POLITECNICO DI MILANO

Master of Science in Engineering of Computing Systems
Department of Electronics, Informatics and Bioengineering

Heuristic-based Optimizations for Genomic Computing

Bioinformatics Group
Politecnico di Milano

Advisors:
Prof. Stefano Ceri
Dr. Abdulrahman Kaitoua

Master Thesis of :
Andrea Gulino - 836681

Academic Year 2016-2017

To my parents and my beloved friends.

Acknowledgements

This experience would not have been possible without the support of my family, professors, mentors, and friends. To my parents, Gianni and Daniela, thank you for your love and financial support and for motivating me throughout all these years. To prof. Stefano Ceri, for giving me the great opportunity to join his group and for his wise advises. To the guys of the Genomic Computing group at Politecnico for your help and for being more than just co-workers. A special thanks to Abdo, for his contribution to this work, for his inspiring passion for research, and for being a good friend. Thanks to Eirini, for making me smile whenever I need it .

Thanks to Andrea, Chiara, Davide, Fabrizia, Gloria, Marianna, for being the friends I can always count on. Thanks to my colleague and friend Alessandro for helping me and standing me over the last 5 years, especially during the year we spent together in Shanghai. Thanks to my High School teachers for giving me an high quality education. Thanks to my grandmother and to all the other members of my family. Thanks to Danilo, Chris and to all the people who have enjoyed a piece of their life with me but I did not mention by name.

Finally, a special thanks to my little brother, Matteo, and to my younger sister, Francesca. I wish you all the best.

Abstract

Next Generation Sequencing (NGS), a high-throughput, massively parallel technology for reading the DNA, 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, responsible of exploring, querying and integrating processed data, so as to give answers to complex biological and clinical questions, ultimately yielding to personalized medicine.

A new holistic approach for tertiary data analysis has been developed by the Genomic Computing team at Politecnico di Milano. The approach, based on the notion of Genomic Data Model (GDM) and on a new high-level language, called GenoMetric Query Language (GMQL), combines data modeling and management, big data, cloud computing, systems architecture and parallel algorithms into the new Genomic Data Management System (GDMS). Together with the system, new abstractions for parallelism have been introduced; the parallelism of massive operations on the genome is based on binning, i.e. the partitioning of the genome into portions so that operations are performed in parallel at each bin.

Even though much efforts have been made to develop an efficient system, there are still cases in which query performance remains problematic. The main reason behind those performance issues is the lack of high level optimizations that, by reasoning on data and query characteristics, are able to put in place smart optimization strategies.

This thesis focuses on the development on one such optimization; optimal binning. Since experiments demonstrated that the bin size is a critical parameter for the overall performance of domain-specific operations, we developed a mathematical model that, taking into account query and data characteristics, allows to predict a bin size that makes binning efficient.

In order to perform optimal binning, we also introduce Genomic Profiling, which, taking into account the specificity of genomic data modeling, quantitatively defines the properties that better characterize a genomic dataset from the point of view of query optimization.

Genomic Profiling and bin optimization become part of the Optimizer, a new module designed to collect the optimizations developed in this thesis and those optimizations that will come in the future.

Abstract

La Next Generation Sequencing (NGS) è nuova tecnologia per il sequenziamento del DNA che sta cambiando la ricerca biologica e la pratica medica, supportata dalla disponibilità a basso costo di un grande quantità di DNA sequenziato di varie specie, tra cui quella umana. Finora, la comunità di ricerca bio-informatica si è concentrata sull'analisi primaria e secondaria (allineamento e correlazione), ma il problema emergente, negli ultimi anni, è la cosiddetta analisi terziaria, orientata all'esplorazione, interrogazione e integrazione dei dati sperimentali, per dare risposta a complessi quesiti biologici e clinici e per favorire lo sviluppo della medicina personalizzata.

Un nuovo approccio all'analisi terziaria è stato sviluppato al Politecnico di Milano dal team di Genomic Computing. L'approccio, basato sulla nozione di Genomic Data Model (GDM) e su un nuovo linguaggio di interrogazione, chiamato GenoMetric Query Language (GMQL), combina le tradizionali teorie sulla modellazione e gestione dei dati alle moderne tecnologie impiegate per l'analisi dei big data, dando vita ad un moderno sistema chiamato Genomic Data Management System (GDMS).

Con il sistema, sono state introdotte nuove astrazioni per rendere possibile il processamento in parallelo di grandi quantità di dati rappresentati DNA; tra queste il binning, ovvero il partizionamento del genoma in intervalli di eguale misura, chiamati bin, che permettono di decomporre una singola operazione sul genoma in un certo numero di operazioni da eseguire in parallelo su ciascun bin.

Nonostante l'impegno speso per sviluppare algoritmi efficienti, ci sono ancora casi in cui l'esecuzione di query complesse risulta problematica in termini di performance. Queste problematiche sono in generale riconducibili ad una carenza di ottimizzazioni di alto livello in grado di tener conto delle caratteristiche dei dati e delle query sottoposte dall'utente.

Questa tesi sviluppa una di queste ottimizzazioni; l'ottimizzazione del binning. Diversi esperimenti, infatti, hanno dimostrato che la dimensione di ogni bin è un parametro che influenza molto la performance complessiva degli

operatori di dominio. Pertanto, abbiamo sviluppato un modello matematico che ci consente di calcolare, caso per caso, la dimensione ottimale di questo parametro.

Per poter implementare questa ottimizzazione, abbiamo anche sviluppato un Profilatore, che, tenendo conto della specificità dei dati genomici, definisce quantitativamente le caratteristiche che meglio descrivono un dataset ai fini dell'ottimizzazione delle query. Il Profilatore e le euristiche per il binning ottimale diventano parte dell'Ottimizzatore, un nuovo modulo del GDMS progettato per contenere e supportare le ottimizzazioni sviluppate in questa tesi e quelle che verranno sviluppate in un prossimo futuro.

Contents

Chapter 1

Introduction

Next Generation Sequencing (NGS), also known as high-throughput sequencing, is a family of technologies for reading the DNA and RNA precisely, quickly and cheaply [1], [2]; in the next decade, it will offer fast (few hours) and inexpensive (hundreds of dollars) readings of the whole human genome [3][1]. Large-scale sequencing projects are spreading, and huge amounts of sequencing data are continuously collected by a growing number of research laboratories, often organized through world-wide consortia (such as ENCODE [4], TCGA [5], 1000 Genomes Project [6], or Roadmap Epigenomics [7]). *Google* recently provided an API to store, process, explore and share DNA sequence reads, alignments, and variant calls, using Google's cloud infrastructure [2]. Answers to fundamental questions for biological and clinical research are hidden in these data, e.g., how protein-DNA interactions and DNA three-dimensional conformation affect gene activity, how cancer develops, how driving mutations occur, how much complex diseases, such as cancer, are dependent on personal genomic traits or environmental factors. Personalized and precision medicine based on genomic information is becoming a reality; the potential for data querying, analysis and sharing may be considered as the biggest and most important big data problem of mankind.

So far, the bioinformatics research community has been mostly challenged by the so-called NGS *primary analysis* (production of "reads", i.e., nucleotide sequences representing short DNA or RNA segments) and *secondary analysis* (alignment of reads to a reference genome and search for specific features on the reads, such as variants/mutations and peaks of expression); but the most important emerging problem is the NGS *tertiary*

[1] `http://www.genome.gov/sequencingcosts/`
[2] `https://cloud.google.com/genomics/`

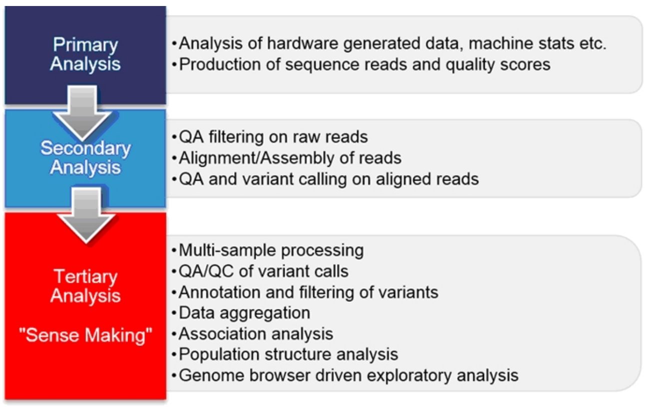

Primary Analysis
•Analysis of hardware generated data, machine stats etc.
•Production of sequence reads and quality scores
Secondary Analysis
•QA filtering on raw reads
•Alignment/Assembly of reads
•QA and variant calling on aligned reads
Tertiary Analysis
"Sense Making"
•Multi-sample processing
•QA/QC of variant calls
•Annotation and filtering of variants
•Data aggregation
•Association analysis
•Population structure analysis
•Genome browser driven exploratory analysis

Figure 1.1: Phases of genomic data analysis.

analysis, concerned with multi-sample processing of heterogeneous information, annotation and filtering of variants, and integration of genomic features (e.g., specific DNA variants, or signals and peaks of expression, i.e., DNA regions with higher density of reads). While secondary analysis targets raw data in output from NGS machines by using specialized methods, tertiary analysis targets processed data in output from secondary analysis, and it is responsible of sense making, e.g., discovering how heterogeneous regions interact with each other (Fig. 1.1).

NGS data are managed by a variety of tools focused on *ad-hoc* processing targeted to specific tasks, data extractions and transformations (e.g., alignment to a reference, mutation and peak calling, reading of gene expression); each tool manages data in specific technology-driven formats, with no emphasis on interoperability, format-independent representations, powerful abstractions, and out-of-the-box thinking and scaling. Cloud-based approaches to genomics have been targeted to speed up specific data extraction, transformation and analysis processes, but not to combine results from different data processes.

A new holistic approach for tertiary data analysis has been developed by the Genomic Computing team at Politecnico di Milano [3]. The project, lead by prof. Stefano Ceri and winner of an ERC Advanced Grant, is the continuation of the work that has been developed as part of the GenData 2020 project [4], founded at Politecnico di Milano in joint with the European Institute for Oncology and the Italian Institute for Technology

The project's main result, so far, is the proposal of a new approach to genomic data modeling and querying that takes advantage of cloud-based

[3]`http://www.bioinformatics.deib.polimi.it/geco/`
[4]`http://gendata.weebly.com/participants.html`

computing to manage heterogeneous data produced by NGS technology. What has been proposed in [18] is the Genomic Data Model (GDM), which encodes processed data in terms of their regions and metadata, and the novel 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.

One of the original aspects of the project is the targeting 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. Even if processed data are much smaller than raw data, they can still be considered as "big data", because each processed file can contain thousands or even millions of genomic regions.

Several scalable algorithms for genomic data processing have been developed during the last years within the project. The last implementation of the system, called Genomic Data Management System (GDMS), is based on Spark and Hadoop. The system can be easily deployed locally, on a user's machine, or on a cluster of machines, and provides interfaces for Scala, Python and R. Moreover, queries can be submitted through a Web Interface to run on a cluster provided by CINECA [5].

Together with the system, new abstractions for parallelism have been introduced [19]; the parallelism of massive operations on the genome is based on binning, i.e. the partitioning of the genome into equally-sized portions so that operations are performed in parallel at each bin. Therefore, instead of solving the problem chromosome-wide, each operation is computed solving several small problems in parallel, each problem involving only those regions belonging to the same bin.

Even though much efforts have been made to develop an efficient system, there are still cases in which query performance remains problematic. The main reason behind those performance issues is the lack of high level optimizations that, by reasoning on data and query characteristics, are able to put in place smart optimization strategies.

This thesis focuses on the development on one such optimization; Optimal Binning. Since experiments demonstrated that the bin size is a critical parameter for the overall performance of domain-specific operations, we developed a mathematical model that, taking into account query and data characteristics, allows to predict a bin size that makes binning efficient. The optimization is designed for JOIN and MAP, the most important domain-

[5]GMQL-V2, http://www.bioinformatics.deib.polimi.it/GMQL/interfaces/

specific GMQL operations: together with Selection and Projection, they allow defining a particular class of GMQL programs, denoted as conjunctive GMQL programs, which constitute the core of the language and are used by most applications.

In order to perform Optimal Binning, we also introduce Genomic Profiling, which, taking into account the specificity of genomic data modeling, quantitatively defines the properties that better characterize a genomic dataset from the point of view of query optimization.

In order to understand the work, it is necessary to give an introduction on the biological model and on the operations allowed by the query language; Section 2.1 presents the region-based Genomic Data Model (GDM) defined in the GenData 2020 project. Then, Section 2.2 presents the Genometric Query Language, the language used by biologists to query the data. Chapter 3 describes binning for JOIN and MAP operations. In Chapter 4 we present the analytical models for computing the optimal bin size of JOIN and MAP, together with the model validation using both synthesized and real datasets. Finally, Chapter 5 presents Genomic Profiling and the Optimizer module.

Chapter 2

Model and Language

2.1 Genomic Data Model

The Genomic Data Model (GDM) is based on the notions of datasets and samples; a dataset is a collection of samples and each sample is composed of two parts, the *metadata*, describing general properties of the sample, and the *region data*, describing portions of the DNA.

2.1.1 Motivation

Genomic processed data is typically characterized by a variety of file formats and by the lack of an attribute-based organization. As advocated by Jim Gray [8], the Genomic Data Model makes data self-describing, providing regions with a schema. However, since biologists are used to work with file-based tools, data is not included into a database; loaders are used to copy data files to a distributed file system on the cloud at their first use.

The schema is made of a fixed part, that ensures the comparability of regions coming from different kinds of processing, and of a variable part describing the features related to the specific kind of processing. Although DNA regions are strings of nucleotides[1], GDM regions are described by a list of *features*, where each feature derives from secondary data analysis.

Since there is no agreed standard for the representation of metadata, GDM represents them as attribute-value pairs; metadata should contain at least the experiment type, the sequencing and analysis method used for data production, the cell line, tissue, experimental condition (e.g., antibody target) and organism sequenced; in case of clinical studies, individual's descrip-

[1]DNA is made of strings of billions of nucleotides (represented by the letters T,C,G,A) enclosed within chromosomes (23 in humans), which are disconnected intervals of the string.

tions including phenotypes. Metadata 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 $\langle chr, left, right, strand\rangle$, called region coordinates, where chr is the chromosome, left and right are the two ends of the region along the DNA coordinates[3]; 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 $\langle id, R, M\rangle$ where:

- *id* is the sample identifier of type *long.*
- R is the set of regions of the sample, built as pairs $\langle c, f\rangle$ of coordinates c and features f; coordinates are arrays of four fixed attributes *chr, left, right, strand* which are respectively typed *string, long, long, string*; features are arrays of typed attributes; we assume attribute names of features to be different, and their types to be any of *string, int, long, double, 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.
- M is the set of metadata of the sample, built as attribute-value pairs $\langle a, v\rangle$, where we assume the type of each value v to be *string.* The same attribute name a can appear in multiple pairs of the same sample (in which case, we say that a is multi-valued).

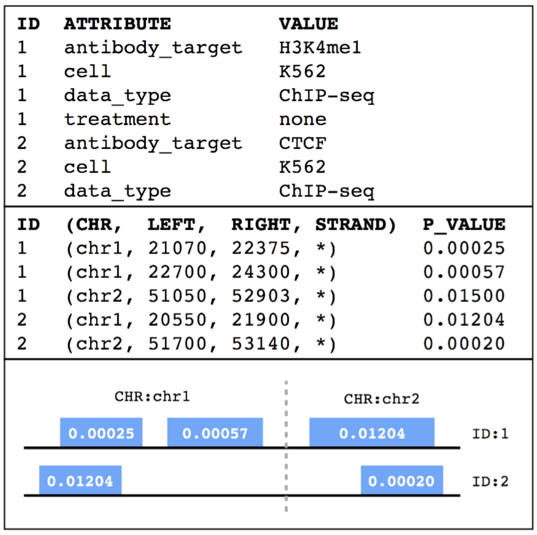
A *dataset* is a collection of samples with the same region schema and with features having the same types; sample identifiers are unique within each dataset. Each dataset is typically produced within the same project

[2]We currently store most of ENCODE [4] and TCGA [5] 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 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 $[left, right)$. Left and right ends can be identical (e.g., when the region represents a single nucleotide polymorphism)



ID	ATTRIBUTE	VALUE
1	antibody_target	H3K4me1
1	cell	K562
1	data_type	ChIP-seq
1	treatment	none
2	antibody_target	CTCF
2	cell	K562
2	data_type	ChIP-seq

ID	(CHR,	LEFT,	RIGHT,	STRAND)	P_VALUE
1	(chr1,	21070,	22375,	*)	0.00025
1	(chr1,	22700,	24300,	*)	0.00057
1	(chr2,	51050,	52903,	*)	0.01500
2	(chr1,	20550,	21900,	*)	0.01204
2	(chr2,	51700,	53140,	*)	0.00020

Figure 2.1: Regions and metadata of a dataset consisting of two samples.

(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

A dataset can be seen as a couple of 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 for modeling annotations, i.e. regions of the genome with known properties

[6]Note that the quadruple $\langle id, chr, left, right\rangle$ is not a key of the region table (because a sample can have multiple regions with the same coordinates), and similarly the pair $\langle id, attribute\rangle$ is not a key of the metadata table (because metadata attributes can be multi-valued).

```
MUTATIONS
schema   = ID, (CHR, LEFT, RIGHT, STRAND), A, G, C, T,
           del, ins, inserted, ambiguos, Max, Error,
           A2T, A2C, A2G, C2A, C2G, C2T
instance = 1, ("chr1", 917179, 917180, "*"), 0, 0, 0, 0,
           1, 0, ".", ".", 0, 0, 0, 0, 0, 0, 0, 0
```

```
RNA-seq
schema   = ID, (CHR, LEFT, RIGHT, STRAND), source, type,
           score, frame, geneID, transcriptionID,
           RPKM1, RPKM2, iIDR
instance = 1, (chr8, 101960824, 101960847, *), "GencodeV10",
           "transcript", 0.026615, NULL, "ENSG00000164924.11",
           "ENST00000418997.1", 0.209968, 0.193078, 0.058
```

Figure 2.2: Examples of schema with one instance for two different types of processed data; coordinates and features are enclosed within two records.

