,3\} 0.590382
\{2,3,4088,4161,1,4\} 0.990850
\{2,2,6115,6470,1,0\} 0.135736
\{2,4,6477,6506,1,5\} 0.266118
\{2,4,7040,7150,1,7\} 0.447849
\{2,2,8052,8300,1,1\} 0.677076
Now the region conditions are evaluated. In order to optimize the execution of the selection, we divide into two steps conditions applied on coordinate values and conditions on feature values. In the first step we can exploit the between operator that, as explained in section 4.4.2, directly select the data without having to actually read them.

\{sid, chr, left, right, strand, x\} score
\{2, 4, 8400, 8885, 1, 6\} 0.389102
\{2, 2, 9564, 9863, 1, 2\} 0.943776

Then are applied the remaining conditions composing a boolean expression used as parameter for the filter operator.

\{sid, chr, left, right, strand, x\} score
\{2, 4, 1129, 1195, 1, 8\} 0.585989
\{2, 4, 6477, 6506, 1, 5\} 0.266118
\{2, 4, 7040, 7150, 1, 7\} 0.447849
\{2, 4, 8400, 8885, 1, 6\} 0.389102

6.4.2 Extend

With the EXTEND operator it’s possible compute aggregation values based on regions and store them as new metadata of the samples. Although conceptually simple, it requires the expensive computation of formulas on the entire region array. The aggregation is one of the most critical operation for a database system, for this reason we propose this operation in order to evaluate the performance of SciDB. Figure 6.11 presents the DAG nodes that compose the operation.

AggregateMD

The AggregateMD node is the only component of the DAG that generate metadata starting from regions. It receives as parameter a list of aggregation functions that has to be executed on each single sample. Let’s see an example requiring the count of the regions and the maximum score value for the classic sample dataset.

67
For each function is generated a single **aggregate** AFL command grouping by sample ids. We can not generate a single command with all the functions because then would be impossible transform the resulting array to produce metadata in the standard form.

```
{sid} aggr_count
 {1} 9
 {2} 9
{sid} aggr_max
 {1} 0.958433
 {2} 0.990850
```

Then are applied, for each result, the name defined by the user for the new metadata. Here the aggregation values are converted to strings according to the standard data model.

```
{sid} name,value
 {1} 'region_count','9'
 {2} 'region_count','9'
{sid} name,value
 {1} 'max_score','0.958433'
 {2} 'max_score','0.990850'
```

Finally the partial results are merged together and, after the generation of the name and value ids, redimensioned in the standard form.

```
{nid_1,nid_2,vid_1,vid_2,sid} name,value
 {15287,32417,47763,59686,1} 'region_count','9'
 {15287,32417,47763,59686,2} 'region_count','9'
```
ExtendMD

The ExtendMD in general unifies two different metadata arrays. In this case is used to add the new aggregation values to the original metadata structure. The following is the result obtained for the example used before.

```plaintext
{nid_1,nid_2,vid_1,vid_2,sid} name,value
{10200,65266,88441,34749,1} 'average', '0.0'
{10200,65266,88441,34749,2} 'average', '0.0'
{15287,32417,47763,59686,1} 'region_count', '9'
{15287,32417,47763,59686,2} 'region_count', '9'
{51397,25895,88642,62213,1} 'type', 'chipseq'
{51397,25895,88642,62213,2} 'type', 'chipseq'
{51397,25895,71479,47349,1} 'type', 'chiapet'
{54141,12575,40219,17307,1} 'max_score', '0.958433'
{54141,12575,33159,66975,2} 'max_score', '0.990850'
{55956,45049,40723,11284,2} 'antibody', 'BRD4'
{55956,45049,10988,47850,2} 'antibody', 'GTF2i'
{60523,56972,64604,81920,2} 'cell', 'hela-s3'
{68488,62526,35912,43235,1} 'organism', 'hg18'
{68488,62526,51166,23206,2} 'organism', 'mm9'
{68488,62526,58003,97420,1} 'organism', 'hg19'
{68488,62526,58003,97420,2} 'organism', 'hg19'
```

6.4.3 Map

The GenometricMap is one of the domain-specific operator (see section 2.2.4). It is a binary operator over the reference and the experiment dataset; it aggregates values of the experiment regions that intersect each reference region. Figure 6.12 shows the nodes generated in the DAG to realize the complete map operation.

The following subsections explain in detail each translation node. In order to show an example of the execution we have to define a new dataset that will be used as reference. The dataset presented at 6.2.4 will be used as experiment.

```plaintext
{nid_1,nid_2,vid_1,vid_2,sid} name,value
{68488,62526,35912,43235,1} 'organism', 'hg18'
{68488,62526,51166,23206,2} 'organism', 'mm9'

{sid,chr,left,right,strand,x} score
{1,3,3000,4000,1,1} 0.285989
```
MetaJoin

The MetaJoin node has to define which sample of the experiment has to be mapped on a certain sample of the reference. The join-by conditions are expressed as a list of attribute names: for each reference sample, just samples in the experiment with the same values for the listed attributes have to be mapped on it.

Let’s see an example: consider a map operation requiring meta-join on the same organism attribute. In our example these condition means that sample 1 of the experiment has to be mapped on sample 1 of the reference (value hg18), and sample 2 on reference sample 2 (value mm9).