(such as genes, with their exons and introns). Schema encodings and one exemplar instance of mutations and RNA-seq data samples are decribed 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:

```
<variable> = OPERATION(<params>) <variables>
```

where each variable stands for a GDM dataset. Operations have associated parameters, 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 resulting 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 `COVER`, (distal) `JOIN` and `MAP`, significantly extend the expressive power of classic relational algebra.

The main design principles of GMQL are *relational completeness* and *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 *grouping* is supported, with the same semantics, in the unary operations `GROUP` and `COVER`, and similarly *joining* is supported, with the same semantics, in the binary operations `JOIN`, `MAP` and `DIFFERENCE`.

Compared with languages which are currently in use by the bio-informatic community, GMQL is *declarative* (it specifies the structure of the results, leaving its computation to each operation's implementation) and *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*[7]). Recently, thanks to the development of Scala, Python and R libraries, GMQL operators can be used following the procedural paradigm.

2.2.2 Predicates Evaluation

Several operations include predicates as their parameters. Predicates are used to select and join samples and are made of arbitrary boolean expressions, as it is customary in relational algebra. Within the predicate, region attributes can be referenced positionally with respect to the schema, i.e., `$0` denotes the first attribute `$1` to the second, and so on. Predicates may be defined either over region attributes or over metadata attributes, as follows:

- 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 $(A\ comp\ V)$ on metadata M is defined as:

  $$p \iff \exists\ (a_i, v_i) \in M : (a_i = A) \wedge (v_i\ comp\ V)$$

  If an attribute referenced in the predicate is missing, the predicate is `unknown`; we use three-value (i.e. `true`, `false`, `unknown`) logic for

[7]`http://pig.apache.org/`

metadata predicates p, and we select samples s for which $p(s)$ is `true` given the above interpretation. The special predicate *missing(A)* is `true` if the attribute A is not present in M.

- 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.[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.

2.2.3 Relational GMQL operations

We briefly describe relational operations; they include six unary operations (`SELECT`, `PROJECT`, `EXTEND`, `MERGE`, `GROUP` and `SORT`) and two binary operations (`UNION` and `DIFFERENCE`).
The standard GMQL unary operations are:

- `SELECT`: keeps in the result the input dataset samples which satisfy a metadata predicate, and then their regions which satisfy a region predicate.

- `PROJECT`: keeps in the result the input metadata and region attributes expressed as parameters. It can also be used to build new metadata attributes as scalar expressions of metadata attributes (e.g., the `age` from the `birthdate`) or as aggregate expressions of region attributes (e.g., the average `p_value` of a sample); it can also build new region attributes as scalar expressions of region attributes (e.g., the `length` of a region as the difference between its `right` and `left` ends).

- `DISTINCT`: applies to regions, sample by sample; if two or more regions of a given sample have identical coordinates, it produces in output only one region of them. New attribute values can be computed as aggregate expressions of attributes of the regions with identical coordinates (e.g., the average `p_value` of regions with identical coordinates).

- `MERGE`: applies to a single dataset and builds a single sample having as regions all the regions of the dataset samples and as metadata the union of all the attributes-values of the dataset samples.

[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.

- AGGREGATE: computes aggregate functions over region values of each sample of a dataset and adds the result as new metadata attributes of the sample.
- GROUP: applies to a single dataset and generates new metadata attributes by computing aggregate functions over the metadata of the dataset samples that share the same value for a specific metadata attribute, called grouping attribute (e.g., the average age of patients classified as cases or controls).
- ORDER: uses metadata attributes to order the samples of a dataset, by adding to each sample an order metadata attribute with the sample ranking value, and possibly to filter the top samples based upon the ordering.

Two GMQL binary operations allow building unions or differences of datasets and samples.

- UNION: applies to two datasets and builds their union, so that each sample of each operand contributes exactly to one sample of the result; if datasets have different schemas, these are merged, and missing values are set to null.
- DIFFERENCE: applies to two datasets and preserves the regions of the first dataset which do not intersect with any region of the second dataset; only the metadata of the first dataset are maintained, unchanged.

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.

Cover

```
<S2> = COVER/FLAT/SUMMIT/HISTOGRAM (<minAcc>, <maxAcc>
          [; groupby: <Am1>, .., <Amn>]
          [; aggregate: <Ar1> AS <g1>, .., <Arn> AS <gn>]) <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

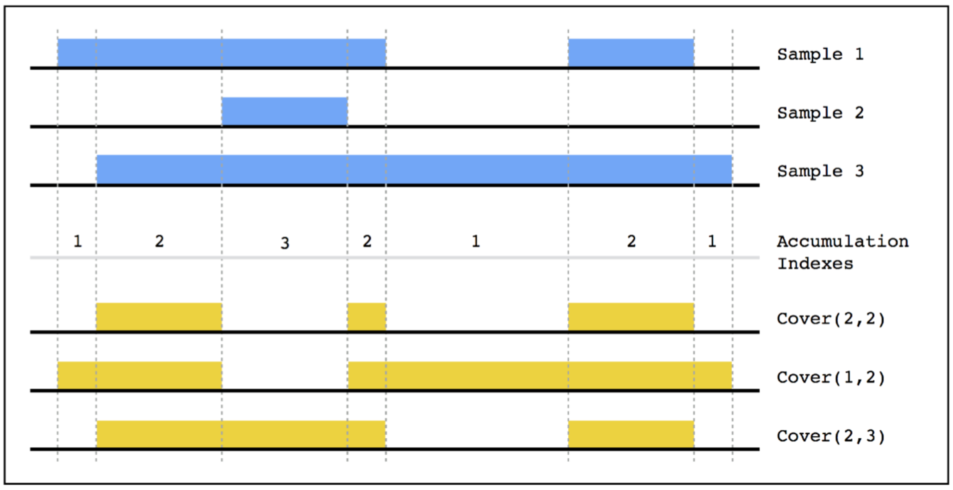


Figure 2.3: Accumulation index and `COVER` *results with three different* `minAcc` *and* `maxAcc` *values.*

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 in $S1$ are assigned to the resulting single sample s in $S2$. Regions of the result sample are built from the regions of samples in $S1$ according to the following condition:

- Each resulting region r in $S2$ is the contiguous intersection of at least `minAcc` and at most `maxAcc` contributing regions r_i in the samples of $S1$ [9]; `minAcc` and `maxAcc` are called **accumulation indexes**[10].

Resulting regions may have new attributes Ar, calculated by means of aggregate expressions over the attributes of the contributing regions. `Jaccard Indexes`[11] are standard measures of similarity of the contributing regions r_i, added as default region attributes. When a `GROUPBY` clause is present,

[9]When regions are stranded, cover is separately applied to positive and negative strands; in such case, unstranded regions are accounted both as positive and negative.

[10]The keyword `ANY` can be used as `maxAcc`, and in this case no maximum is set (it is equivalent to omitting the `maxAcc` option); the keyword `ALL` stands for the number of samples in the operand, and can be used both for `minAcc` and `maxAcc`. Cases when `maxAcc` is greater than `ALL` are relevant when the input samples include overlapping regions.

[11]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.

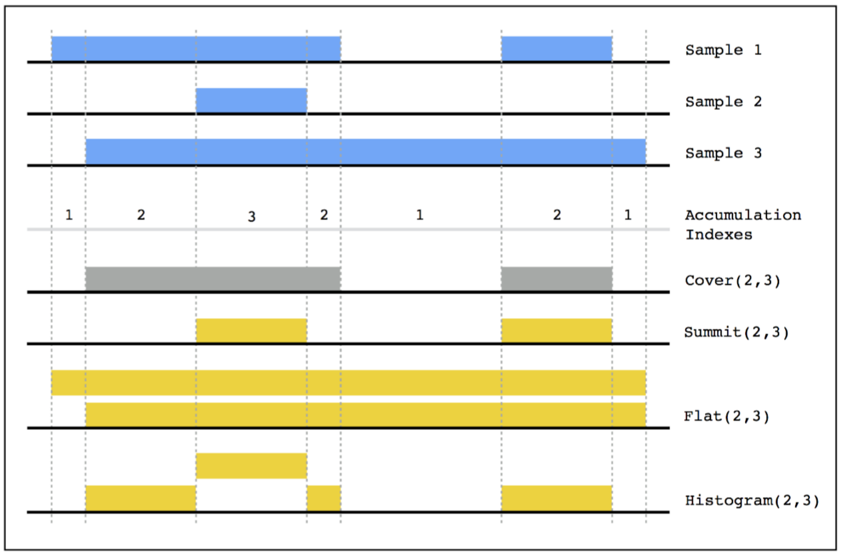


Figure 2.4: Accumulation index and `COVER` *variants results.*

the samples are partitioned by groups, each with distinct values of grouping metadata attributes (i.e., homonym attributes in the operand schemas) and the cover operation is separately applied to each group, yielding to one sample in the result for each group.
For what concerns variants:

- `FLAT` 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`).

- `SUMMIT` returns only those portions of the result regions of the `COVER` where the maximum number of regions intersect (more 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).

- `HISTOGRAM` returns the nonoverlapping regions contributing to the cover, each with its accumulation index value, which is assigned to the **AccIndex** region attribute.

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 the accumulation index and then the regions resulting from setting the `minAcc` and `maxAcc` parameters respectively to $(2, 2)$, $(1, 2)$, and $(2, 3)$. Fig. 2.4 shows an example with variants.

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 `pValue` of the overlapping regions and their Jaccard indexes; the result has one sample for each input `cell`.

```
RES = COVER(2, 3; groupby: cell; aggregate:
                                    pValue AS MIN(pValue)) EXP;
```

Map

```
<S3> = MAP([<Ar1> AS <g1>, .., <Arn> AS <gn>]
       [;][joinby: <Am1>, .., <Amn>]) <S1> <S2>;
```

`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 each matrix row is a vector of numbers[12]. 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 $s_2 =< id_2, R_2, M_2 >$ (according to the GDM notation), the new sample $s_3 =< id_3, R_3, M_3 >$ is constructed; id_3 is generated from id_1 and id_2[13], the metadata M_3 are obtained by merging metadata M_1 and M_2, and the regions $R_3 = \{< c_3, f_3 >\}$ 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

[12]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.

[13]The implementation generates identifiers for the result by applying hash functions to the identifiers of operands, so that resulting identifiers are unique; they are identical if generated multiple times for the same input samples.

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. For each region, a field named `count_LeftDSName_RighDSName` is added, storing the result of `Count` aggregate. 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 $s1$ of the first dataset $S1$ we consider the regions of the samples $s2$ of $S2$ that satisfy the join condition. Syntactically, the clause consists of a list of attribute names, which are homonyms from the schemas of S1 and of S2; 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.

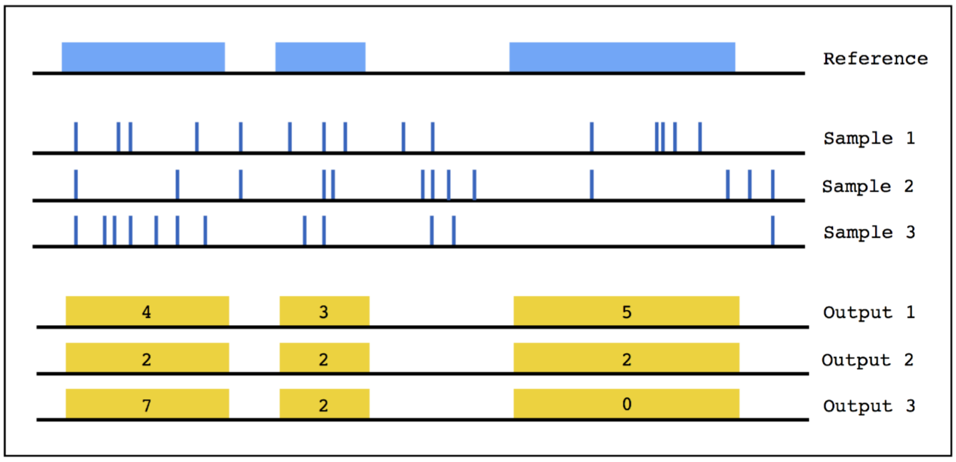


Figure 2.5: Example of map using one sample as reference and three samples as experiment, using the Count aggregate function.

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 with 3 regions and three mutation experiment samples, the output consists of three samples, each 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 (3×3) genome space.

In the example below, the `MAP` operation counts how many mutations occur in known genes, where the dataset `EXP` contains DNA mutation regions and `GENES` contains the genes.

```
RES = MAP() GENES EXP;
```

Join

```
<S3> = JOIN([<genometric-pred>][;] [output: <coord-gen>]
        [;] [joinby: <Am1>, .., <Amn>]) <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 predicate** 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 `S1` and `S2`, 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 $S1$ and s_2 of $S2$ that satisfy the joinby condition. If the clause is omitted, then the Cartesian product of all pairs s_1 of $S1$ and s_2 of $S2$ 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 $< r_i, r_j >$ of regions, with $r_1 \in s_1$ and $r_j \in s_2$, by assigning the role of **anchor region**, in turn, to all the regions of $s1$, and then evaluating the genometric predicate condition with all the regions of $s2$. From every pair $< r_i, r_j >$ that satisfies the join condition, a new region is generated in s_{12}.

From this description, it follows that the join operation yields 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.[14] A genometric predicate is a sequence of distal conditions, defined as follows:

[14]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).

- `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[15]. When this clause is not present, distal conditions apply to both the directions of the genome.

- `MD(K)`[16] 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[17].

- `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 `Max` is a problem-specific maximum distance.

Example. The following strings are legal genometric predicates:

```
DGE(500), UP, DLE(1000), MD(1)
DGE(50000), UP, DLE(100000)
DLE(2000), MD(1), DOWN
MD(100), DLE(3000)
```

[15] *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. (Remaining regions of the experiment are overlapping with the anchor region). For the *negative strand*, ends and disequations are exchanged.

[16] Also: MINDIST, MINDISTANCE.

[17] 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.

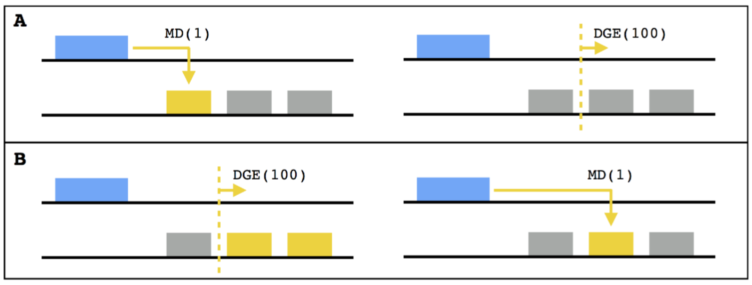


Figure 2.6: Different semantics of genometric clauses due to the ordering of distal conditions; excluded regions are gray.

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[18].

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. The coordinates c_{ij} are generated according to the `coord-gen` clause, which has four options [19]:

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.

[18]The 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.*

[19]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.

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[20]. Note that the result uses the coordinates of the experiment.

```
RES = JOIN(MD(1), DGE(120000); output: RIGHT) TSS HM;
```

2.3 Utility Operations

2.3.1 Materialize

```
MATERIALIZE <S1> INTO file_name;
```

The `MATERIALIZE` operation saves the content of a dataset `S1` in a file, whose name is specified, and registers the saved dataset in the system to make it seamlessly usable in other GMQL queries. All datasets defined in a GMQL query are, by default, temporary; to see and preserve the content of any dataset generated during a GMQL query, the dataset must be materialized. Any dataset can be materialized, however the operation is time expensive; for best performance, only relevant data should be materialized.

[20]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

Binning Algorithms

In this chapter we describe the functioning of binning algorithms for `JOIN` and `MAP` operations. The early implementations of the system used to translate GMQL into PIG[1]. Simple user-defined Java functions, that took advantage of the ordering of regions along the reference genome, allowed to produce the results, but with limited parallelism. In order to make an implementation that was more suitable for a big data problem, methods developed for temporal databases were adapted and new binning algorithms were introduced. Binning Algorithms make the implementations of domain specific operators highly parallel by partitioning the genome into much smaller, identical partitions called **bins**. Then, each operation, instead of being computed along the whole chromosome, is computed within each bin. High parallelism is achieved since every bin can be processed in parallel independently from the others. Something similar to our binning is used by databases of annotations of the UCSC Genome Browser [10] in order to speed up the search for portions of the genome that must be loaded within the same browser window. To the best of our knowledge, binning has not been used for parallelizing genomic operations; the use of time portions of equal size for temporal interval joins is studied in [11] in the context of map-reduce. We next focus on the implementation of domain-specific operations.

3.1 Join

Before going into the details of Join implementation, we discuss clause evaluation order, Join binning strategy, and the relation between binning and the query-specific search space.

[1]Apache Pig https://pig.apache.org/

3.1.1 Evaluation Steps

As discussed in Section 2.2.4, the result of a Join operation is influenced by the order of execution of the distal conditions specified in the query; the reason is that the min distance clause (`MD`) clause is not commutative with the greater distance clause (`GTE`) and with the stream clause (`UP/DOWN`); instead less distance clause `DLE` is commutative with all other clauses, and stream and greater distal clauses are commutative with each other. Thus, the evaluation of a genometric predicate is performed in 3 steps, where clauses within each step are commutative and each step can be missing:

- *Step 1* evaluates the `DLE` clause, the `DGE` and the stream clauses preceding the `MD` clause; if a query-specific `DLE` clause is not present, then `DLE(Max)` is added, where `Max` denotes the maximum biological distance.
- *Step 2* evaluates the `MD` clause.
- *Step 3* evaluates the stream and greater distal clauses after the `MD` clause.

Examples. The genometric predicate:

```
DGE(300), MD(3), DOWN
```

produces the following three steps:

```
Step 1: DGE(300), DLE(Max)
Step 2: MD(3)
Step 3: DOWN
```

The genometric predicate: `DOWN, MD(10), DGE(1000), DLE(2500)` produces the following three steps:

```
Step 1: DOWN, DLE(2500)
Step 2: MD(10)
Step 3: DGE(1000)
```

Simpler predicates may be evaluated into a single step, e.g. `DGE(1000), UP, DLE(5000)`.

3.1.2 Binning and Search Space

Efficient executon, in cloud computing, is achieved by means of parallelism; in genomic computing, such parallelism is achieved by means of **binning**,

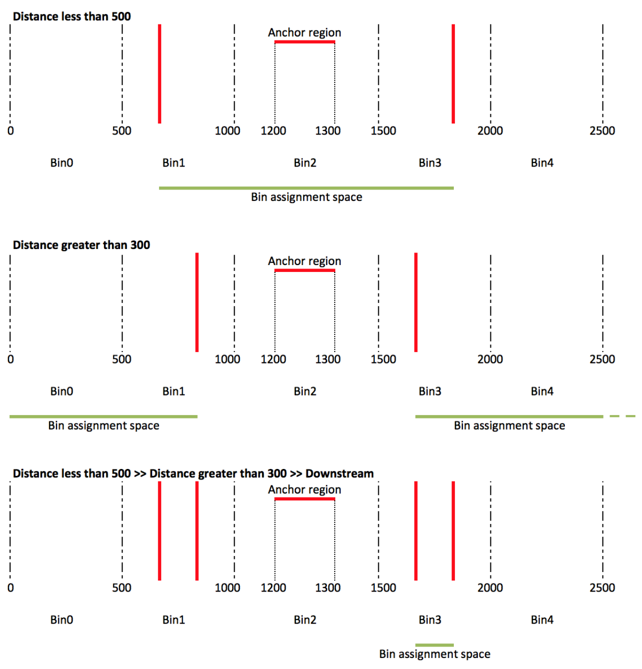


Figure 3.1: Search spaces for three distal clauses, Step 1.

i.e., splitting the genome into disjoint sections (bins) so that large computations can be distributed among them. The process of binning splits every chromosome of the genome into several bins of equal size S; for each chromosome, bins are progressively numbered starting from 0 up to the *i-th* bin, that spans from $S \times i$ to $S \times (i + 1) - 1$. For a given bin size S, a point placed at i bases from the chromosome start is assigned to bin $b(i) = \lfloor i/S \rfloor$. Intervals between a left end l_i and a right end r_i are assigned to the bins between $b(l_i)$ and $b(r_i)$.

In order to effectively evaluate distal clauses, each anchor region is associated with its **search space**, consisting of intervals of bins that may include matching regions of the experiment; search spaces are built according to the distal conditions of Step 1; it includes all potential matches, as Steps 2 and Step 3 are filters of the regions produced by Step 1. Consider an anchor region with left end l and right end r; let M be the maximum distance and let B_c denote the last bin of each chromosome c [2]. Then:

- If the clause is $LTE(d)$, then the search space is the interval of bins between $b(l - d)$ (excluding bins with $i < 0$) and $b(r + d)$ (excluding bins with $i > B_c$).

[2]Given that chromosomes have different sizes, B_c is a specific number for each chromosome.

- If the clause is $LTE(d_1)$ and $GTE(d_2)$, with $d_1 > d_2$, then the search space is the two intervals of bins between $b(l - d1)$ and $b(l - d2)$ (excluding bins with $i < 0$) and between $b(r+d_2)$ and $b(r+d_1)$ (excluding bins with $i > B_c$).

- If the clause is $GTE(d_1)$, then the search space is the two intervals of bins between $b(l - M)$ and $b(l - d_1)$ (excluding bins with $i < 0$) and between $b(r + d_1)$ and $b(r + M)$ (excluding bins with $i > B_c$).

When the `UP/DOWN` clause is present, the search space is limited to the upstream/downstream directions of the genome. A representation of the search space for the anchor region as effect of the `DLE` and `DGE` clauses is shown in Fig. 3.1 (cases 1 and 2); the third case shows the effects of combining the `DLE`, `DGE` and `DOWN` clauses.

3.1.3 Evaluation of Distal Clauses in Step 1

This construction allows a parallel evaluation of join predicates. In particular, the following theorem holds due to the way in which search spaces are constructed:

Theorem 1. *The join predicate between an anchor region and any experiment region falling outside of its search space is false.*

In addition, we would like to evaluate the Step 1 join predicate between given regions of the anchor and experiment in a given bin only, so as to generate the corresponding result region only once, avoiding duplicates. The following theorem provides a solution of this problem.

Theorem 2. *If the Step 1 predicate between an anchor region and an experiment region is true, it can be tested in a given bin, denoted as* **testing bin**.

Proof. We build the proof by considering four cases which exhaustively cover the relationships between anchor and experiment regions, and defining the testing bin for each of them.

- Assume that *the experiment is at the left of the anchor*, i.e. the experiment's left end is strictly less than the anchor's left end and the experiment's right end is less than or equal to the anchor's right end. Then, the testing bin is the experiment bin with greatest number (the one at the smallest distance from the anchor); the predicate can be true only if the portion of experiment region within the testing bin

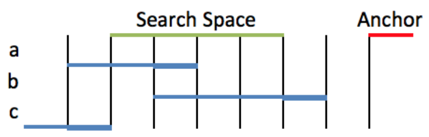


Figure 3.2: Experiment regions at the left of the search space.

intersects with the search space. Some examples are shown in Fig. 3.2, where the testing bin is denoted by a thicker trait. The predicate can be true in case (a) (when the testing bin falls within the search space) and is false in case (b) (as the region is too close to the anchor) and (c) (as the region is too distant from the anchor).

- The case when *the experiment is at the right of the anchor* is symmetric; in such case, the experiment's right end is strictly greater than the anchor's right end and the experiment's left end is greater than or equal to the anchor's left end. Then, the testing bin is the experiment bin with the smallest number; also in such case, the predicate can be true only if the portion of experiment region within the testing bin intersects with the search space.

- Assume that *the experiment is included within the anchor*. Recall that by construction the search space either properly includes the anchor region or does not overlap with it. Thus, the experiment can satisfy the join predicate only if it intersects with the search space in anyone of its bins; conventionally, we may use as testing bin the experiment bin with the smallest number. This case is illustrated in Fig. 3.3.

- Finally, assume that *the anchor is included in the experiment*. Then, the anchor is at negative distance from the experiment, and again the search space either properly includes the anchor region or does not overlap with it; it follows that the join predicate between the region and the anchor can be true only if the search space includes the anchor. Conventionally, we may use as testing bin the anchor bin with the smallest number. This case is illustrated in Fig. 3.4.

□

Thanks to Theorem 2, at each bin B we evaluate Step 1 conditions just for those pairs of experiment and anchor regions such that B is their testing bin; thus, we either discard the pair of regions, or produce the resulting regions exactly once. This result is used by the parallel execution strategy which is next discussed.

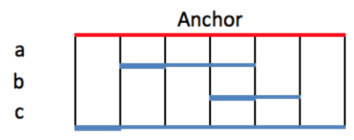


Figure 3.3: Experiment regions enclosed within the anchor region.

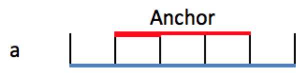


Figure 3.4: Anchor region enclosed within the experiment region.

3.1.4 Join Execution Strategy in Spark

Fig. 3.5 illustrates the flow of Spark operators for implementing a join operation. We recall that joins require first to select the pairs of samples that need to be joined, using a metadata predicate, and then to compute the result regions, using a genometric predicate. The operation applies to two datasets, respectively called *anchor* and *experiment*; as a running example we consider the join with:

```
Step 1: DGE(140), DLE(500)
Step 2: MD(1)
Step 3: DOWN
```

computed on:

```
Anchor: Id, chromosome, start, stop
1 C1 150 160
1 C1 285 390
Experiment: Id, chromosome, start, stop
2 C1 10  20
2 C1 430 550
2 C1 750 780
```

Throughout the examples of this section, we do not consider strands; in reality, join predicates evaluation is defined only between regions with compatible strand[3]. We also do not consider region values, they are carried along with each region and concatenated in the result[4].

[3]Positive and negative strands are not compatible, and they are both compatible with undefined strands.

[4]With big value sizes, it is convenient to project the values prior to Block 1 and then join them to resulting regions within Block 5.

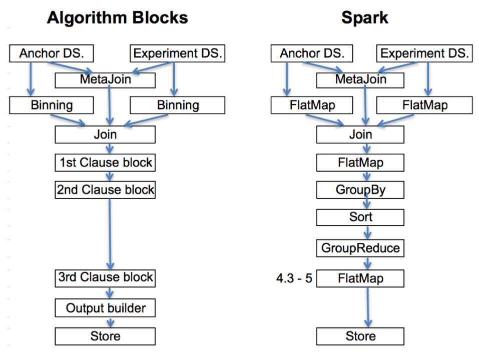


Figure 3.5: Operators for encoding the Join algorithm in Spark.

- Block 1 (Metajoin) produces in output, for each anchor sample, the *join list* of the experiment samples that must be joined to it.

  Example. The join list of sample 1 is [2].

- Block 2 (FlatMap) is responsible of copying regions to the bins:

  - For every anchor region and bin b intersecting with the search space, it generates a copy of the anchor region and assigns it to bin b, by adding the attribute `Bin` (b) and the attribute `SBin` (the bin in which the anchor region starts.)
  - For every sample of the join list and for every bin b intersecting with each experiment region, it generates a copy of the experiment region, adding to it the attribute `Bin` (b) and the attributes `SBin` (the bin where the experiment region starts) and `EBin` (the bin where the experiment region ends.)

  Note that anchor regions are replicated at the bins intersecting their search space, computed in this block, and experiment regions are replicated at the bins which intersect with them. The added attributes allow to test with a simple predicate if the current bin b is the testing bin of a given pair of anchor and experiment regions, based on the four cases of Theorem 2.

  Example. With a bin size $B = 100$, the first anchor region is copied to the bins $0, 2-6$, the second anchor region is copied to the bins $0-8$; the experiment regions is copied to the bins 0, $4-5$, and 7.

- Block 3 (Join) joins the anchor and experiments by `chrom` and `bin`. In this way, for any pair of anchor and experiment samples to be joined and for any of their anchor and experiment regions, all the relevant data are available at all bins, hence also at their testing bin. This operation is the most expensive, as it may join millions of regions to millions of regions; it is effectively computed by the `Join` operator, available in Spark.

Example. The following pairs are produced:

Bin	Chr	Id1	SB1	L1	R1	Id2	L2	R2
B0	C1	1	B1	150	160	2	10	20
B4	C1	1	B1	150	160	2	430	550
B5	C1	1	B1	150	160	2	430	550
B7	C1	1	B1	150	160	2	750	780
B0	C1	1	B3	285	390	2	10	20
B4	C1	1	B3	285	390	2	430	550
B5	C1	1	B3	285	390	2	430	550
B7	C1	1	B3	285	390	2	750	780

- Block 4.1 (FlatMap in Spark) performs Step 1, by computing the distance between the regions in each row and then selecting only the rows of the testing bins where the distal conditions hold; testing bins are determined as indicated in the four cases of the proof of Theorem 2. This step is computed in parallel in each bin; in Spark is a `FlatMap`.

  Example. The following pairs are produced:

Bin	Chr	Id1	SB1	L1	R1	Id2	SB2	EB2	L2	R2	D
B4	C1	1	B1	150	160	2	B4	B5	430	550	270
B0	C1	1	B3	285	390	2	B0	B0	10	20	265
B7	C1	1	Bin3	285	390	2	B7	B7	750	780	360

- Block 4.2 (GroupBy, Sort, GroupReduce) performs Step 2, by selecting experiment regions based upon their minimal distance from anchor regions; it is implemented by the `GroupBy`, `Sort` and `GroupReduce` operators, but it requires data shuffling for collecting the experiment regions at nodes where sorting by distance and $top-k$ selection can be performed. The latest implementation reduces data shuffling with an alternative implementation, which adds a sort operation at each bin, producing at each bin the $top - k$ regions; these need to be moved, while all other regions can be discarded.

  Example. The following pairs are produced:

Bin	Chr	Id1	SB1	L1	R1	Id2	SB2	EB2	L2	R2	D
B4	C1	1	B1	150	160	2	B4	B5	430	550	270
B0	C1	1	B3	285	390	2	B0	B0	10	20	265

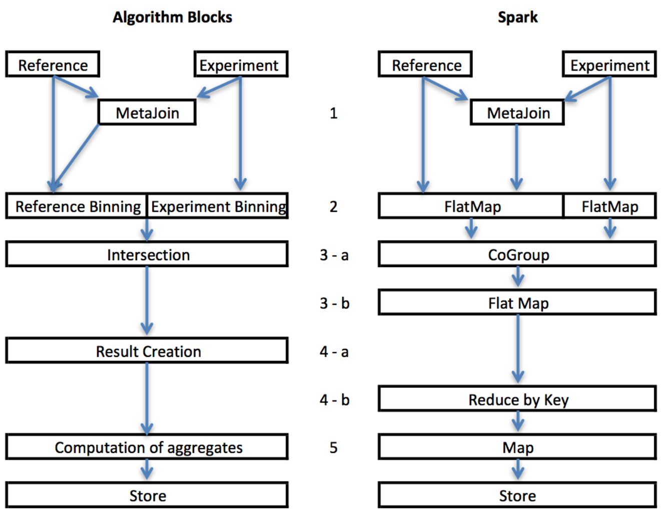


Figure 3.6: Operators for encoding the Map algorithm in Spark.

- Block 4.3 (FlatMap) performs Step 3, by further reducing the filtered regions according to the distal conditions of Step 3. It uses the `FlatMap` operator.

  Example. In the example, the condition `DOWN` filters one pair, producing:

  ```
  Bin Chr  Id1 SB1 L1 R1   Id2 SB2 EB2 L2 R2 D
  B4  C1   1 B1 150 160    2 B4 B5 430 550 270
  ```

- Block 5 (FlatMap) is responsible of outputing the resulting pairs, by computing their sample identifier and their region coordinates according to the coordinate composition option and is executed together with block 4.3

  Example. We finally obtain the following result, where a new sample identifier is generated as a hash function of the identifiers of the two operands, and the resulting region is obtained by concatenating the operand regions:

  ```
  Id         Chr  Start Stop
  Hash(1,2) C1  150   550
  ```

3.2 Map

The encoding of this problem as a sequence of operations for Spark is shown in Fig. 3.6. The algorithm requires to bin the two datasets, then group them by sample pair, chromosome and binning. The cogrouped datasets are used to compute intersections within the bins. The resulting intersecting regions then are used as input for the aggregate functions.

The following example is to describe Map algorithm operation shown in 3.6, here we count the experiment regions intersecting with reference regions; The input is:

```
Anchor: Id, chromosome, start, stop
1 C1 150 235
Experiment: Id, chromosome, start, stop
2 C1 10  230
```

- Block 1 (Metajoin) produces in output, for each reference sample, the *map list* of the experiment samples that must be mapped to it based on the metadata condition for joining samples.
- Block 2 (Experiment Binning) is responsible of copying experiment regions to the bins. For every experiment region and bin *b* intersecting with the experiment, it generates a copy of the region for every bin *b*; only the attributes which are used by aggregate functions are copied.

  Example. With bins of size 100, the following copies are generated:

  ```
  Id Chr Bin Start Stop
  2  C1  0 10 230
  2  C1  1 10 230
  2  C1  2 10 230
  ```

  Note that a list of attribute values are generated, but no attribute value is needed for computing the `COUNT`.
- Block 2 (Reference Binning) is responsible of copying reference regions to the bins. For every reference region of a given sample, for every bin *b* intersecting with the reference, and for every experiment sample in its map list, a copy of the reference region is built, having as attributes the concatenation of `Id, Chr, Bin, Start, Stop` of the reference with the `Eid` of the experiment and with a new attribute `H` obtained by hashing all the attributes except the bin; this attribute is later used for assembling all copies relative to the same reference and experiment regions.

  Example. The following copies are generated:

  ```
  Id Chr Bin Start Stop Eid H
  1  C1  1 150 235 2 567
  1  C1  2 150 235 2 567
  ```

- Block 3-a (CoGroup) is responsible of computing a partial map within each bin. It joins references and experiment by `Eid`, `Chr` and `Bin`; if the join succeeds, it further selects resulting tuples by considering only the bins where either the reference region or the experiment region starts (note that this bin exists and is unique by construction). At each selected pair, a portion of the aggregate function is computed. A new region is built, having as attributes the concatenation of `Chr, Bin` with `Rid, Start, Stop, H` of the reference and `EId, EStart, EStop, V` of the experiment; `V` stores the experiment values to be used by the aggregate functions (in the case of `Count`, it stores `1`.) If the join fails, thanks to the CoGroup constructor, all the reference information is stored to the result, with null values stored for the experiment; in this way, all reference regions are correctly accounted.

  Example. The following copies are generated, and the second one is then filtered:

  ```
  Chr Bin RId Start Stop H EId EB EStart Estop V
  C1  1   1 150 235 567    2 c1 1 10 230 [1]
  C1  2   1 150 235 567    2 c1 2 10 230 [1]
  ```

- Block 4 (Assembling) is responsible of assembling all copies corresponding to the same reference and experiment at one node, through data shuffling; the operation is performed thanks to a `reduce phase` which uses the `Hash` attribute. Partial sums are performed for computing `COUNT`, and lists of attribute values are concatenated within a bag.

  Example. In the example, the two regions are reduced to one, as they have the same hash attribute. The following region is generated:

  ```
  Rid Chr Start Stop Val
  567 C1 150 235 1
  ```

- Block 5 (Aggregating) is responsible of computing aggregate functions, by applying them to the bag of values built at block 4. This step does not apply to the running example.

Chapter 4

Optimal Binning

In the previous chapter we discussed how binning algorithms work conceptually and how they have been implemented in Spark. We also anticipated that the choice of the bin size can become critical for the performance of GMQL operations.

This chapter starts pointing out the importance of the choice of a good bin size and discussing the current solution adopted by GMQL. Then, the most important factors affecting the choice of a good bin size are discussed and *optimal binning*, the optimization at the core of this thesis, is presented. In contrast with unoptimized binning, where the bin size was chosen statically for each operator, optimal binning enables the choice of the bin size at runtime, based on the characteristics of the input data and query. In sections 4.4 and 4.5 we present simple heuristics that allow to determine an optimal bin size for JOIN and MAP operators and we evaluate the goodness of those heuristics through some experiments on both synthesized and real data.