The result produced by a MetaJoin node is an array with the schema presented in 6.3.1, having as dimensions the ids for each valid pairs of reference and experiment samples, and as attribute the id assigned to the resulting sample. Follows the procedure used to generate the described result.

- If the condition list is empty, the result is generated joining all the sample ids for the reference with all the sample ids for the experiment. In this case all the experiment regions will be mapped on each reference.

- Otherwise, for each condition (attribute name), metadata of both reference and experiment are filtered according to the attribute name and redimensioned in order to have just value ids and sample id as dimensions.
Then the obtained arrays are joined on the same value ids in order to produce the pairs of samples that satisfy the current condition. The result is then redimensioned and the new id is computed.

Finally each single condition result is joined with the others in order to compute the intersection of the pairs generated by each single condition. In our example we have one single condition and the last step is not applied.

**CombineMD**

The CombineMD node generates the metadata according to the meta-join conditions. For each pair computed by the previous node, a new sample is generated and metadata of the contributing samples are copied and merged. Let’s see the procedure implemented by the translation node.

The meta-join array is joined with the one representing the reference metadata according to the $ANCHOR$ sid. In this way the metadata are replicated once for each resulting sample involving a certain reference one. Then the $sid$ dimension is replaced with the new id.

The same operation is repeated on the experiment metadata, obtaining the attributes of the experiment contributing to each result.
Finally the two partial results are merged into the same array obtaining the metadata for the new dataset. Colliding values are not replicated.

The GenometricMapRD node is the real core of the map operation. It is based on the region intersection computation and this introduce a huge complexity dealing with big data. In section 4.5.1 was presented the binning strategy proposed by Paradigm4 in order to speed up this operation. To better understand the steps required to produce the result in SciDB, we now present the mapping procedure without binning strategies, that will be treated in depth in the next chapter.

- The first step of the computation is find all the intersecting regions according to the meta-join samples pairs.
• After a convenient redimension, the meta-join array is joined with the region array of reference on same sample id. Doing that we are replicating the reference regions once for each sample in the result.

\{ANCHOR$sid, EXPERIMENT$sid, chr, ANCHOR$x\}
RESULT$sid, ANCHOR$score, ANCHOR$left, ANCHOR$right, ANCHOR$strand
\{1,1,3,1\} 10001,0.285989,3000,4000,1
\{1,1,2,2\} 10001,0.437849,3500,4200,1
\{2,2,3,1\} 20002,0.575989,2230,4500,1
\{2,2,4,2\} 20002,0.647649,7100,9200,1

• Then the previous array is joined with the experiment region array, on same sample id and chromosome. The result is a list of region pairs.

\{ANCHOR$sid, EXPERIMENT$sid, chr, ANCHOR$x, EXPERIMENT$x\}RESULT$sid, ANCHOR$score, ANCHOR$left, ANCHOR$right, ANCHOR$strand, EXPERIMENT$score, EXPERIMENT$left, EXPERIMENT$right, EXPERIMENT$strand
\{1,1,2,2,0\} 10001,0.437849,3500,4200,1,0.477766,3292,3701,1
\{1,1,2,2,1\} 10001,0.437849,3500,4200,1,0.940126,7024,7326,1
\{1,1,3,1,3\} 10001,0.285989,3000,4000,1,0.818601,1743,2222,1
\{1,1,3,1,6\} 10001,0.285989,3000,4000,1,0.755845,3137,3381,1
\{1,1,3,1,5\} 10001,0.285989,3000,4000,1,0.958433,3406,3625,1
\{1,1,3,1,4\} 10001,0.285989,3000,4000,1,0.499267,3856,4213,1
\{1,1,3,1,7\} 10001,0.285989,3000,4000,1,0.943462,9091,9317,1
\{1,1,3,1,2\} 10001,0.285989,3000,4000,1,0.718422,9815,9918,1
\{2,2,3,1,3\} 20002,0.575989,2230,4250,1,0.590382,2153,2512,1
\{2,2,3,1,4\} 20002,0.575989,2230,4250,1,0.990850,4088,4161,1
\{2,2,4,2,8\} 20002,0.647649,7100,9200,1,0.585989,1129,1195,1
\{2,2,4,2,5\} 20002,0.647649,7100,9200,1,0.266118,6477,6506,1
\{2,2,4,2,7\} 20002,0.647649,7100,9200,1,0.447849,7040,7150,1
\{2,2,4,2,6\} 20002,0.647649,7100,9200,1,0.389102,8400,8885,1

• Once obtained all the possible regions pairs, the array is filtered considering just the pairs the really intersect and with compatible strands.