4.1 Motivation

As discussed in Chapter 3, the rationale of binning is to reduce the number of regions to be considered within each bin, instead of computing chromosome-wide cross products; however, regions that cross bin borders must be replicated. Large bins reduce replication of regions, but they lead to the production and matching of many pairs of regions within each bin; conversely, small bins increase replication and therefore the generation of matching regions that should not be produced in output. In Fig.4.1., we show the execution time of some Join operations as a function of the bin size. In these experiments we can see how the bin size is affecting the execution time. The

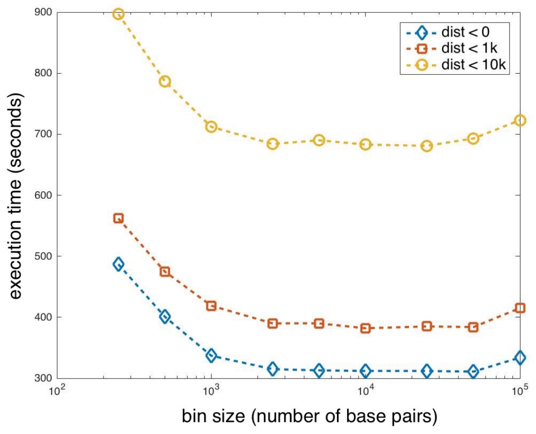


Figure 4.1: Execution time of Join as a function of bin size in logarithmic scale (Spark).

execution time functions clearly show the existence of some bin sizes that are better than the others; too small and too big bin sizes make the execution time increase significantly, coherently with what said before. In these cases, bin sizes between $0.5 \cdot 10K$ and $5 \cdot 10K$ can be considered optimal.

The existence of an optimality range for the bin size was already clear when binning algorithms were designed, as discussed in [19]. It was also clear the dependence of the optimal bin size on the characteristics of the input data and query. The solution that was adopted at that time, was choosing the bin size statically (once for all) for each domain-specific operator of the language.

Table 4.1: Static bin sizes for Jon and Map operators.

Operation	Bin Size [bases]
Join	$5K$
Map	$7K$

The static bin sizes, listed in Table 4.1, represent a good compromise between several experiments made on the datasets available in the GMQL Repository in 2015. This means that the choice of those bin sizes fits, somehow, the characteristics of a specific collection of datasets. However, as the system evolves and more users get involved, we expect our repository to be more heterogeneous in terms of the characteristics of the datasets it contains. Indeed, users can upload their own datasets and generate new datasets as result of a query execution.

Since we do not make any assumption on the characteristics of the datasets in our repository, we must ensure that our system performs well independently of those characteristics. With unoptimized binning, the risk of long execution times or non-terminating executions, due to the saturation of cluster resources, is very high and gets higher as the datasets and query characteristics become different from those we have always been used to. Therefore, the implementation of optimal binning is a key aspect for the efficiency and scalability of the system.

4.2 Features for Optimal Binning

The way we find an optimal bin size is based on the minimization of a mathematical function that models the execution time of domain-specific operators as a function of the bin size; e.g., considering one of the executions in Fig.4.1, we find the optimal bin size minimizing a function that suitably fits the execution time curve.

Before going into execution time modeling, we tried to understand which factors affected the behavior of the execution time function. Input data size is typically one of the most important factors to be taken into account when one tries to estimate the execution time of a generic application. However, binning algorithms strictly depend on a specific data model and query language, and data size is not enough to model the execution time of operations. Indeed, the execution time of the operations performed at each bin depends on the amount of region replicates generated for that bin, that, on its turn depends on some features, e.g. region length, not related to the input data size. In addition to data and query characteristics, the execution time is also dependent on the computing infrastructure running the system. A list of the main factors influencing the execution time, and consequently the optimal bin size, is here presented:

- *Number of regions* contained in each sample. Intuitively, when regions increase in number, smaller bins are needed in order to avoid lots of comparisons within each bin.

- *Number of samples* contained in each input dataset. Having more samples implies more regions in each bin; again, smaller bins are needed in order to avoid lots of comparisons.

- *Regions length*: when regions are longer than bins, they are going to produce many replicates, increasing the cost of data processing and the number of useless tests; bigger bins are needed.

Table 4.2: Symbols used to describe data and query features.

Symbol	Meaning
S	Number of samples in a dataset
N	Number of regions
w	Average region length
m	Minimum left coordinate
M	Maximum right coordinate
LT	Less than clause argument
GT	Greater than clause argument
str	1 if stream clause defined , 0 if missing

- The *distribution of regions* along the chromosome affects the load of each bin and with it the overall number of regions to be compared.
- *Physical characteristics of the cluster*: the amount of memory and the computational power of the cluster affect the shape of the execution time function and, consequently, the range of optimality.
- *Query*: operator arguments affect the optimality range; e.g. increasing the search space, defined by the genometric predicate of the JOIN operation, more replicates are produced for the same anchor region; larger bins are needed to reduce the number of replicates.

The extraction of data and query properties is discussed in Chapter 5, where we describe the Optimizer and Genomic Profiling. Table 4.2 lists the properties required by the heuristics defined in the following sections. Note that N, w, m and M are computed for each sample of a dataset.

In order to have faster profiling and to keep our heuristics adequately simple, we took the following decisions:

- We opted for a single parameter per sample instead of chromosome-specific parameters, e.g. we compute the average region length for each sample without discriminating between chromosomes.
- We did not consider the distribution of regions. We assume that, in every sample, regions are uniformly distributed along a single chromosome that spans from m to M.
- Physical characteristics of the cluster are not explicitly taken into account. As discussed in the next session, we retune some parameters whenever the cluster configuration changes.

4.3 Modeling the execution time

In this section we describe the methodology that has been used to build the heuristics for optimal binning of JOIN and MAP.

The methodology is based on the modeling of the execution time as a function of the bin size. From now on, we will refer to this function as the *cost function*. What we expect, is to find a curve similar to those we showed in Fig. 4.1. Once we come up with a good approximation of the cost function, all we need to do is to take the bin size that minimizes the curve and consider it as the optimal bin size.

We model the cost function of an operator as a linear combination of the dominant time complexities associated to the operations involved in the algorithm implementing the operator. If an algorithm has time complexity

$$f(n) = n^2 + n + log(n)$$

we may discard the logarithm and consider only two (dominant) time complexities, the quadratic and the linear, and then model the execution time as a linear combination of the two.

Usually, time complexity is expressed as a function of the variable input size n; in our case we are interested in modeling the execution time as a function of the bin size b.

Formally, the cost function associated to an operator is expressed as:

$$c(b) = \sum_i a_i \cdot \tau_i(b) + k \tag{4.1}$$

where b is the bin size, $c(b)$ the cost function, $\tau_i(b)$ is the i-th dominant time complexity of the algorithm implementing the operator, a_i a scale factor for the i-th time complexity and k is a quantity that does not depend on b. Note that it must be $b > 0$, since negative or null bin sizes make no sense. If this function has a local minimum it is possible to find the optimal bin size (b^*) as:

$$b^* = argmin_b \quad c(b)$$

i.e. the bin size that minimizes the cost function.

Scale factors can be thought as the amount of time required to process each input entry (e.g. a region), so they depend on the availability of computational resources. For this reason, we retune scale factors whenever we change the cluster configuration. Moreover, in the following sections we will discard any term of the cost function expression that does not depend on *b*. Discarding constants does not change the position of the minimum and preserves the "shape" of the curve. In other words, our cost function does not estimate the real execution time, but we are still able to say which bin corresponds to the minimal execution time and know the gain in time we get by using a bin size instead of another.

4.4 Join

The Join operation produces results that can grow quadratically both in the number of input samples and in the number of regions, and is the most critical GMQL operation from a computational point of view. In this section, we present and motivate the heuristics used to predict the optimal bin size for this operation. Before building the final cost function, using the methodology described in the previous section, it is necessary to understand how data are replicated by binning algorithms. For this purpose, in section 4.4.1 we build some heuristics that allow to estimate the amount of regions available at each bin after binning. Then, in section 4.4.2, we analyze the complexity of the Join implementation and we use replication estimates to define the cost function and the optimal bin size heuristics. Finally, we validate the heuristics through some tests performed on both synthesized and real data. We assume that reader already knows how the Join operation works (sec. 2.2.4) and how the binning of its input dataset is performed (sec. 3.1).

4.4.1 Estimating replication

As explained in sec. 3.1, binning algorithms replicate regions of a datasets in each bin overlapping with the region, when binning is performed on the experiment dataset, or overlapping with the search space[1], when binning is performed on the anchor dataset. Estimating the number of generated replicates [2] with a given bin size is fundamental to know how many computations, e.g. comparisons, are performed at each bin. In the following, we build the heuristics that allow us to compute the number of replicates generated by the binning of Join input datasets.

[1]Differently from the previous chapters and from its original definition, in this chapter we don't refer to "search space" as an interval of bins but as an interval of coordinates; e.g. the search space associated to a reference region of coordinates $[100, 200]$, given a genometric predicate `DGE(100), DLE(200), DOWN`, is the interval $[300, 400]$

[2]We use the term *replicates* to refer to all the region instances, each one associated to one bin, produced as a consequence of binning.

Binning the Experiment Dataset

The heuristic defined in this section is the simplest and most recurrent heuristic in this work; it will be the core of upper level heuristics and will be extended later to define more complex estimates.

As already said in Chapter 3, a region of the experiment dataset is replicated in those bins intersecting with the region.

Theorem 3. *The average number of replicates generated by the binning of a single region of the experiment dataset is given by:*

$$r_{single}(b) = \frac{l-1}{b} + 1 \tag{4.2}$$

where l is the region length and b is the bin size. The average is intended over all the possible positionings of the region within the chromosome, assuming that every position is equally probable.

Proof. To prove this theorem we start defining the concept of *minimum frame.*

Definition 1. *The Minimum Frame μ for a generic region of length l is the mimum number of bins, of fixed size b, that can contain that region.*

The minimum frame can be computed as the least succeeding integer (ceiling) of the fraction between the region length and the bin size:

$$\mu = \left\lceil \frac{l}{b} \right\rceil$$

Example. The minimum number of bins of size 2 that can contain a region of length 3 can be computed as

$$\mu = \left\lceil \frac{3}{2} \right\rceil = 2$$

In most cases, the region will simply overlap with a number of bins corresponding to its minimum frame. However, depending on how the region is positioned w.r.t. the bin borders, the region may overlap with an additional bin (generating also a new replicate). Therefore, the number of generated replicates for a single region of the experiment can either be μ or $\mu + 1$.

Example. In figure Fig.4.2, both examples use the same region length (50 bases) and the same bin size (80 bases). The minimum frame is computed as $\mu = \left\lceil \frac{50}{80} \right\rceil = 1$. In *EX1*, the number of generated replicates equals the

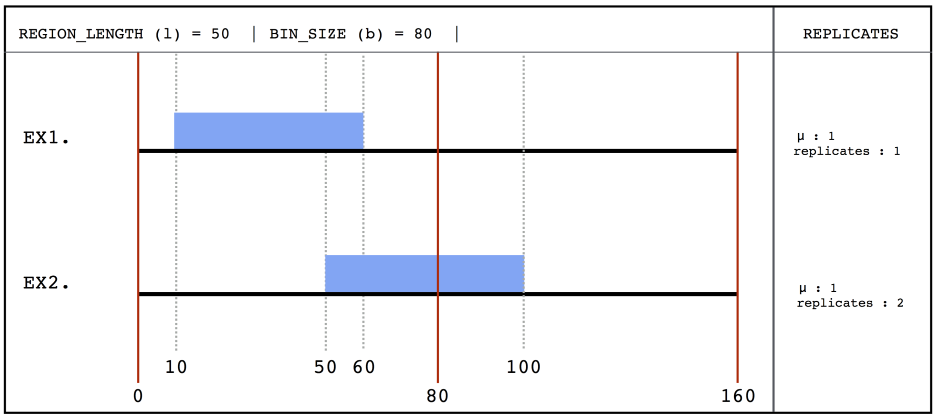


Figure 4.2: Two regions having the same length generating different numbers of replicates depending on their position w.r.t. the bin borders.

minimum frame. In *EX2*, due to the particular positioning of the region w.r.t. the bin borders, one additional replicate is generated.

Now, we could simply choose to estimate the number of replicates of a binned experiment region either with μ (underestimating) or $\mu + 1$ (overestimating). However, even if the error in the estimation of the replicates for a single region could be considered accceptable, when we try to estimate the replicates for all the regions in a datasets, this error becomes relevant and cannot be neglected.

The solution we use is to take a weighted average that considers how frequently the region is fully contained in a minimum frame and how frequently it "crosses" the borders of a minumum frame, overlapping with an additional bin, assuming that a region can be located at any position with equal probability. If we think to move a region along the chromosome from left to right, one base at time, the relative position of the region w.r.t. the bin borders becomes the same after b moves. This implies that the number of bins it overlaps, depending on how it is positioned, repeats with period b. Considering b possible positions, the number of times the region overlaps with μ bins equals the number of times the region remains within the borders of a minimum frame, i.e.:

$$n_\mu = \mu \cdot b - l + 1 \tag{4.3}$$

where $\mu \cdot b$ is the length (in bases) of the minimum frame. The number of times the region overlpas with $\mu + 1$ bins is instead the number of remaining positions, i.e.:

$$n_{\mu+1} = b - n_\mu = b \cdot \Big(1 - \mu\Big) + l - 1 \tag{4.4}$$

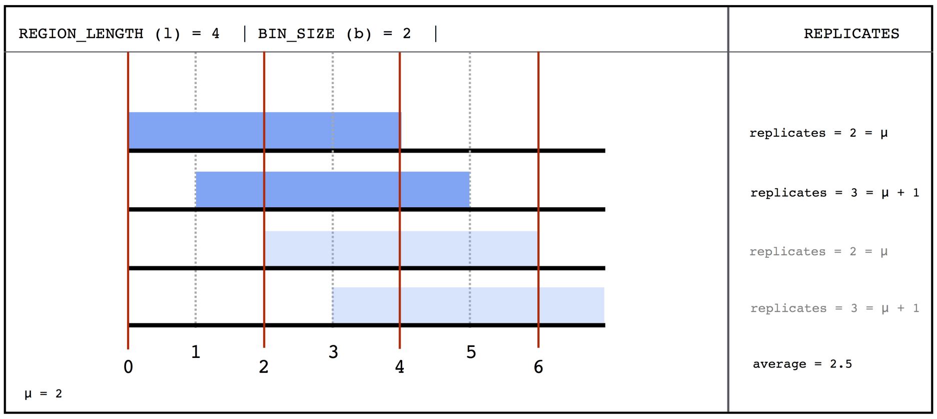


Figure 4.3: Moving the same region along the chromosome one base at time generates different number of replicates that are periodic with period 2 (bin size).

Note that (4.4) can equal zero, for example when $l = 3$ and $b = 2$. Finally, the weighted average is given by:

$$\frac{n_\mu \cdot \Big(\mu\Big) + n_{\mu+1} \cdot \Big(\mu + 1\Big)}{b} = \frac{l-1}{b} + 1$$

i.e. (4.2). □

Example. In Fig.4.3, we show how moving a region one base at time from left to right, the number of replicates is periodic with period that equals the bin size (2 bases). The number of possible positions we have to consider is then 2. The number of replicates that are generated after binning can either be $\mu = 2$ or $\mu + 1 = 3$. The number of positions in which the generated replicates are 2 can be computed with (4.3), i.e.

$$n_2 = \mu \cdot b - l + 1 = 4 - 4 + 1 = 1$$

The number of positions in which one more bin is intersected is computed with (4.4), i.e.

$$n_{2+1} = b \cdot \Big(1 - \mu\Big) + l - 1 = -2 + 4 - 1 = 1$$

Then, the average number of replicates can be computed with (4.2), i.e.

$$r_{single}(b) = \frac{l-1}{b} + 1 = 1.5 + 1 = 2.5$$

.

Now, we would like to estimate the number of replicates generated by the binning of a whole sample. If a sample contains N regions, and l_i is the length of the i-th region in the sample, $i = 1...N$, the number of replicates generated by the whole sample can be computed as:

$$\rho_{E_i}(b) = \sum_{i=1}^{N} \left(\frac{l_i - 1}{b} + 1 \right)$$

The previous sum, can be written as:

$$\rho_{E_i}(b) = \frac{\sum_{i=1}^{N} l_i - N}{b} + N$$

taking out N we can write the previous expression as:

$$\rho_{E_i}(b) = N \cdot \left(\frac{\frac{\sum_{i=1}^{N} l_i}{N} - 1}{b} + 1 \right)$$

where $\frac{\sum_{i=1}^{N} l_i}{N}$ is the average length of the regions in the sample. From now on we refer to the average region length as w:

$$w = \frac{\sum_{i=1}^{N} l_i}{N}$$

Finally, the total number of replications $\rho_{E_i}(b)$ generated by the binning of the i-th sample of the experiment dataset can be estimated by:

$$\rho_{E_i}(b) = N_i \cdot \left(\frac{w_i - 1}{b} + 1 \right) \tag{4.5}$$

where N_i and w_i are, respectively, the number of regions and the average length of the regions in the i-th sample.

Binning the Anchor Dataset

As already said in Chapter 3, a region of the anchor dataset is replicated in those bins that overlap with the search space associated to that region. The search space is defined by the distal clauses `DLE(n)` (distance $\leq n$) and `DGE(n)` (distance $\geq n$), and by the stream clause `UP` (upstream) or `DOWN` (downstream) in the genometric predicate. In the following, we refer to the argument of the `DGE` clause as LT and to the argument of the `DLE` clause as GT.

To keep our presentation simple we do not discuss the cases in which the distal clauses have a negative argument, so, from now on we assume that $LT \geq 0$ and $GT \geq 0$.

We start considering the simple case in which only the `DLE` clause is defined in the genometric predicate. In this case, the search space, like a

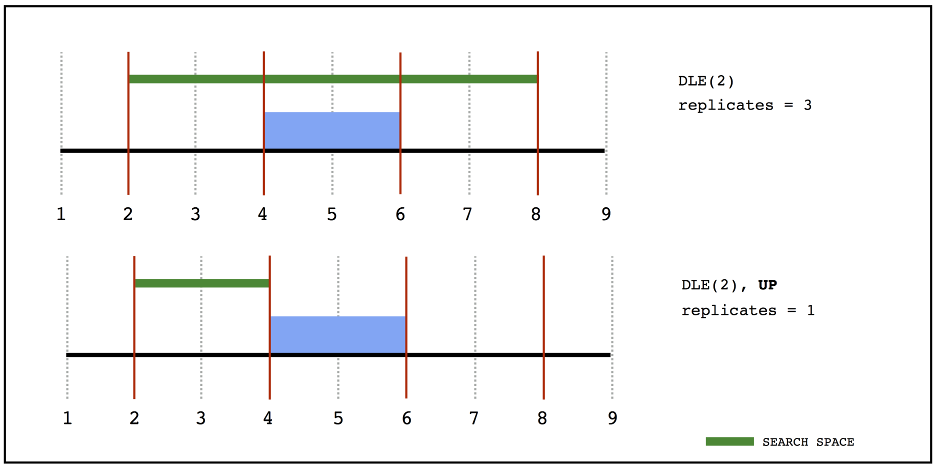


Figure 4.4: Anchor binning with two different queries on the same anchor region with no DGE clause.

region, is a contiguous interval defined over the chromosome space and its length can be computed as:

$$2 \cdot LT + l$$

where LT is the argument of the `DLE` clause (in bases) and l the region length. Since it is a contiguous interval, we can still use the expression (4.2) to compute the number of replicates, but, instead of using the length of the experiment region, we consider the length of the search space, i.e..

$$r_{single}(b) = \frac{(2 \cdot LT + l) - 1}{b} + 1$$

If the stream clause (either `UP` or `DOWN`) is defined , the length of the search space becomes simply LT, and, since the search space is still contiguous, we can exploit again (4.2) to compute the number of replicates:

$$r_{single}(b) = \frac{(LT) - 1}{b} + 1$$

To generalize the two cases, we introduce the term str, that equals 1 if the stream clause is defined, 0 if the stream clause is missing from the genometric predicate, and we estimate the number of replicates with:

$$r_{single}(b) = \frac{(2 \cdot LT + l) - str \cdot (LT + l) - 1}{b} + 1$$

We can estimate the total number of replicates generated by the binning of the i-th sample of the anchor dataset, when no `DGE` is present in the genometric predicate, with:

$$\rho^{(1)}_{A_i}(b) = N_i \cdot \left(\frac{2 \cdot LT + w_i - str \cdot (LT + w_i) - 1}{b} + 1 \right) \tag{4.6}$$

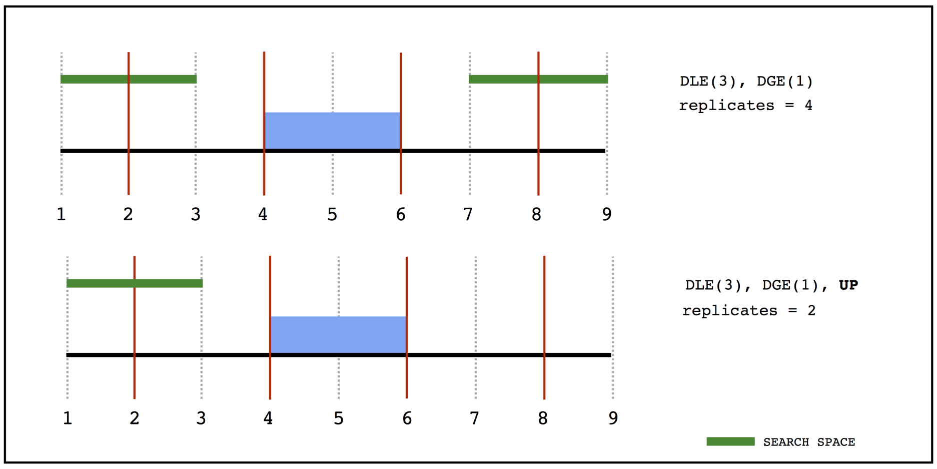


Figure 4.5: Anchor binning with two different queries on the same anchor region with DGE clause.

where N_i is the number of regions in the i-th sample.

When the `DGE` clause is present in the genometric predicate, we can distinguish two main cases:

- *stream clause is present*: corresponding to the second example in Fig.4.5, the search space is a single contiguous interval.
- *no stream clause*: corresponding to the first example in Fig.4.5, the search space is made of two distinct contiguous intervals having the same length.

In the first case, the search space is still a single contiguous interval having length $LT - GT$. Again, we exploit (4.2) to estimate the number of replicates generated by a single anchor region:

$$r_{single}(b) = \frac{(LT - GT) - 1}{b} + 1$$

The estimation of the number of replicates for the i-th sample of the reference dataset is then given by:

$$\rho^{(2)}_{A_i}(b) = N_i \cdot \left(\frac{(LT - GT) - 1}{b} + 1 \right) \tag{4.7}$$

In the second case, the search space is no more contiguous but is made of two distinct contiguous intervals, each having length $LT - GT$. We can estimate the number of intersected bins as double the number of bins that are intersected by one of the two intervals of the search space, i.e.:

$$2 \cdot \left(\frac{(LT - GT) - 1}{b} + 1 \right) \tag{4.8}$$

However, when the bin size is larger than the gap between the two intervals, i.e. $b > 2 \cdot GT + l$, the number of intersected bins may not correspond to the number of generated replicates. This happens because, when the bin size exceeds the gap, the two intervals may be overlapping with the same bin. In this case, (4.8) would consider the overlappings separetely, wrongly estimating one more replicate of the anchor region. Therefore, when $b > 2 \cdot GT + l$ we need to use another estimation that takes into account this difference. Instead of reasoning on 'how much' to subtract to (4.8) in order to get the correct estimate, we can make a simple observation: if the bin size exceeds the gap between the two intervals of the search space, there is no possibility to encounter empty bins between the leftmost and rightmost ends of the search space, because every bin overlapping with the gap will also overlap with at least one base of the search space preceding or succeeding the gap itself. In other words, when $b > 2 \cdot GT + l$, the presence of the gap can be neglected and the search space can be considered as a single contiguous interval of length $2 \cdot LT + l$, as if the `DGE` was not defined at all. Therefore, when the `DGE` clause is present in the genometric predicate but the stream clause is not, we can estimate the replicates generated by the binning of a single anchor region as:

$$r_{single}(b) = \begin{cases} 2 \cdot \left(\frac{(LT-GT)-1}{b} + 1 \right) & b \leq 2 \cdot GT + l \\ \\ \frac{(2 \cdot LT + l) - 1}{b} + 1 & b > 2 \cdot GT + l \end{cases}$$

That is, a piecewise function with a derivative discontinuity in $b = 2 \cdot GT + l$. The behavior of the function is shown in Fig.4.6.

In order to formalize the estimation of the number of replicates for the whole anchor dataset, we define:

$$r_i(b) = \begin{cases} 2 \cdot \left(\frac{(LT-GT)-1}{b} + 1 \right) & b \leq 2 \cdot GT + w_i \\ \\ \frac{(2 \cdot LT + w_i) - 1}{b} + 1 & b > 2 \cdot GT + w_i \end{cases} \tag{4.9}$$

Then, the estimation of the number of replicates generated by the binning of the i-th sample of the reference dataset when the `DGE` clause is present in the genometric predicate but the stream clause is not, is given by:

$$\rho^{(3)}_{A_i}(b) = N_i \cdot r_i(b) \tag{4.10}$$

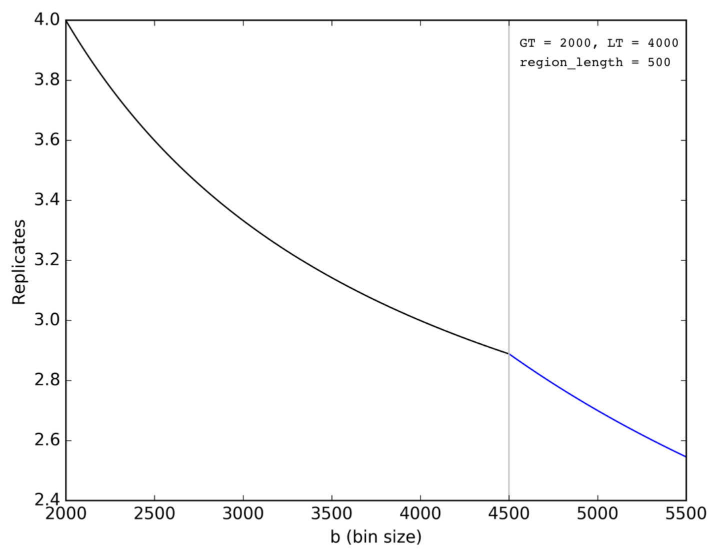


Figure 4.6: Behavior of the function estimating the number of replicates generated by the binning of an anchor region with `DGE` defined and without stream clause.

Finally, the number of replicates generated by the binning of the i-th sample of the reference dataset is given, by (4.6), (4.10) and (4.7), depending on the case. All the cases are summarised by the following:

$$\rho_{A_i}(b) = \begin{cases} N_i \cdot \left(\frac{2 \cdot LT + w_i - str \cdot (LT + w_i) - 1}{b} + 1 \right) & \text{no } \texttt{DGE} \\ N_i \cdot \left(\frac{(LT - GT) - 1}{b} + 1 \right) & \texttt{DGE}, \texttt{stream} \\ N_i \cdot r_i(b) & \texttt{DGE} \text{ , no } \texttt{stream} \end{cases} \tag{4.11}$$

where $r_i(b)$ is defined in (4.9).

Estimating the number of replicates in each bin

Now, we would like to estimate, for each binned sample of the anchor (experiment) dataset, the number of replicates that ends up into a single bin. Here we care only of those bins that, after binning, contain at least one replicate; we call them *useful bins* and we use *U(b)* to refer to their cardinality as a function of the bin size.

In order to compute the number of per-bin replicates $\delta_i(b)$, of a sample i, we take the total number of replicates generated by that sample and we divide it by the number of useful bins:

$$\delta_i(b) = \frac{\rho_i(b)}{U_i(b)}$$

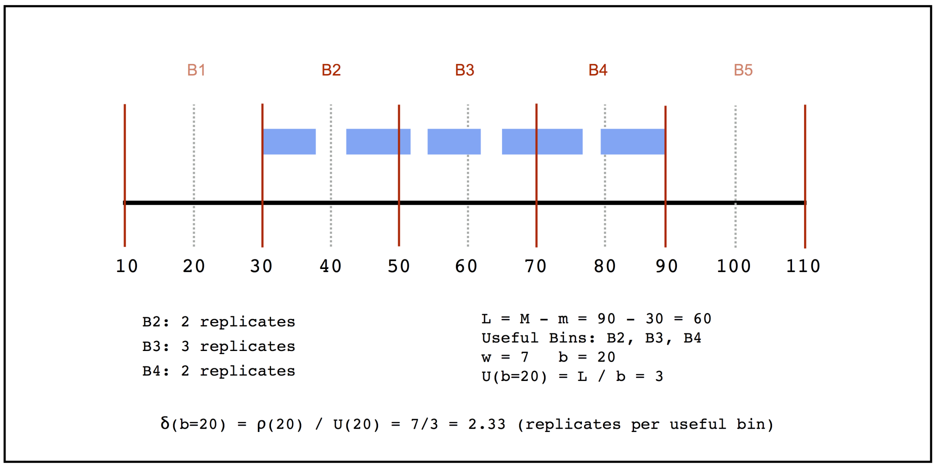


Figure 4.7: Example of estimation of the number of replicates in each useful bin of an experiment sample.

A precise estimation of *U(b)* would require the knowledge of some characteristics of the dataset, such as the distribution of region distances and a measure of the number of regions overlapping within the same sample, that we do not take into account for the reasons provided in 4.2.
We estimate the number of useful bins for the i-th sample of a dataset as:

$$U_i(b) = \frac{M - m}{b} = \frac{L_i}{b} \tag{4.12}$$

where L_i is difference between the *stop* coordinate of the rightmost region (M) an the *start* coordinate of the leftmost region (m) in the sample. This estimate tends to overestimate the number of useful bins when the bin size is smaller than the distance between regions, which, in most cases, corresponds to an unoptimal situation.
Finally, the number of per-bin replicates for the i-th anchor sample is given by:

$$\delta_{A_i}(b) = b \cdot \frac{\rho_{A_i}(b)}{L_i} \tag{4.13}$$

and for the experiment dataset:

$$\delta_{E_i}(b) = b \cdot \frac{\rho_{E_i}(b)}{L_i} \tag{4.14}$$

An example is provided in Fig. 4.7.

4.4.2 Modeling the cost function

In order to build a model of the cost function, using the approach described in 4.3, it is necessary to identify, within the algorithm implementing the Join operation, the dominant time complexities that depend on the bin size. We can neglect every other time complexity that is not dependent on the bin size, because, as discussed in 4.3, constants do not change the "shape" of the cost function and the position of optimal bin size.

For all the operations preceding the Join (Spark Join, see Fig. 3.5), that performs the cross product between the anchor regions and the experiment regions in each bin, the dominant time complexity can be assumed linear in the number of replicates, i.e. :

$$\tau_1(b) = \sum_{i=1}^{S_A} \rho_{A_i}(b) + \sum_{i=1}^{S_E} \rho_{E_i}(b) \tag{4.15}$$

For the Join (Spark Join), if $\beta(b)$ is the number of bins generated with a bin size b, the time complexity can be expressed as:

$$\tau_2(b) = \beta(b) \cdot \left(\sum_{i=1}^{S_A} \delta_{A_i}(b) \cdot \sum_{i=1}^{S_E} \delta_{E_i}(b) \right) \tag{4.16}$$

where $\sum_{i=1}^{S_A} \delta_{A_i}(b)$ is the total number of anchor regions replicates in each bin, $\sum_{i=1}^{S_E} \delta_{E_i}(b)$ is the total number of experiment regions replicates in each bin, and $\beta(b)$ estimates the number of bins having at least one reference region and at least one anchor region, i.e. the number of bins in which anchor and experiment regions can be joined.
We estimate $\beta(b)$ as:

$$\beta(b) = min\Big(max_{i=1..S_A}\big(U_i(b)\big), max_{i=1..S_E}\big(U_i(b)\big) \Big)$$

That is, we estimate the number of bins having both anchor and experiment regions as the minimum between the maximum number of useful bins among the anchor samples and the maximum number of useful bins among the experiment samples. Replacing the definition of $U_i(b)$, given by (4.12), it is possible to write $\beta(b)$ as:

$$\beta(b) = \frac{1}{b} \cdot min\Big(max_{i=1..S_A}\big(L_i\big), max_{i=1..S_E}\big(L_i\big) \Big)$$

We define L_* as

$$L_* = min\Big(max_{i=1..S_A}\big(L_i\big), max_{i=1..S_E}\big(L_i\big) \Big)$$

and simplify the expression of $\beta(b)$ as follows:

$$\beta(b) = \frac{L_*}{b}$$

Finally, the cost function, according to the representation given by (4.1), in which we neglect the term k, can be expressed as:

$$c(b) = a_1 \cdot \tau_1(b) + a_2 \cdot \tau_2(b)$$

Replacing (4.15) and (4.16), the cost function becomes

$$c(b) = a_1 \cdot \left(\sum_{i=1}^{S_A} \rho_{A_i}(b) + \sum_{i=1}^{S_E} \rho_{E_i}(b) \right) + a_2 \cdot \frac{L_*}{b} \cdot \left(\sum_{i=1}^{S_A} \delta_{A_i}(b) \cdot \sum_{i=1}^{S_E} \delta_{E_i}(b) \right) \quad (4.17)$$

4.4.3 Behavior of the cost function and optimality

Replacing every term with its definition and arranging the equation in a different way, so that the dependence on b is clear, (4.17) can be expressed as:

$$c(b) = \frac{1}{b} \cdot (a_1 \cdot P + a_2 \cdot Q) + b \cdot a_2 \cdot R + (a_1 \cdot T + a_2 \cdot V)$$

where a_1 and a_2 are parameters to be tuned , while P, Q, R, T, and V are constants[3] introduced to make the equation more readable and will defined soon for each specific case.
We use this definition of the cost function to describe all the cases except the one in which the `DGE` is present in the genometric predicate but the stream clause is not present; that case is treated separately.
Dropping constant terms from the equation, the cost function becomes:

$$c(b) = \frac{1}{b} \cdot (a_1 \cdot P + a_2 \cdot Q) + b \cdot a_2 \cdot R$$

Case 1: `DGE` not present

In this case the cost function is defined as:

$$c_1(b) = \frac{1}{b} \cdot (a_1 \cdot P_1 + a_2 \cdot Q_1) + b \cdot a_2 \cdot R \quad (4.18)$$

where:

$$P_1 = \sum_{i=1}^{S_A} N_{A_i} \cdot (2 \cdot LT + w_{A_i} - str \cdot (LT + w_{A_i}) - 1) + \sum_{i=1}^{S_E} N_{E_i} \cdot (w_{E_i} - 1)$$

$$Q_1 = L_* \cdot \sum_{i=1}^{S_A} \frac{N_{A_i} \cdot (2 \cdot LT + w_{A_i} - str \cdot (LT + w_{A_i}) - 1)}{L_{A_i}} \cdot \sum_{i=1}^{S_E} \frac{N_{E_i} \cdot (w_{E_i} - 1)}{L_{E_i}}$$

[3]not depending on b; can be computed as a combination of features.

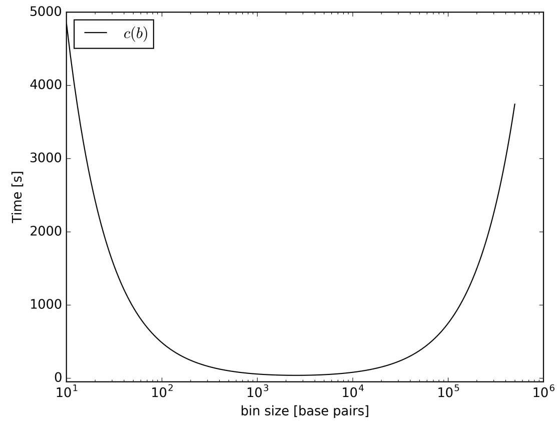


Figure 4.8: Behavior of the cost function for all the cases except the case in which the DGE is present in the genometric predicate without stream clause.

and:

$$R = L_* \cdot \sum_{i=1}^{S_A} \frac{N_{A_i}}{L_{A_i}} \cdot \sum_{i=1}^{S_E} \frac{N_{E_i}}{L_{E_i}} \tag{4.19}$$

The behavior of the cost function is depicted in Fig.4.8. In this case, the cost function is a strictly convex function that admits a single global minimum. The minimum, i.e. the optimal bin size is given by:

$$b_1^* = \sqrt{\frac{\frac{a_1}{a_2} \cdot P_1 + Q_1}{R}} \tag{4.20}$$

Case 2: DGE and stream clause present

In this case the cost function is defined as:

$$c_2(b) = \frac{1}{b} \cdot (a_1 \cdot P_2 + a_2 \cdot Q_2) + b \cdot a_2 \cdot R \tag{4.21}$$

where:

$$P_2 = \sum_{i=1}^{S_A} N_{A_i} \cdot (LT - GT - 1) + \sum_{i=1}^{S_E} N_{E_i} \cdot (w_{E_i} - 1)$$

$$Q_2 = L_* \cdot \sum_{i=1}^{S_A} \frac{N_{A_i} \cdot (LT - GT - 1)}{L_{A_i}} \cdot \sum_{i=1}^{S_E} \frac{N_{E_i} \cdot (w_{E_i} - 1)}{L_{E_i}}$$

while R is still defined by 4.19.