\{ANCHOR$sid, EXPERIMENT$sid, chr, ANCHOR$x, EXPERIMENT$x\}RESULT$sid, ANCHOR$score, ANCHOR$left, ANCHOR$right, ANCHOR$strand, EXPERIMENT$score, EXPERIMENT$left, EXPERIMENT$right, EXPERIMENT$strand
\{1,1,2,2,0\} 10001,0.437849,3500,4200,1,0.477766,3292,3701,1
\{1,1,3,1,6\} 10001,0.285989,3000,4000,1,0.755845,3137,3381,1
\{1,1,3,1,5\} 10001,0.285989,3000,4000,1,0.958433,3406,3625,1
\{1,1,3,1,4\} 10001,0.285989,3000,4000,1,0.499267,3856,4213,1
\{2,2,3,1,3\} 20002,0.575989,2230,4250,1,0.590382,2153,2512,1
\{2,2,3,1,4\} 20002,0.575989,2230,4250,1,0.990850,4088,4161,1
\{2,2,4,2,5\} 20002,0.647649,7100,9200,1,0.266118,6477,6506,1
\{2,2,4,2,7\} 20002,0.647649,7100,9200,1,0.447849,7040,7150,1
\{2,2,4,2,6\} 20002,0.647649,7100,9200,1,0.389102,8400,8885,1
• After the generation of the intersecting regions, the second step of the map is aggregate values based on experiment features. Let’s consider for example a query requiring the count of intersecting regions for each reference. If the previous intermediate result is redimensioned on the reference coordinates, each pair generated involving this region will collide. Exploiting the possibility to define aggregation functions for the colliding values of a `redimension` statement, we obtain the values required by the map operation on the intersecting regions.

{sid,chr,left,right,strand,x} score,count
{10001,3,3000,4000,1,1} 0.285989,3
{10001,2,3500,4200,1,2} 0.437849,1
{20002,3,2230,4250,1,1} 0.575989,2
{20002,4,7100,9200,1,2} 0.647649,2
Chapter 7

Bidimensional Binning

The genometric map operation, presented in section 6.4.3, is based on range intersection of the genomic regions. This operation needs to check all the possible region pairs in order to find the intersecting ones. In the presented procedure was used a cross join to generate these pairs.

It is known that a complete cross product of two datasets, dealing with really big data, is not suitable because the execution time will increase exponentially. For this reason, in various big data applications, were designed binning strategies in order to split up the work, run the operations on smaller local data, exploiting parallelism, and then recombine the final result.

In this chapter we will present a new binning strategy, called bidimensional binning, designed to speed up the range intersection operation. Section 7.1 introduces the motivations for the new strategy and the design directives. Section 7.2 presents the representation adopted for region data. Section 7.3 explains the method used to reduce the search space and select just the regions really interested in the intersection. Finally, section 7.4 is dedicated to the implementation of the method and the exploiting of the parallelism.

7.1 Motivations

In section 4.5.1 was introduced the binning strategy proposed by Paradigm4. This method, that we call monodimensional binning, splits the genome space into small bins, assigns to the bins the regions, eventually copying them, and then runs the operation on each single bin. To avoid information loss, the bin size has to be greater than (or equal to) the maximum region length and each region crossing the bin borders has to be replicated.

This method can be generalized, to allow smaller bin sizes, increasing
Figure 7.1: Monodimensional binning using different replication factors.

The number of replications. Equation 7.1 shows the possible values for the bin size replicating the regions \( r \) times.

\[
\text{bin} \_\text{size} \geq \frac{\text{max} \_\text{length}}{r - 1} \quad (7.1)
\]

The presented strategy is the classical binning method, adopted by several applications on various technologies (our implementations of GMQL on Spark and Flink use it [5]).

In general, to implement this technique, is enough define the bin size and replicate regions once for each intersected bin. Using SciDB we have to apply the same operation on every cell in the array, hence, a dynamical replication is not feasible. For this reason the replication factor has to be previously defined and it limits the possible bin size values. Figure 7.1 shows the effect of different replication factors: each cell is replicated \( r \) times and each copy is assigned to a bin or discarded, if not necessary. Optimize the monodimensional binning means evaluate the trade off between replication cost and speed up obtained with a smaller binning.

Our intention is design a new binning strategy considering the SciDB constraints and trying to exploit its storage management, joining just regions with an high probability of intersection.
7.2 Representation

The regions array (section 6.2.2) was defined as a 6-dimensional structure: sample id, chromosome id, left end, right end, strand id and a enumeration value. Considering sample, chromosome and strand ids as classification values, we can project the regions as points in a bidimensional space, using left and right ends as axes. Figure 7.2 shows an example of the plan obtained from a dataset. By definition, all the points stay above the primary diagonal because the right end value has to be greater than the left one.

\[
\text{bin}(x) = \left\lfloor \frac{x}{\text{bin\_size}} \right\rfloor
\]  

(7.2)

Using this representation, the splitting of the genome space into bins generates a grid on the plan. Equation 7.2 shows the function used to split the genome. We call bidimensional bin (or box) each ”box” of the grid. A point falling in the bidimensional bin \((n,m)\) means that the corresponding region starts in the \(n\)-th bin and end in the \(m\)-th one. It should be noted that each region is assigned to one, and only one, bidimensional bin, and that it’s sufficient to know its end points to find the right box (equation 7.3).

\[
\text{box}(P) = \left( \text{bin}(P_{\text{left}}), \text{bin}(P_{\text{right}}) \right)
\]  

(7.3)
The last consideration about the representation is on the maximum region length. If we know this value we can assert that every box above the diagonal parallel passing for the longest region point will be for sure empty. This observation will be useful in section 7.4.

### 7.3 Search space window

Let’s consider now two datasets that you want to use for a genometric map operation. Figure 7.3 shows the bidimensional representation for the reference and the experiment datasets.

Using this representation, for each box in the reference we can divide the experiment regions into three groups: the ones that for sure intersect all the reference region in the box, the ones that may intersect, and the ones that for sure doesn’t intersect. Exploiting this partitioning, we can reduce the search space for each reference box to a specific window in the experiment that includes just the regions that actually can intersect the first ones.

Let’s use as example the box (2, 3) containing the point $P$ highlighted in the figure. $P$, as all the points in the same box, starts somewhere in the second bin and finishes in the third one. It means that it crosses the border between bins 2 and 3. Two regions intersect if they share at least one point of the linear space; hence, all the regions in the experiment crossing the boundary between bins 2 and 3 for sure intersect the region represented by $P$. A region crosses the 2-3 boundary if starts before the bin 3 and ends after the bin 2. The following theorem generalizes this condition.
Theorem 1. Let \( R = (l_R, r_R) \) be a box in the reference and \( E = (l_E, r_E) \) a box in experiment, then the regions represented in \( R \) intersect all the regions represented in \( E \) if the following sufficient condition is verified:

\[
(l_E < r_R) \land (r_E > l_R)
\]  

(7.4)

Applied to the previous example, the theorem says that every box \((l, r)\) in the experiment having \( l < 3 \) and \( r > 2 \), contains regions that surely intersect the point \( P \). The condition of theorem 1 can be graphically represented on the experiment plan. Figure 7.4a shows the window for our example.

In the same way we can define the window of the regions that may intersect the reference. Recalling the example, all the regions that ends within bin 2, the bin of \( P \) left end, are good candidates for the intersection. All the region starting from bin 3 have high probability of intersection, too. Theorem 2 generalize these conditions.

Theorem 2. Let \( R = (l_R, r_R) \) be a box in the reference and \( E = (l_E, r_E) \) a box in experiment, then the regions represented in \( E \) are good candidates for intersection with the reference if the following condition is verified:

\[
(l_E = r_R) \lor (r_E = l_R)
\]  

(7.5)

The application of the theorem in our example asset that every box \((l, r)\) contains good candidates if \( l = 3 \) or \( r = 2 \). In this case the graphical representation generates 2 rectangular areas (figure 7.4b).
The previous theorems can be synthesized into a single condition. Considering that all the regions stay above the primary diagonal, by definition all the boxes \((l, r)\) with \(l > r\) are empty. The following is the theorem that define the search space window for a specific reference box.

**Theorem 3.** Let \(R = (l_R, r_R)\) be a box in the reference. The search space window that defines the subset of the experiment regions that are actually required to be checked in order to verify the intersection, is composed by all the boxes \(E = (l_E, r_E)\) that verify the following condition:

\[
(l_E \leq r_R) \land (r_E \geq l_R)
\]  

(7.6)

**Proof.** To prove the theorem is sufficient consider all the possibilities that make the condition false and see what them imply. There are three possible conditions that turn false the expression 7.6 (remember that \(r \geq l\) for every non-empty boxes)

- \((l_E > r_R)\). In this case all the experiment regions will start after the end of the reference, without therefore intersect it.

- \((r_E < l_R)\). In this case the experiments will end before the begin of the reference, preventing the intersection.
• \(( l_E > r_R \) \land ( r_E < l_R )\). This case is impossible because the condition can be true just for regions with \( l > r \), and they are not allowed by definition.

Figure 7.5 shows the search space window obtained for our example. The final window is a rectangular area: this is an important feature for the implementation because, as will be discussed in the next section, enable the exploitation of the SciDB characteristics.

7.4 Implementation

The bidimensional binning basically defines, for each portion (box) of the reference, the minimum portion of the experiment array that should be used to compute the range intersection. In this way the map operation can be applied independently on each couple of sub-arrays. Figure 7.6 shows how is executed the genometric map using this strategy.

In order to speed up the execution it’s possible exploit the parallelism, opening one thread for each non-empty box in the reference. We define as non-empty all the boxes between the primary diagonal of the plan and the parallel passing by the longest region point. Each thread then selects the regions of the assigned box, the ones of the corresponding window from experiment and run the map operation.

The selection of the sub-arrays is optimized by the native structure of SciDB: the MAC storage system and the between operator enable a direct access to the selected regions without a complete read of all the array.

In order to use this kind of execution inside our GMQL implementation, it was necessary change a little bit the proposed architecture. When the GenometricMapRD node is translated, the main thread of the translation is suspended, then it is executed the map, and when the intermediate result was materialized, the main thread can be recovered. In this way the usage of the bidimensional binning on the nodes that require a range intersection doesn’t modify the general behaviour of the module.
Figure 7.6: Execution schema on map using bidimensional binning.
Chapter 8

Experiments

This chapter is dedicated to the experimental results obtained testing the new implementation. The first step was verify the correctness of the system. We used a predefined set of queries that were executed using both the SciDB and Spark implementation. From the comparison comes out that the new SciDB-based system implementation is correct and produces the expected outputs.

Then we have defined performance tests that we have executed on Amazon EC2 machines. Section 8.1 presents the details of the execution environment. The first test, section 8.2, aims to compare the performance obtained by the usage of SciDB and Spark for the execution of GMQL queries. The second experiment, section 8.3, is focused on the evaluation of the two binning techniques applied to the SciDB implementation.