The behavior of the cost function is still the same as in the previous case, i.e. the one depicted in Fig.4.8. In this case, the cost function is a strictly convex function that admits a single global minimum.
The minimum, i.e. the optimal bin size is given by:

$$b_2^* = \sqrt{\frac{\frac{a_1}{a_2} \cdot P_2 + Q_2}{R}} \tag{4.22}$$

Case 3: DGE present without stream clause

In this case the cost function is a piecewise function defined as:

$$c_3(b) = \begin{cases} c(b)_{left} & b \leq 2 \cdot GT + w_r \\ c(b)_{right} & b > 2 \cdot GT + w_r \end{cases}$$

where w_r is the average region length of the anchor dataset and:

$$c(b)_{left} = \frac{1}{b} \cdot (a_1 \cdot P_{left} + a_2 \cdot Q_{left}) + a_2 \cdot b \cdot R_{left}$$

$$c(b)_{right} = \frac{1}{b} \cdot (a_1 \cdot P_{right} + a_2 \cdot Q_{right}) + a_2 \cdot b \cdot R_{right}$$

and:

$$R_{right} = L_* \cdot \sum_{i=1}^{S_A} \frac{N_{A_i}}{L_{A_i}} \cdot \sum_{i=1}^{S_E} \frac{N_{E_i}}{L_{E_i}}$$

$$P_{right} = \sum_{i=1}^{S_A} N_{A_i} \cdot (2 \cdot LT + w_{A_i} - 1) + \sum_{i=1}^{S_E} N_{E_i} \cdot (w_{E_i} - 1)$$

$$Q_{right} = L_* \cdot \sum_{i=1}^{S_A} \frac{N_{A_i} \cdot (2 \cdot LT + w_{A_i} - 1)}{L_{A_i}} \cdot \sum_{i=1}^{S_E} \frac{N_{E_i} \cdot (w_{E_i} - 1)}{L_{E_i}}$$

$$R_{left} = 2 \cdot R_{right}$$

$$P_{left} = \sum_{i=1}^{S_R} N_{A_i} \cdot 2 \cdot (LT - GT - 1) + \sum_{i=1}^{S_E} N_{E_i} \cdot (w_{E_i} - 1)$$

$$Q_{left} = L_* \cdot \sum_{i=1}^{S_A} \frac{N_{A_i} \cdot 2 \cdot (LT - GT - 1)}{L_{A_i}} \cdot \sum_{i=1}^{S_E} \frac{N_{E_i} \cdot (w_{E_i} - 1)}{L_{E_i}}$$

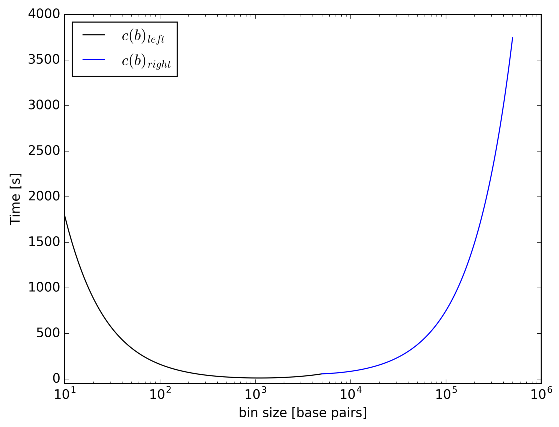


Figure 4.9: Behavior of the cost function in the case in which the DGE is present in the genometric predicate without stream clause.

The behavior of the cost function is depicted in Fig.4.9. In this case the function may have local minima. The global minimum is evaluated considering the global minimum of $c(b)_{left}$ and $c(b)_{right}$:

$$b^*_{left} = \sqrt{\frac{\frac{a_1}{a_2} \cdot P_{left} + Q_{left}}{R_{left}}}$$

$$b^*_{right} = \sqrt{\frac{\frac{a_1}{a_2} \cdot P_{right} + Q_{right}}{R_{right}}}$$

The global minimum b^*_3 depends on the position of these two minima w.r.t. the critical point $c = 2 \cdot GT + w_r$:

- if $b^*_{left} \leq c$ and $b^*_{right} \geq c:$ the minimum is:

$$b^*_3 = \begin{cases} b^*_{left} & if \quad c(b^*_{left})_{left} = min\Big(c(b^*_{left})_{left}, c(b^*_{right})_{right}\Big) \\ b^*_{right} & if \quad c(b^*_{right})_{right} = min\Big(c(b^*_{left})_{left}, c(b^*_{right})_{right}\Big) \end{cases}$$

- if $b^*_{left} > c$ and $b^*_{right} \geq c:$ the minimum is:

$$b^*_3 = b^*_{right}$$

- if $b^*_{left} \leq c$ and $b^*_{right} < c:$ the minimum is:

$$b^*_3 = b^*_{left}$$

- if $b^*_{left} > c$ and $b^*_{right} < c$: this is a limit case since the minimum equals the critical point:

$$b^*_3 = 2 \cdot GT + w_r$$

4.4.4 Model validation

Each of the following tests compares the behavior of a real execution of the Join operation with the behavior predicted by the model built in the previous subsections. In order to validate the model, the implementation has been extended with some scripts that are able to generate synthesized datasets according to some predefined characteristics, run the query with different bin sizes and measure the execution time associated to each bin size. Each bin size has been tested several times and the minimum value (in seconds) has been taken at the end. Each plot shows two tests that differ only for one characteristic, either of the datasets or of the query. For each test the real execution is represented as a set of markers (round or diamond) connected by a straight dashed line. A marker at coordinate (b, t) represents the execution time t measured binning with bin size b. The model of the execution time is represented as a continuos line close to the dashed line representing the real execution. Since the model does not include the prediction of constant terms, that do not affect the minimization problem, the straight line would always be positioned close to the x-axis. To improve the visualization, the straight lines representing the prediction have been horizontally aligned to the respective dashed lines representing the real execution. Synthesized data are close to the assumptions made for our heuristics: regions are almost uniformly distributed and belong to the same chromosome. Variations shown in each test are made w.r.t. to a default set of dataset features reported in Table 4.3. Last test is instead performed on real datasets, where

Table 4.3: Default synthesized data features.

Feature	Value
S_A	1
S_E	5
N	$25 \cdot 10^4$
w	100
$M - m$	$5 \cdot 10^7$

the assumptions do not hold anymore.

Anchor Samples

Fig. 4.10. shows the execution of the same query first on an anchor dataset made of 1 sample and then on a anchor dataset of 50 samples. As the number of anchor samples increases the optimality range shrinks and the curve becomes more steep. Moreover the optimal bin size becomes smaller, as expected.

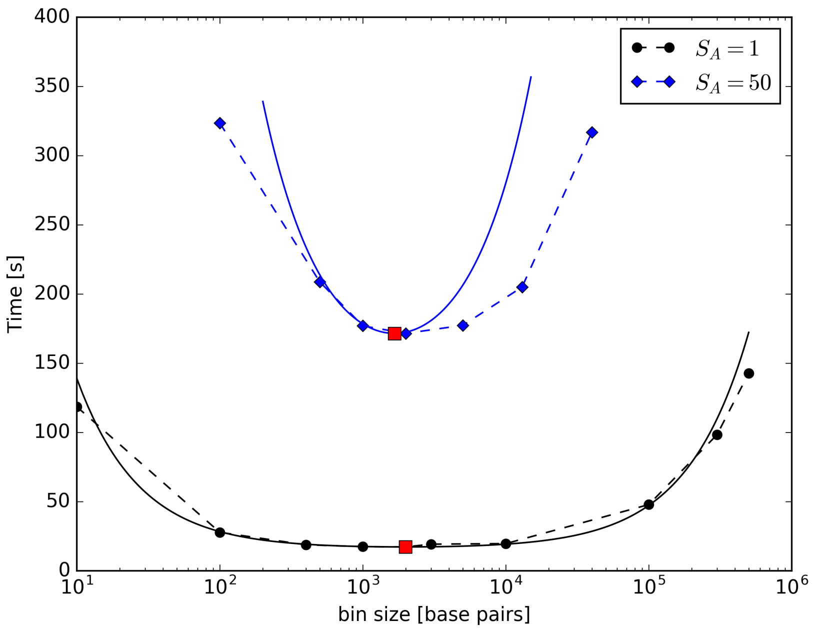


Figure 4.10: Real behavior and prediction of the same query on two datasets with different number of anchor samples.

Experiment Samples

Fig. 4.11. shows the execution of the same query first on a experiment dataset made of 5 samples and then on a experiment dataset of 20 samples. As the number of experiment samples increases the optimality range shrinks and the curve becomes more steep. Moreover the optimal bin size becomes smaller, as expected.

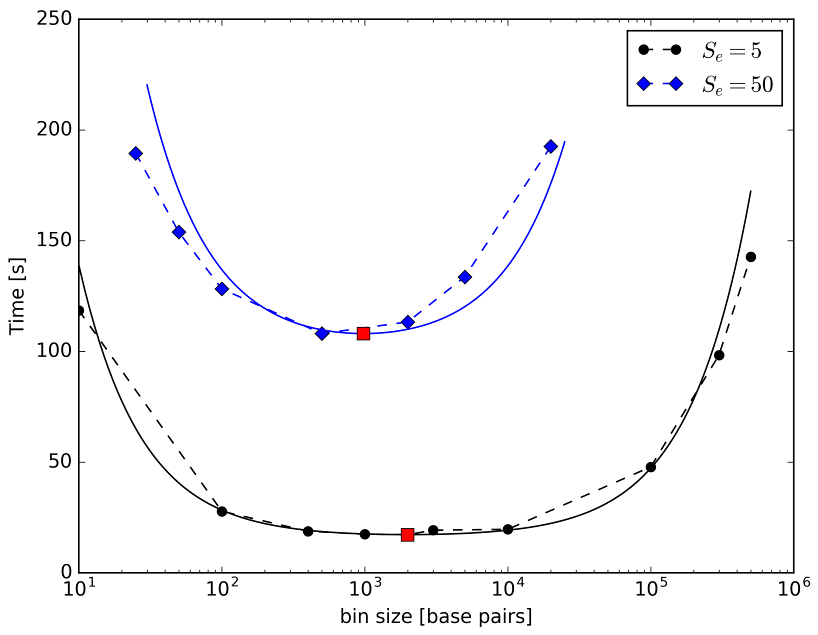


Figure 4.11: Real behavior and prediction of the same query on two datasets with different number of experiment samples.

Number of Regions

Fig. 4.12. shows the execution of the same query first on datasets made of 10k regions per sample and then on datasets made of 250k regions per sample. As the number of regions increases the optimality range shrinks and the curve becomes more steep. The optimal bin size becomes smaller.

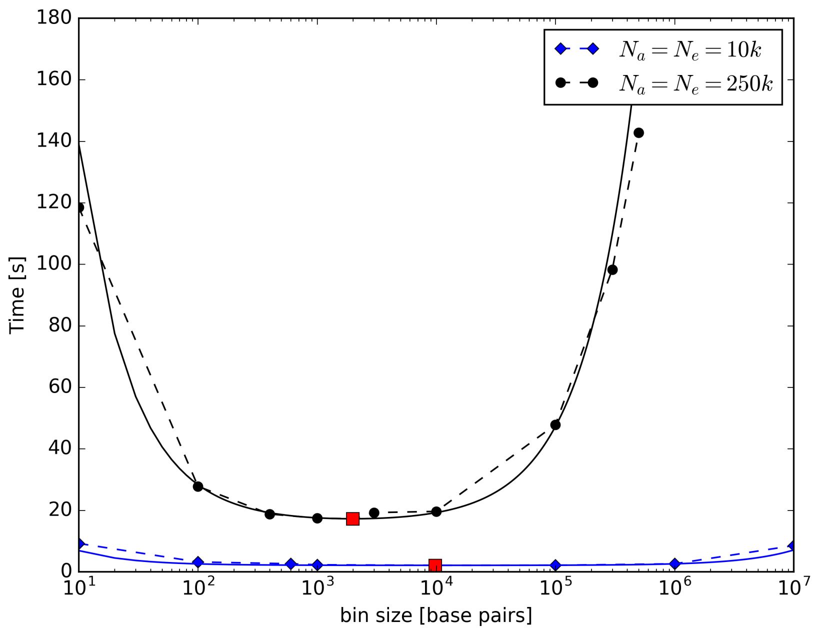


Figure 4.12: Real behavior and prediction of the same query on datasets with different number of regions.

Region length

Fig. 4.13. shows the execution of the same query first on a dataset with average region length of 10 base pairs and then on a dataset with average region length of 1000 base pairs. As the region length increases the optimal bin size becomes bigger.

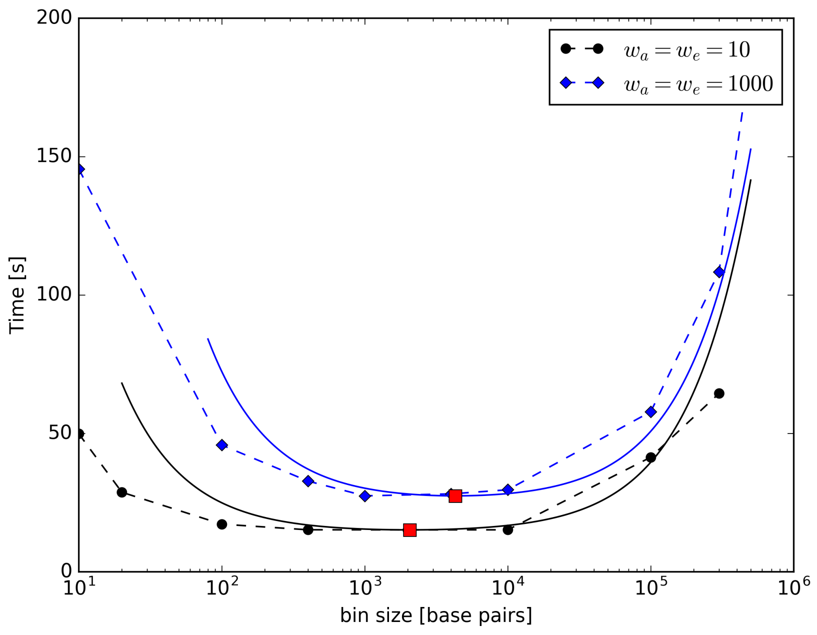


Figure 4.13: Real behavior and prediction of the same query on datasets with different average region length.

Search Space

Fig. 4.14. shows the execution of two different queries on the same datasets. Bigger search space requires bigger bins in order to avoid the production of many replicates.

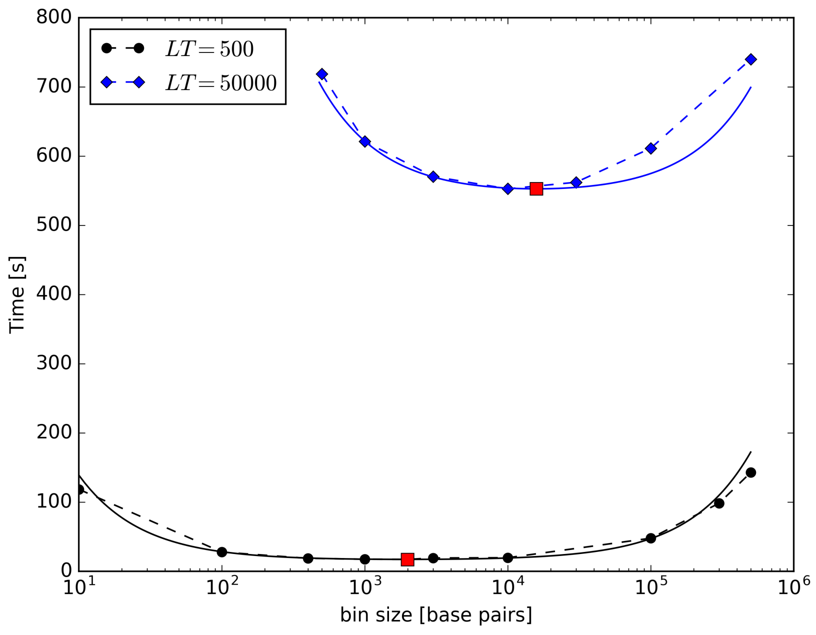


Figure 4.14: Real behavior and prediction of two different queries on the same datasets.

Real Data

This test was made on some samples taken from two real dataset available in our Repository, known as Bed Annotations (Anchor) and Narrow Peaks (Experiment), joined with a search space of length 10^6 bases. Datasets' features are summarized in Table 4.4. Tests were executed on an Amazon EMR Cluster made of 1+5 instances of type m3.2xlarge.
In this case we wanted to check how much our model was robust to violations of the assumptions we made in sec. 4.4.1. Note that the execution time is expressed in minutes.