8.1 Execution environment

All the tests were executed on Amazon Elastic Compute Cloud (EC2) machines. Both for SciDB and Spark were required r3.4xlarge (memory optimized) instances, with 16 virtual CPUs, 122 GB of memory and 1x320 GB of SSD storage.

For the tests on the SciDB implementation we used the AMI (Amazon Machine Image) provided by Paradigm4 on the North Virginia data center. The image provides SciDB 15.12 already installed on a Ubuntu 14.04.2 LTS machine.
8.2 Technology comparison

Before the realization of this work, GMQL was already implemented on Spark and Flink frameworks. Both of them are based on the Map-Reduce paradigm and their performance comparison was treated in [5]. In order to compare the results obtained by our implementation on SciDB, in terms of execution time, we have designed an experiment suit to compare our work with the Spark implementation. The decision to use Spark instead of Flink is due to the fact that the developers of GMQL have decided to focus more on this platform.

Considering the structure of each GMQL operation and the frequency of their usage on the deployed instances of the system, we have decided to focus the technology comparison on three different kind of operations: the filtering of the regions, the aggregation of values and the range intersection. The following sections present the GMQL queries used to realize these operation and the execution time obtained on various datasets.

8.2.1 Datasets

For the comparison of the technologies, we have decided to use synthetic datasets. The generation of the regions was configured to obtain data similar to the real cases: each dataset contains a total of 50K regions on 22 different chromosomes, the regions are distributed uniformly on the genome with a length between 100 and 500 bases. The following is the table containing the information about the used datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Regions</th>
<th>Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>REF</td>
<td>2.3 MB</td>
<td>50 600</td>
<td>1</td>
</tr>
<tr>
<td>DS_1</td>
<td>3.8 MB</td>
<td>101 200</td>
<td>2</td>
</tr>
<tr>
<td>DS_2</td>
<td>38.0 MB</td>
<td>1 012 000</td>
<td>20</td>
</tr>
<tr>
<td>DS_3</td>
<td>375.0 MB</td>
<td>10 120 000</td>
<td>200</td>
</tr>
<tr>
<td>DS_4</td>
<td>3.76 GB</td>
<td>101 200 000</td>
<td>2000</td>
</tr>
</tbody>
</table>

The datasets are designed for evaluating the scale-up of performance for the various operations.

8.2.2 Region filtering

The first experiment is about the filtering and the selection of regions: Good performances in data fetching are important for all the operators. In order to test it we can use the SELECT operator of GMQL. The following are the three queries used for the test.
Q1: SELECT(chr='chr1') DS_X
Q2: SELECT(score>0.9) DS_X
Q3: SELECT(chr='chr1' and score>0.9) DS_X

Results (in seconds) obtained running the query on the four datasets of the experiment are reported in the following table. Figure 8.1 shows the corresponding graphs, for the size axis we used the logarithmic scale to better understand the difference between the behaviours of the two technologies.

<table>
<thead>
<tr>
<th>Test</th>
<th>DS_1 time(s)</th>
<th>DS_2 time(s)</th>
<th>DS_3 time(s)</th>
<th>DS_4 time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 SciDB</td>
<td>0.110s</td>
<td>0.136s</td>
<td>0.385s</td>
<td>4.515s</td>
</tr>
<tr>
<td>Q1 Spark</td>
<td>4.391s</td>
<td>6.063s</td>
<td>9.403s</td>
<td>43.645s</td>
</tr>
<tr>
<td>Q2 SciDB</td>
<td>0.161s</td>
<td>0.581s</td>
<td>5.673s</td>
<td>58.137s</td>
</tr>
<tr>
<td>Q2 Spark</td>
<td>4.640s</td>
<td>6.447s</td>
<td>10.299s</td>
<td>46.049s</td>
</tr>
<tr>
<td>Q3 SciDB</td>
<td>0.123s</td>
<td>0.140s</td>
<td>0.284s</td>
<td>2.035s</td>
</tr>
<tr>
<td>Q3 Spark</td>
<td>4.478s</td>
<td>6.145s</td>
<td>9.813s</td>
<td>44.015s</td>
</tr>
</tbody>
</table>

Looking the results, there are big differences using SciDB on Q$S1$ and Q$S3$ respect to Q$S2$. This result is explained by the fact that SciDB implementation exploits the between operator when compiles the first and the last queries. Comparing the results, when SciDB can access to the data using
the optimized between operator, it outperforms Spark, that always requires a entire reading of the data to produce the output. On the contrary, when even SciDB is forced to read each single cell to apply the filtering operation, the execution times of SciDB and Spark are similar, and in one case, with the largest dataset, Spark is slightly faster than SciDB.

8.2.3 Aggregation

The second experiment is on the aggregation which is used by many operators in GMQL (e.g. MAP, COVER, EXTEND); we focus on EXTEND. We have tested various aggregation functions, but came out that there are no differences in execution time. The following is the query used for the test.

\[ Q4: \text{EXTEND(count(\ast)) DS}_X \]

Results (in seconds) obtained running the query on the four datasets of the experiment are reported in the following table. Figure 8.2 shows the corresponding graphs, for the size axis we used the logarithmic scale as in the first test.