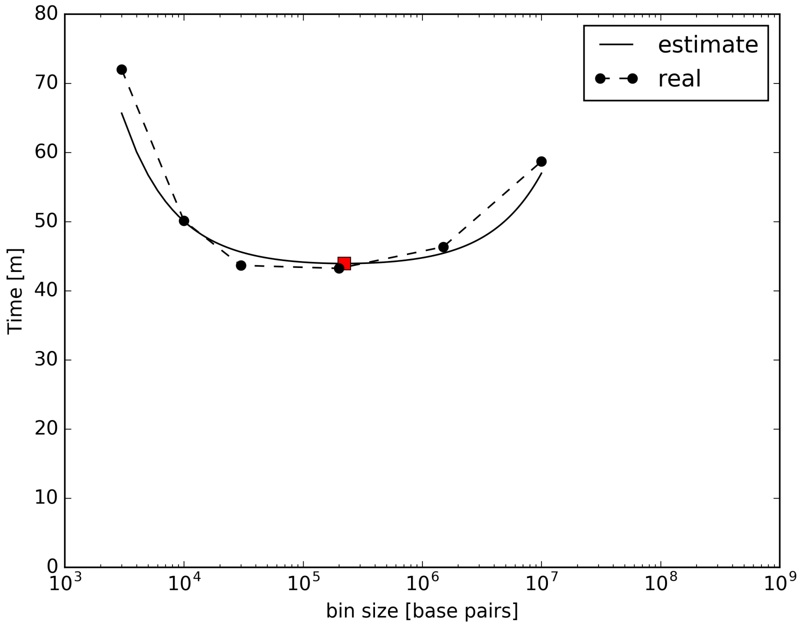


Figure 4.15: Prediction and real execution of a Join operation on real datasets.

Table 4.4: Datasets' Features . S, N and w are averaged over the number of samples.

	Anchor	Experiment
S	15	30
N	$6.7 \cdot 10^4$	$1.8 \cdot 10^5$
w	$6.1 \cdot 10^4$	$2.3 \cdot 10^3$
$L = M - m$	$2.49 \cdot 10^8$	$2.49 \cdot 10^8$

4.5 Map

4.5.1 Estimating replication

Binning algorithm for Map operation works in the same way for both the reference and the experiment datasets and is similar to the binning of the experiment dataset in the Join operation. The main difference is in the binning of the reference dataset, because the binning algorithm replicates each binned reference sample as many times as the number of experiment samples. Therefore, the total number of replicates $\rho_{R_i}(b)$ generated by the binning of the i-th sample of the reference dataset can be estimated by:

$$\rho_{R_i}(b) = S_e \cdot N_{R_i} \cdot \left(\frac{w_{R_i} - 1}{b} + 1 \right) \tag{4.23}$$

where N_{R_i} ind w_{R_i} are, respectively, the number of regions and the average length of the regions in the i-th sample and S_e is the number of samples in the experiment dataset. Instead, the total number of replications $\rho_{E_i}(b)$ generated by the binning of the i-th sample of the experiment dataset can be estimated in the same way as we did for the Join operation, i.e.:

$$\rho_{E_i}(b) = N_{E_i} \cdot \left(\frac{w_{E_i} - 1}{b} + 1 \right) \tag{4.24}$$

where N_{E_i} ind w_{E_i} are, respectively, the number of regions and the average length of the regions in the i-th sample.

Estimating the number of replicates in each bin

The number of per-bin replicates for the i-th reference sample is given by:

$$\delta_{R_i}(b) = b \cdot \frac{\rho_{R_i}(b)}{L_i} \tag{4.25}$$

and for the experiment dataset:

$$\delta_{E_i}(b) = b \cdot \frac{\rho_{E_i}(b)}{L_i} \tag{4.26}$$

where, in both cases

$$L_i = M_i - m_i$$

as already discussed in 4.4.1

4.5.2 Modeling the cost function

As for the Join operation, in order to build a model of the cost function, using the approach described in 4.3, it is necessary to identify, within the algorithm implementing the Map operation, the dominant time complexities that depend on the bin size.
For all the operations preceding the comparisons between binned regions, the dominant time complexity is assumed to be linear in the number of replicates, i.e. :

$$\tau_1(b) = \sum_{i=1}^{S_R} \rho_{R_i}(b) + \sum_{i=1}^{S_E} \rho_{E_i}(b) \tag{4.27}$$

Differently from the Join operation, in which regions within each bin are compared after they are joined (spark join), comparisons among regions for the Map operation are performed using a more efficient algorithm that sorts binned regions with a time complexity of $O(n \log n)$. If $\beta(b)$ is the number of bins generated with a bin size b, the dominant time complexity related to comparisons can be expressed as :

$$\tau_2(b) = \beta(b) \cdot \left(\sum_{i=1}^{S_R} \delta_{R_i}(b) \cdot \log_2 \big(\sum_{j=1}^{S_R} \delta_{R_j}(b) \big) + \sum_{i=1}^{S_E} \delta_{E_i}(b) \cdot \log_2(\delta_{R_i}(b)) \right) \tag{4.28}$$

where $\sum_{i=1}^{S_R} \delta_{R_i}(b) \cdot \log_2 \big(\sum_{j=1}^{S_R} \delta_{R_j}(b) \big)$ and $\sum_{i=1}^{S_E} \delta_{E_i}(b) \cdot \log_2(\delta_{R_i}(b))$ are, respectively, the complexity of sorting reference regions and experiment regions in each bin. The difference in the two expressions is due to the different way binned regions are sorted for the reference and for the experiment. While experiment regions in each bin are sorted sample by sample, all reference regions are sorted in each bin independently on the sample they belong to.
As seen for the Join, $\beta(b)$ estimates the number of bins having at least one reference region and at least one experiment region in it, i.e. the number of bins in which reference and experiment regions can be compared. The estimation of $\beta(b)$ is the same as for the Join operation, i.e.:

$$\beta(b) = \frac{L_*}{b}$$

where L_* was estimated as:

$$L_* = min\Big(max_{i=1..S_R}(L_i), max_{i=1..S_E}(L_i) \Big)$$

Finally, the cost function, according to the representation (4.1) in which we neglect the term k, can be expressed as:

$$c(b) = a_1 \cdot \tau_1(b) + a_2 \cdot \tau_2(b) \tag{4.29}$$

4.5.3 Behavior of the cost function and optimality

Replacing every term with its definition, removing everything that does not depend on b and introducing some approximation for the logarithm, the cost function (4.29) can be expressed as:

$$c(b) = \frac{1}{b} \cdot (a_1 \cdot P + a_2 \cdot Q) + a_2 \cdot \frac{\log_2(b)}{b} \cdot R + a_2 \cdot S \cdot \log_2(b)$$

where a_1 and a_2 are parameters to be tuned , while P, Q, R and S, are constants[4] introduced to make the equation more readable and are defined as follows:

$$P = \sum_{i=1}^{S_R} N_{R_i} \cdot (w_{R_i} - 1) + \sum_{i=1}^{S_E} N_{E_i} \cdot (w_{E_i} - 1)$$

$$Q = S_E \cdot L_* \cdot \sum_{i=1}^{S_R} \frac{N_{R_i} \cdot (w_{R_i} - 1)}{L_{R_i}} \cdot \log_2 \left(\sum_{j=1}^{S_R} \frac{N_{R_i}}{L_{R_i}}\right) + \sum_{i=1}^{S_E} \frac{N_{E_i} \cdot (w_{E_i} - 1)}{L_{E_i}} \cdot \log_2 \left(\frac{N_{E_i}}{L_{E_i}}\right)$$

$$R = L_* \cdot \left(S_E \cdot \sum_{i=1}^{S_R} \frac{N_{R_i} \cdot (w_{R_i} - 1)}{L_{R_i}} + \sum_{i=1}^{S_E} \frac{N_{E_i} \cdot (w_{E_i} - 1)}{L_{E_i}} \right)$$

$$S = L_* \cdot \left(S_E \cdot \sum_{i=1}^{S_R} \frac{N_{R_i}}{L_{R_i}} + \sum_{i=1}^{S_E} \frac{N_{E_i}}{L_{E_i}} \right)$$

The behavior of the cost function is depicted in Fig.4.16. Differently from the cost function defined for the Join operation, the cost function here is more critical on its left part than in its right part, where now the growth is not so steep. This means that taking bigger bins is typically a safe choice, and that a prediction that overestimates the optimal bin size does not create big performance issues.

[4]not depending on b; can be computed as a combination of features.

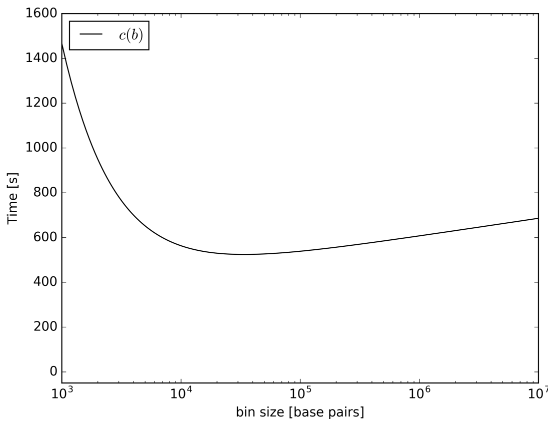


Figure 4.16: Behavior of the cost function for the Map operator.

The minimum is still found nullifying the first derivative, but only an approximate solution can be found due to the nature of the equation. Our solution is computed by:

$$b^* = -\frac{R}{S} \cdot W\left(- \frac{S \cdot 2^{-\frac{a_1 \cdot p + a_2 \cdot Q - a_2 \cdot R}{a_2 \cdot R}}}{R} \right) \tag{4.30}$$

where W is the Lambert-W function.

4.5.4 Model validation

Tests are presented in the same way they have been presented for the Join.

Reference Samples

Fig. 4.17. shows the execution of the same query first on a reference dataset made of 1 sample and then on a reference dataset of 10 samples.

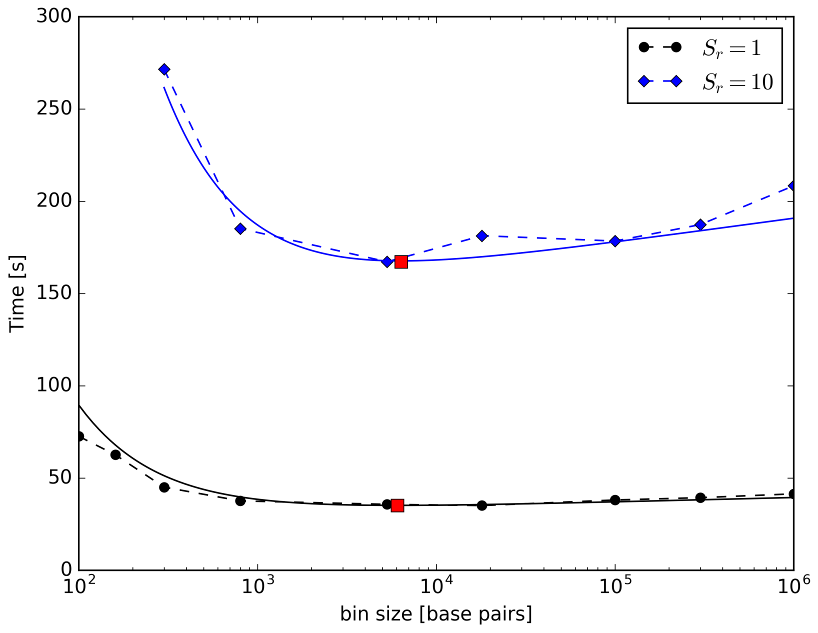


Figure 4.17: Real behavior and prediction of the same query on two datasets with different number of reference samples.

Experiment Samples

Fig. 4.18. shows the execution of the same query first on a experiment dataset made of 2 samples and then on a experiment dataset of 40 samples.

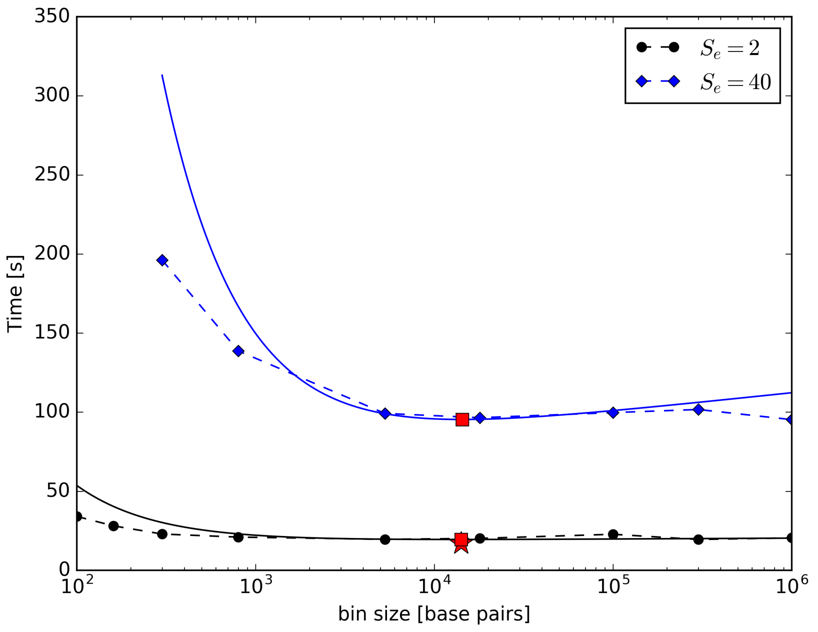


Figure 4.18: Real behavior and prediction of the same query on two datasets with different number of experiment samples.

Number of Regions

Fig. 4.19. shows the execution of the same query first on datasets made of 10k regions per sample and then on datasets made of 250k regions per sample.

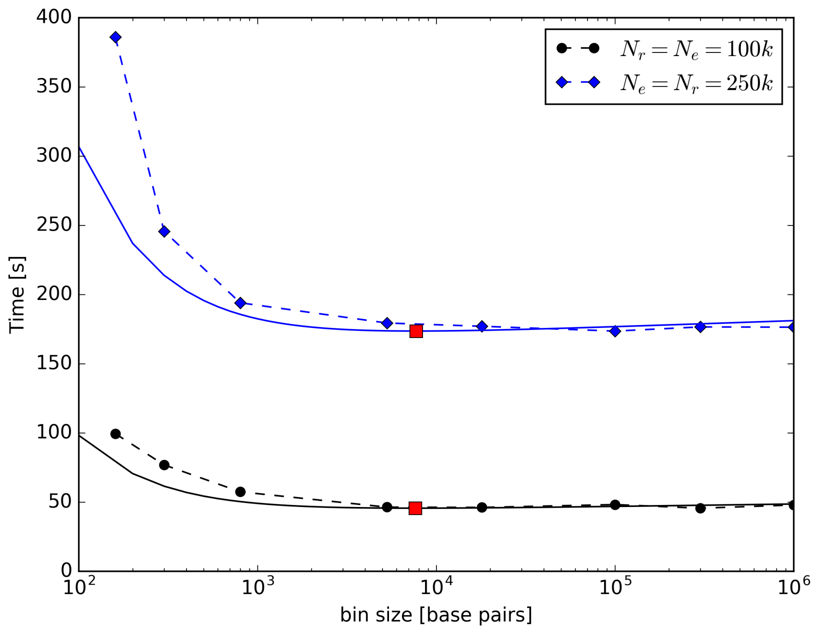


Figure 4.19: Real behavior and prediction of the same query on datasets with different number of regions.

Region length

Fig. 4.20. shows the execution of the same query first on a dataset with average region length of 10 base pairs and then on a dataset with average region length of 500 base pairs. As the region length increases the optimal bin size becomes bigger.

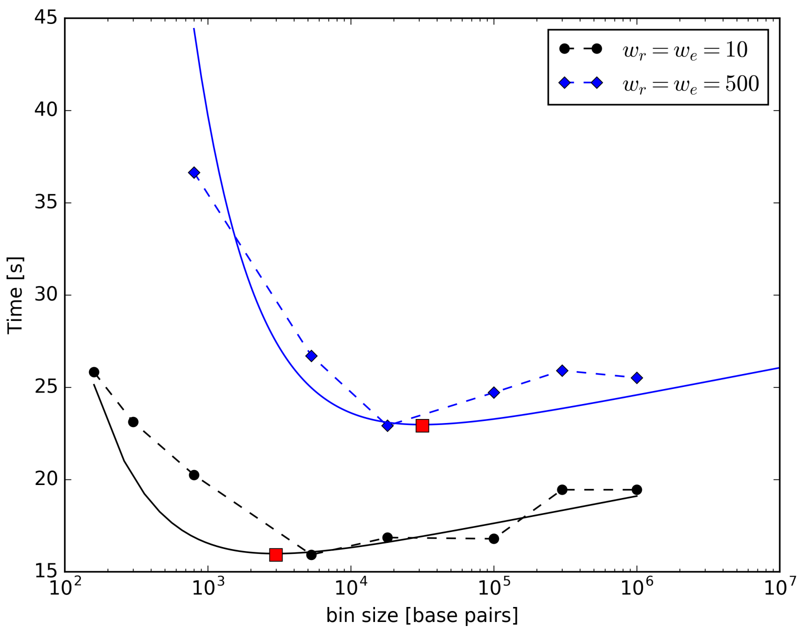


Figure 4.20: Real behavior and prediction of the same query on datasets with different average region length.

Real Data

This test was made on some samples taken from two real dataset available in our Repository, known as Bed Annotations (Anchor) and Narrow Peaks (Experiment). Datasets' features are summarized in Table 4.5. Tests were executed on an Amazon EMR Cluster made of 1+5 instances of type m3.2xlarge.
Even in this case we wanted to check how much our model was robust to violations of the assumptions we made in sec. 4.4.1.

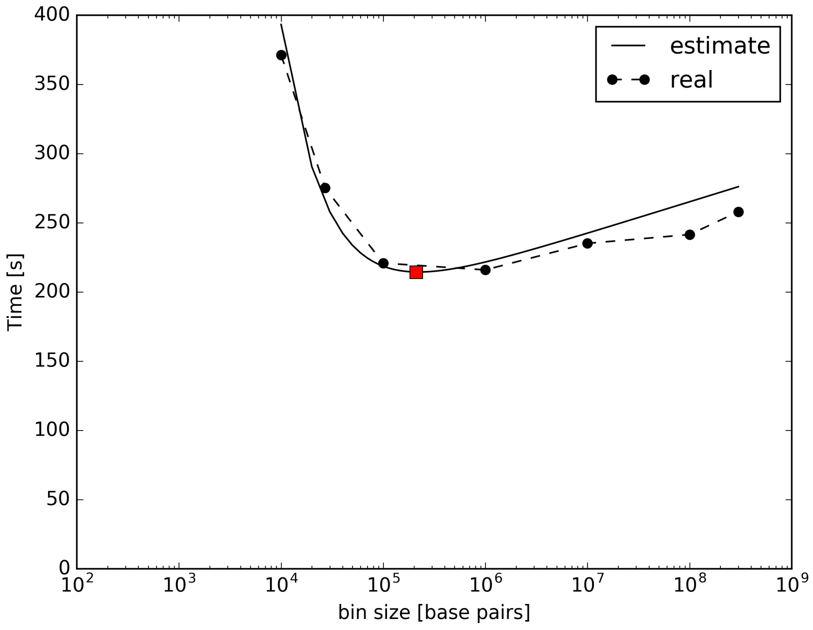


Figure 4.21: Prediction and real execution of a Map operation on real datasets.