<table>
<thead>
<tr>
<th>Test</th>
<th>DS_1</th>
<th>DS_2</th>
<th>DS_3</th>
<th>DS_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q4 ) SciDB</td>
<td>0.155s</td>
<td>0.169s</td>
<td>0.307</td>
<td>1.747s</td>
</tr>
<tr>
<td>( Q4 ) Spark</td>
<td>10.667s</td>
<td>18.730s</td>
<td>29.094s</td>
<td>133.938s</td>
</tr>
</tbody>
</table>

In this case we see a huge difference between the two technologies. SciDB, exploiting its storage management, is not strongly sensitive to the increase of data size, and it’s able to produce the final result in just a few seconds even for big amounts of regions.
8.2.4 Range intersection

Range intersection is the base of the genometric \text{MAP} in GMQL. The map is probably the simples of the three domain specific operators, but the other two, although very different for the user, are not so different in the implementation, and also require range intersection. We decided to base our tests on the map because it uses the most basic range intersection operation.

For the testing, we used 	extit{monodimensional binning} for the SciDB implementation for two reasons: first, it’s the same strategy adopted by Spark, this allows a pure comparison of the technology without algorithmic disparity, second, because it represents the “state of the art” method and we postpone the comparison of monodimensional and bidimensional binning to the next section section. The following is the query used for the test.

\texttt{Q5: MAP(count(\ast)) REF DS_X}

In this case it was necessary perform a tuning of the operation to find the \textit{bin size} value that minimize the execution time on both the technologies.
Figure 8.3a shows the curve obtained running the query on the DS_2 dataset by changing the binning size. The best configuration is 40K bases as binning size.

Results (in minutes) obtained running the query on the four datasets of the experiment are reported in the following table. Figure 8.3b shows the corresponding graphs.

<table>
<thead>
<tr>
<th>Test</th>
<th>DS_1</th>
<th>DS_2</th>
<th>DS_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q5 SciDB</td>
<td>0.15m</td>
<td>1.41m</td>
<td>15.91m</td>
</tr>
<tr>
<td>Q5 Spark</td>
<td>0.12m</td>
<td>0.57m</td>
<td>3.82m</td>
</tr>
</tbody>
</table>

This time the difference between the two technologies was evident already using the DS_3 dataset. Spark always outperforms SciDB, and the growth rate of the execution time is significantly higher.

### 8.3 Binning strategy comparison

The second experiment aims to show the performance obtained using the bidimensional binning, comparing it with the classic monodimensional strategy. Also in this test we use the genometric map as operation for the comparison, simply counting the intersecting regions. In this case we decided to use real data coming from biogenomic projects and see how the strategies work increasing the number of samples. The following subsection describe the used datasets.

#### 8.3.1 Datasets

As reference of the map, we used two datasets of the RefSeq collection provided by the NCBI (National Center for Biotechnology Information). We refer to these datasets as GENES and PROMOTERS. The first one is the collection of regions representing human genes. The length of the regions of this dataset is strongly variable going from 19 bases up to 24Mb, with an average of 60Kb. The second is uniform because artificially generated: a promoter is a region close to the begin of a gene, where the enzymes responsible for DNA transcription bind. These regions are arbitrary defined starting 2000 bases before, and finishing 1000 bases after, the start of each gene. There are more promoters than genes because each gene can have more than one transcript starting site (TSS). The tests reported in the next sections are the same, but we want see the methods on datasets with different characteristics.
For the experiment we used a ChIP-seq dataset provided by the ENCODE Project. It is a narrow peak dataset, so the region length is in the order of hundreds of bases. For our experiments we changed the number of samples of this dataset in order to see how the two binning strategies scale increasing the number of data. How it can be seen, we used relatively small datasets, but they are enough to show the behaviour of the methods.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Regions</th>
<th>Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROMOTERS</td>
<td>1.5 MB</td>
<td>49 052</td>
<td>1</td>
</tr>
<tr>
<td>GENES</td>
<td>0.8 MB</td>
<td>23 033</td>
<td>1</td>
</tr>
<tr>
<td>ENCODE</td>
<td>17 MB</td>
<td>363 537</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>41 MB</td>
<td>938 753</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>57 MB</td>
<td>1 264 764</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>108 MB</td>
<td>2 230 698</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>155 MB</td>
<td>3 227 907</td>
<td>32</td>
</tr>
</tbody>
</table>

### 8.3.2 Promoters

To compare the performance obtained by the two methods it’s necessary define the best configuration for both the strategies. Figure 8.4a reports the tuning test executed on the dataset with 2 samples using the monodimensional binning (we have verified that increasing the number of samples the best configuration doesn’t change). The best bin size obtained with the 2x replication is 3M bases. If the optimal bin size is greater than the maximum region length (3K bases in this case), an higher replication of the regions is useless because it’s not necessary obtain smaller binning sizes. Figure 8.4b reports the results for the binimensional binning. In this case there is just one parameter and the optimal configuration is 10M bases.
Figure 8.5: Execution times of map using PROMOTERS as reference with monodimensional and bidimensional binning.

Results (in minutes) obtained running the test on the experiment datasets using the two optimal configurations are reported in the following table, we reported even some times obtained without binning strategies (0D binning). Figure 8.5 shows the corresponding graphs.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>0D</th>
<th>1D</th>
<th>2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCODE-2</td>
<td>13.07m</td>
<td>1.46m</td>
<td>2.61m</td>
</tr>
<tr>
<td>ENCODE-4</td>
<td>26.32m</td>
<td>2.75m</td>
<td>4.01m</td>
</tr>
<tr>
<td>ENCODE-8</td>
<td>5.31m</td>
<td>4.77m</td>
<td></td>
</tr>
<tr>
<td>ENCODE-16</td>
<td>12.22m</td>
<td>6.06m</td>
<td></td>
</tr>
<tr>
<td>ENCODE-32</td>
<td>36.83m</td>
<td>8.93m</td>
<td></td>
</tr>
</tbody>
</table>

The results prove that the bidimensional binning has a greater constant time, because the number of open threads depends on the characteristic of the reference and not on the size of the experiment. Anyway the big difference between the two methods complexity fills the gap rapidly and 2D binning starts to outperform the classical 1D strategy already with 1 million of regions.
8.3.3 Genes

Now we propose the same experiment using the GENES dataset as reference. With this test we want to see if there are differences due to the nature of the used datasets. Figures 8.6a and 8.6b report the test on the optimal binning configuration using the new reference. For dimensional binning we obtain a similar curve with the minimum on 16Kb. The monodimensional binning shows now a different behaviour: the maximum region length is 25Mb and more replications are required to obtain smaller bin sizes. The graph shows the curves produced with various replication factors (remember that the replication factor introduces a lower bound limit for the bin size). We obtained as best configuration 4Mb bins with 8 replications. Using the new optimal configuration we executed the test on the experiment datasets. Follows the table with the execution times (in minutes). Figure 8.7 shows the corresponding graphs.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Binning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0D</td>
</tr>
<tr>
<td>ENCODE-2</td>
<td>6.35m</td>
</tr>
<tr>
<td>ENCODE-4</td>
<td>13.23m</td>
</tr>
<tr>
<td>ENCODE-8</td>
<td>18.20m</td>
</tr>
<tr>
<td>ENCODE-16</td>
<td>8.70m</td>
</tr>
<tr>
<td>ENCODE-32</td>
<td>23.52m</td>
</tr>
</tbody>
</table>

The results of this second experiment follow the previous ones: the bidimensional strategy pays an higher constant time but has a lower growth rate. This result means that the two technologies can be configured and optimized to work in the same way on different kind of datasets, but for the monodimensional binning, when the maximum region length starts to
generate strong limitations on bin sizes, requires a deeper analysis of the best working condition, having to define the replication factor in addition to the binning size.

Figure 8.7: Execution times of map using GENES as reference with monodimensional and bidimensional binning.
Chapter 9

Related Work

9.1 Genomic Data Management

Several organizations are considering genomics at a global level. Global Alliance for genomics and Health (GA4GH) is a large consortium of over 200 research institutions with the goal of supporting voluntary and secure sharing of genomic and clinical data; their work on data interoperability is producing a conversion technology for the sharing of data on DNA sequences and genomic variation\(^1\). Google recently provided an API to store, process, explore, and share DNA sequence reads, alignments and variant calls, using Google’s cloud infrastructure\(^2\).

We compare GMQL with recent papers on genomic data management. Works by Röhm and Blakeley [27], by Tata, Patel et. al. [35], [36], and by Bafna et al. [1], [21] address the querying of NGS data using either SQL (in the former case) or SQL extensions (in the latter two cases). Use of plain SQL was attempted in [27], which highlights the performance bottlenecks of conventional SQL optimization when dealing with domain-specific functions and parallelization.

Tata, Patel et. al. developed Periscope [35], [36], a system supporting matching operators over DNA, encoded as character strings; they report very fast execution times. Several domain-specific extensions are proposed in the language GQL [1] to overcome SQL limitations in expressing genomic computations. Each of this systems highlights specialized processing for given aspects (e.g., string matching) and internally performs genome processing tasks which are also provided by ad-hoc specific tools (e.g., alignment and mutation calling). Carrying out such tasks from within an integrated

\(^1\)Global Alliance Genomics API. http://ga4gh.org/#/documentation
system is potentially very effective, as no information is left outside of the
data management system, but it typically limits the types of supported
queries; moreover, many widely available repositories (e.g., [8],[9],[38]) pro-
vide datasets resulting after the calling processes. Neither Periscope nor
GQL integrate metadata within their computation, which is only addressed
to genomic data.

Other works have proposed the embedding of query processing functions
within libraries that can be integrated with programs [7], [26]. In particular,
[26] presents a rather elegant mathematical formalism, based on set algebra;
its authors propose a genomic region abstraction (that may represent reads,
genomic variants, mutations, and so on) and then define a set of region op-
erations, delivered as the Genomic Region Operation Kit (GROK) library.
In comparison, GROK supports lower-level abstractions than our GMQL
and some low-level operations (e.g., flipping regions) that are not directly
supported by our GMQL, but they must be embedded into C++ program-
ing language code. Furthermore, high-level declarative operations, such
as JOIN and MAP, can be encoded in GROK, but they must be invoked from
line editors or C++ programs. GROK shows excellent performance on desktop
systems, but it is unsuitable for parallelization and does not deal with
metadata.

Several other tools focus on specific data formats or tackle specific needs
and processing requirements; among them, BEDOPS and BEDTools apply
to the BED format; as discussed in the previous section, they are not
designed for cloud computing, as they are used from within software en-
vironments for bioinformatics (e.g., BioPerl, BioPython, R, Bioconductor,
etc.).

Recent work by Nordberg et al. [25] presents BioPig, a set of Pig exten-
sions for specific analysis tasks. BioPig includes three modules that can be
used in the early phase of data analysis for processing the raw data files pro-
duced by NGS machines. SeqPig [29] is a similar framework also based on
Pig, while SparkSeq [39] presents data analysis tasks implemented on Spark
[40]. All these works are complementary to our, and indicate a growing
interest in parallel processing for genomics.

9.2 Scientific Data Management Systems

In 80’s, the scientific and database communities started to identify and for-
malize the general characteristics of different scientific areas [30, 31]. This
research produced the requirements for statistical and scientific databases,
including physical and logical organization, memory access methods and
query operators. The analysis of the applications led to the conclusion that SQL, even using ad-hoc extensions, would be inappropriate for many scientific fields, such as geographical information systems (GIS) [13] and biological data [11]. Maier et al. [22] understood that the key problem of relational DBMS in scientific applications is the lack of support for ordered data structures, like multidimensional arrays and time series. The database community responded by proposing various new DBMS supports, extensions of the standard SQL language and algebraic frameworks for ordered data.

There are a lot of projects focused on these objectives, but few systems can handle sizable arrays efficiently. ArrayDB [23] is a prototype array database system, which is mainly used for processing small 2D images. MonetDB [37] is a column-store DBMS for spatial applications, now it supports SciQL [20] as query language. RasDaMan [3] is a domain-independent multidimensional array DBMS. It decomposes arrays into tiles stored as BLOBS in a conventional DBMS, tiles are the storage and access units. RasDaMan provides a SQL-92 based query language RasQL [2] to manipulate the structures. The main server acts as middleware for the fetching of the right tiles and the execution of RasQL queries, after the translation into the "BLOB semantic".

SciDB [6], which was used in this thesis, is a recent array data management system developed by Paradigm4 and designed to fit exactly the need of the science community. It divides the arrays into chunks and distributes them on a cluster of instances using hashing functions based on dimensions values. Each cell can contain a set of attributes stored using a column-based strategy.
Chapter 10

Conclusions and Future Work

In this thesis we developed a new implementation of GMQL, a new language for querying NGS genomic data. We used SciDB, a data management system for scientific applications which provides abstraction from storage management, data distribution, and optimized parallel execution. Other implementations of GMQL use cloud computing frameworks (Pig, Flink and Spark) to execute operations directly on files; the SciDB implementation covers all the GMQL language and represents a complete and stable alternative. In comparison, we obtained better performance on various operators which can exploit the data model of SciDB, but worse performance on massive operations (such as map and join), where the implementation based on Spark is faster. However, a lot of options exists for improving our baseline SciDB implementation.

Along this objective, we focused on a new abstractions for parallelism; we realized that parallelism of massive operations on the genome requires binning, i.e. the partitioning of the genome into portions so that operations are performed in parallel at each bin. We then studied the monodimensional binning supported by the current SciDB implementation, and we designed bidimensional binning, an alternative strategy also applicable to the SciDB data model. We focused on range intersection, where the challenge is to build a binning strategy comparing as few regions as possible, by limiting the comparison just to the regions that can intersect. In comparison, bidimensional binning obtains better performance on several datasets and scales better with data of increasing size.

The future development of the project will be mainly divided into two threads. First, we want to make this architecture more stable, so as to
make it available to users, deployed on the servers which support GMQL (currently, the most powerful site is hosted at Cineca, a Consortium for parallel and high performance computing which hosts products from many Italian universities.) Second, we want to focus on research for improving the performance of our baseline SciDB implementation, by further improving the support of massively parallel operations and by extending optimization to other GMQL operators. We next discuss the optimization opportunities that we foresee.

A first area for improvement is metadata management. Given that metadata are typically two-three order of magnitude smaller than region data, we can consider alternative implementations of metadata; in particular, we think that an hybrid solution with in-memory technologies for the management of metadata could considerably speed up the execution. In addition, we are currently designing a specific optimization of GMQL translation, called metadata first, that consists of executing all DAG nodes on the metadata before considering any DAG node on the regions; such optimization is possible only when the query has specific properties, but can lead to huge performance gains, as the region data that do not contribute to the results would not even be loaded. Such optimization applies to all GMQL implementations, including the one using SciDB.

Another important area for improving the performance is the exploitation of materialization options, that are feasible with particular queries. The database community has already produced various algorithms and techniques to understand which node of a DAG has to be materialized to obtain better performance. In future we plan to further analyze GMQL operators in order to define precise cost functions that will enable the utilization of such solutions. We expect that intermediate results could be materialized whenever a binary operation has unbalanced input operands, so that the small-size operand could be materialized and copied to several nodes.

Finally, for what concerns the bidimensional binning strategy, our next step will be development of dedicated AFL operators. We are currently implementing bidimensional binning by opening high-level threads that run AFL queries on single bins, but native AFL operators could exploit the knowledge of the physical data model and of data distribution in order to optimally implement region comparison operations using our binning strategy. Another important goal will be the definition of a predictive model for the optimal binning configuration, based on the datasets characteristics. We also want to work on a more formal definition and analysis of bidimensional binning, by extending its range of applicability and by adapting it to different contexts, including non array-based systems.