Table 4.5: Datasets' Features . S, N and w are averaged over the number of samples.

	Reference	Experiment
S	1	2034
N	$4.9 \cdot 10^4$	$9.4 \cdot 10^4$
w	$3 \cdot 10^4$	$4 \cdot 10^6$
$L = M - m$	$2.49 \cdot 10^8$	$2.49 \cdot 10^8$

Chapter 5

GDMS Optimizer

In the previous chapter we developed and tested the optimal bin size estimators for two diverse domain specific operations. Optimal bin size estimators take in input a set of simple features extracted from the datasets, from the query, and from the results of the operations on metadata to produce as output an optimal bin size.
In this chapter we present the Optimizer, the module in charge of implementing optimal binning, and the Genomic Profiler, which, taking into account the specificity of genomic data modeling, quantitatively defines the features that better characterize a genomic dataset from the point of view of query optimization. Moreover, we discuss Profile Estimation, required to quickly compute profiles of intermediate results. At the end we describe the functioning of the Optimizer Controller, which detects operations requiring binning optimization and computes the features of their input dataset, eventually providing and optimal size for binning. Before going into details, we briefly describe the system architecture and introduce the DAG (Directed Acyclic Graph) of Operators, a way to represent the dependencies among the operations of GMQL queries.

5.1 Query Translation

A GMQL script is made of a set of operations, each acting on two different data structures: region data and metadata.
The following is an example of a GMQL script:

```
GENES = SELECT() ANNOTATIONS;
PEAKS = SELECT() BED_PEAKS;
MAPPED = MAP() PEAKS GENES;
SELECTED = SELECT(Count_PEAKS_GENES>0) MAPPED;
```

```
RELEVANT = COVER(1,2) SELECTED;
MATERIALIZE RELEVANT INTO OUTPUT;
```

The script above is compiled using the GMQL compiler (a syntax-directed translator), producing a Directed Acyclic Graph (DAG), whose nodes represent operations and arcs represent the data flow among the operations; each query in the above script will produce at least two nodes in the DAG (region and meta operations) while others are translated into even more than two operations, e.g. `SELECT` operation that generates nodes `IRReadRD`, `IRSelectRD`,`IRReadMD`, and `IRSelectMD`. The DAG corresponding to the above query is show in Fig.5.1. The dependencies between the regions DAG from/to the meta DAG are shown as dashed arrows.

All the parameters in the DAG except `BIN SIZE` are extracted from the GMQL script. For the domain-specific operators, highlighted in light blue, `BIN SIZE` is calculated by the Optimizer and injected in the DAG nodes. This parameter will tell the implementation of the DAG nodes the size that has to be used to bin the input datasets. The Optimizer, soon described in details, parses the DAG to identify the presence of one or more domain-specific operators and enables a set of tasks, like profiles estimation, to accomplish the optimization strategy that was decided for that case.

5.2 Architecture

The architecture of the GDMS (Genomic Data Management System), shown in Fig.5.2, is a four layered architecture:

- An **Access Layer**, supporting:
  - An Intermediate Representation API (Scala)
  - Python and R interfaces
  - Command line interface
  - Web Services and Web Interface
- The **Engine Components**, including:
  - GMQL Compiler, for compiling a GMQL query into a DAG (which embodies execution plans).
  - DAG Manager, for supporting the creation and dispatching of DAG operations to other components.
  - Server Manager, for managing multi-user execution and their access capabilities.

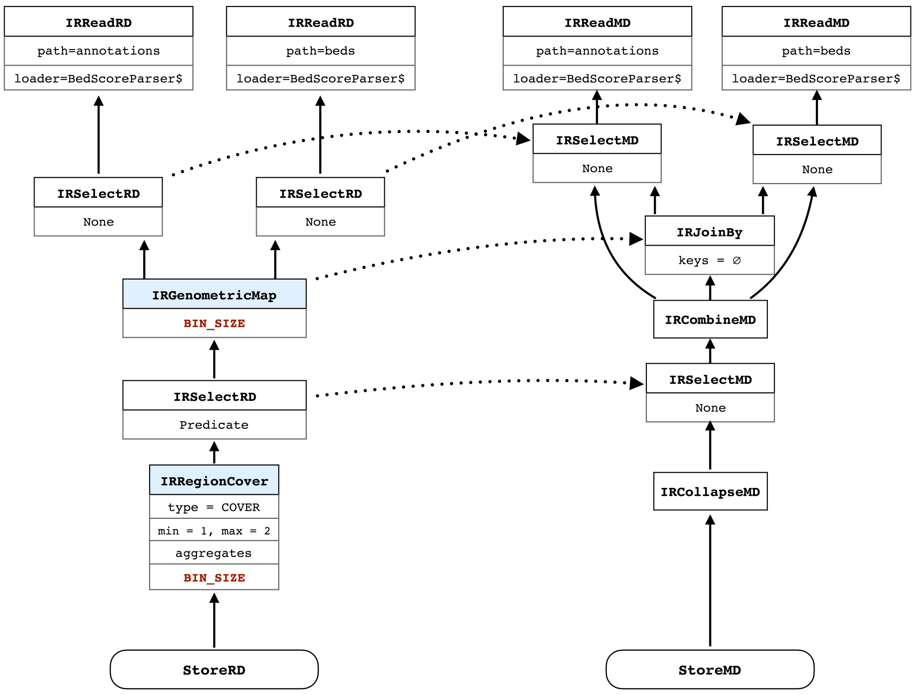


Figure 5.1: Example of operator DAG for a GMQL query.

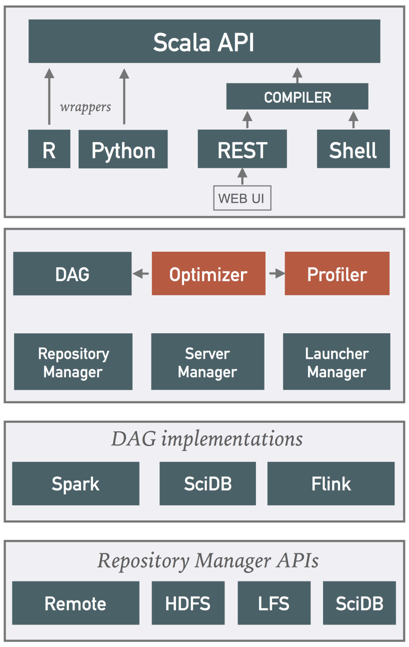


Figure 5.2: GDMS architecture.

    - Launcher Manager, for launching the execution of different implementations with different deployment modes.

- The **Implementation Components** (or executors), including three different implementations.
- The **Repository Manager APIs**, to manage the repository depending on the deployment mode.

What was listed so far were the components of the system before the development of the work discussed in this thesis. With the introduction of Optimal Binning two new components were added to the system:

- The **Genomic Profiler**: performs datasets profiling, extracting a set of features that are useful both for optimization (dynamic binning) and for data visualization (on the web interface).
- The **Optimizer**: in charge of deciding the best optimization strategies and putting them in place. This module will include also other types of optimizations that may be developed in the future and that are not directly related to optimal binning (e.g. dynamic resource allocation).

5.3 Genomic Profiler

In Chapter 4, we defined some heuristics that, given some input features, are able to output an estimate of the optimal bin size for binning. A list of features required for optimizations is shown in Table 5.1. Note that that the size of a dataset can be computed as:

$$size(DS) = \sum_{i=1...S} N_i \cdot C_i$$

The last column shows from where these features are extracted. In particular, the features can be extracted:

- From the DAG: the extraction of query parameters is easy since it is performed parsing the DAG.
- From the results of meta operations
- From dataset profiles

While extracting features from the DAG and from meta results is immediate, extracting features from the dataset requires more efforts. For this

Table 5.1: Features computed for Optimizations

Feature	Description	Origin
S	Number of samples in a dataset	Meta Operations
N	Number of regions in a sample	Data Profiling
w	Average region length	Data Profiling
m	Minimum left coordinate	Data Profiling
M	Maximum right coordinate	Data Profiling
C	Size of the region schema	Data Profiling
LT	Less than condition argument	DAG
GT	Greater than condition argument	DAG
str	Stream clause present / missing	DAG

purpose a new *Profiler* module has been added to the system. The Profiler consists of a Spark Application that can be submitted at any time to produce an XML file containing the desired features. The XML file is stored within the dataset folder with the name `profile.xml` and is intended to be used by the Optimizer, for optimal binning and future optimizations, and by the Web Interface for visualization. By default, a profiling task runs when a new dataset is added to the repository, either as a result of a query execution or when a new dataset gets imported by the Importer module. Local, HDFS and remote executions are supported. The default level of profiling granularity to compute data features is set to be on the sample level regardless of the chromosome and of the strand, since this level of granularity is enough for producing the details required by our estimators. Moreover, the code is easily extensible to support additional features extraction and different levels of granularity. Reaching chromosome granularity may be too expensive and is used only if requested by the user. An example of the XML file generated by the Profiler is shown in Fig.5.3

```
<GMQLDatasetProfile>
        <property name="samples">12</property>
        <property name="size_kb">107000526</property>
        <property name="profiling_time">7</property>
        <samples>
          <sample id="-162429929417499290"
          name="RefSeqGenes.bed">
              <property name="num_reg">50653</property>
              <property
              name="avg_reg_length">56033</property>
              <property name="min">6010</property>
              <property name="max">249213345</property>
          </sample>
          <sample>
            ...
          </sample>
         </samples>
</GMQLDatasetProfile>
```

Figure 5.3: Real example of XML file generated by the Profiler.

5.4 Optimizer

The Optimizer, whose structure is depicted in Figure. 5.4, is the module of the system in charge of controlling and implementing the available optimization strategies. The module is composed by:

- A **Heuristics Provider**: is a collection of all the heuristics used for optimization, including the mathematical models and optimal solutions defined in Chapter 4 for Optimal Binning.
- A **Profile Estimator**: used to estimate the features of intermediate results. The Profile Estimator is used to produce data profiles with higher performance than scanning the data for building a real profile, but on the expense of the accuracy. We choose to use the data profile estimator to avoid profiling big data on the fly, thus reducing the optimization process overhead and gain higher overall performance.
- A **Controller**: parses the DAG to detect the possibility of putting in place an optimization strategy. Depending on the DAG, it selects the proper heuristics from the Heuristics Provider.

Now we go into the details of the two main components, the Data Profiler Estimator and the Controller.

Profile Estimator

In order to compute an optimal bin size for a given domain-specific operator, the Optimizer requires the profile of the operator's input datasets. Most of the times, those input datasets are the results of a sequence of operations

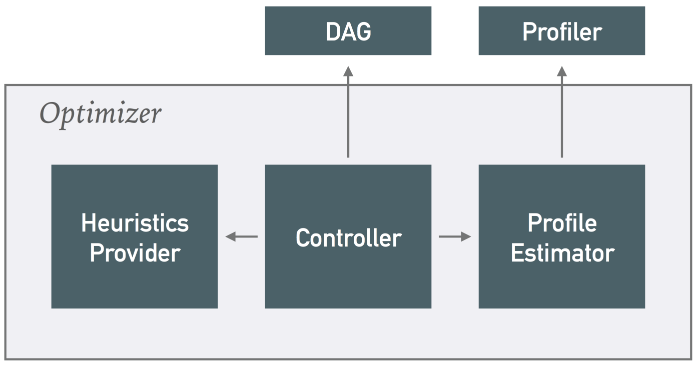


Figure 5.4: Structure of the Optimizer Module

previously applied to the original datasets loaded from the Repository. For this reason, profiles of the operator's input datasets may significantly differ from profiles of the original datasets computed by the Profiler module.

A possible solution to the problem may consist in re-profiling the results of every operation preceding the domain specific operations. However, re-profiling every time might be quite expensive and the total time spent in re-profiling may overcome the gain in time we get from performing optimal binning instead of unoptimal binning.

Another solution consists in defining a set of heuristics, ideally one for each DAG node, that allow to quickly estimate how the profile changes at the output of an operator.

The approach we use is mixed; we defined a set of simple heuristics for a subset of the language operators and we left the cases in which the output of an operator is hard to be estimated with good approximation. In that case, re-profiling is performed. Notice that re-profiling is made on datasets that are already loaded in memory, therefore there is no disk reads overhead.

Table 5.2 shows the availability of profile estimation heuristics for all GMQL operators. In case the operator is a `PROJECT` or an `EXTEND`, the output profile does not differ from the input profile . Another simple case is `UNION`; the profile of the result is built merging the two input profiles. When no region predicate is contained, the output profile of the `SELECT` is built dropping from the input profile the descriptions of those samples not satisfying the predicate on metadata. Other cases in which profiles are estimated are the following:

- `MERGE`: the output profile consists of a single sample in which:
  - *number of regions* equals the total number of regions contained

Table 5.2: Overview of the available heuristics for dataset profile estimation.

Operator	Condition	Comment
`SELECT`	no region predicate	see detailed descr.
`PROJECT`		$p_{out} = p_{in}$
`EXTEND`		$p_{out} = p_{in}$
`UNION`		$p_{out} = \bigcup p_{in}$
`MERGE`	no grouping	see detailed descr.
`GROUP`		$p_{out} = p_{in}$
`ORDER`	no regions top-k	$p_{out} = p_{in}$
`DIFFERENCE`		re-profiled
`MAP`		see detailed descr.
`JOIN`	no `INT` or `CAT` in `coord-gen`	see detailed descr.
`COVER`		re-profiled

in all the samples of the input profile; formally:

$$N = \sum_{i=1..S} N_i$$

- *average region length* is computed as the weighted average over all the samples of the input profile (weight is the number of regions); formally:

$$w = \frac{\sum_{i=1..S} N_i \cdot w_i}{N}$$

- *min left* coordinate and *max right* coordinate are computed respectively as the minimum left coordinate and the maximum right coordinate over all the samples of the input profile; formally:

$$(m, M) = (min(m_1, ..., m_S), max(M_1, ..., M_S))$$

- `MAP`: the result of a MAP operator produces samples that have the same characteristics of samples in the reference dataset. Output profile is then built by replicating input samples' profiles depending on the result of the joinby condition; formally:

$$(N, w, m, M)_{ij} = (N, w, m, M)_i$$

where sample ij is the sample generated mapping experiment j to reference i.

- `JOIN`: The output profile of a Join operation is estimated with the heuristics here presented. The notation $i \bowtie j$ is used to denote the sample of the output dataset generated by joining the i-th sample of the anchor dataset with the j-th sample of the experiment dataset. The number of samples in the output dataset depends on the input samples satisfying the meta-join condition and has as upper bound $S_{ref} \cdot S_{exp}.$. The number of regions in output is estimated as:

  $$N_{i \bowtie j} = N_i \cdot s \cdot \delta_j$$

  where s is the length of the *search space*, while δ_j is an estimation for the region density of the j-th experiment sample:

  $$\delta_j = \frac{N_j}{M_j - m_j}$$

  Other features are estimated depending on the `coord-gen` option, as follows:

  If `coord-gen = RIGHT`, then:

  $$(w, m, M)_{i \bowtie j} = (w, m, M)_i$$

  If `coord-gen = LEFT`, then:

  $$(w, m, M)_{i \bowtie j} = (w, m, M)_j$$

  Otherwise, if If `coord-gen = INT | CAT` reprofiling is performed.

5.4.1 Optimization Controller

The job of the Controller starts right after the DAG is submitted for execution and before the operations start being executed. The controller parses the DAG looking for the domain-specific operators for which optimal binning is defined, i.e. MAP and JOIN. If one or more domain-specific operators are found, the Controller enables the Profile Estimator component and sets a flag on every node of the DAG that requires its output profile to be estimated or computed.

Once the execution starts, the Controller orders profile prediction or computation of the output of every GMQL operation that finishes its execution, according to the flag set on the corresponding DAG node. Note that estimation is always made at runtime, either because it may depend on a profile that has to be computed or because it may depend on the result of a meta-operation affecting the cardinality of the samples in output.

When the execution reaches a domain-specific operation the profiles of its input datasets are combined with the operation parameters and, depending on the case, the proper heuristic is selected to compute an optimal size for binning.

Chapter 6

Conclusions and Future Work

In this thesis we developed a set of heuristics used to optimize genomic binning, an approach for parallelizing massive operations on the genome. Heuristics were derived from a mathematical model of the execution time that takes into account simple features extracted from the data, through an ad-hoc Profiler, and from a graph-based representation of the query (DAG). Moreover, in order to cut re-profiling times, we developed additional heuristics to estimate the profile of intermediate results. Even though the proposed models are quite simple and based on some assumptions, that usually do not hold in reality, we proved their robustness through some experiments on real data. Thanks to optimal binning we improved the efficiency of our system, reduced the execution time of complex queries and the consumption of computational resources.

We believe that the optimizations developed in this thesis can be extended outside the specific domain of genomics. In fact, our DNA regions are just geometric intervals in a two-dimensional space that may have a different semantics; e.g. they may represent time intervals.

Our plans for the future involve a systematic study of binning optimizations over multiple operations. Indeed, when multiple domain specific operations occur in the same query, binning of some datasets could be avoided; e.g. when two operations share the same input dataset, we may choose to bin it only once with a bin size that is acceptably good for both, or we may think to keep a result binned if some other domain-specific operation is using it as input. These kind of optimizations require the introduction of new heuristics, changes in the implementation, and a theoretical study of the dependencies among operations in the DAG. Moreover, the developed models and the information provided by the Profiler could be combined to implement dynamic resource allocation, so that a proper quantity of resources, e.g. number of executors and amount of memory, is assigned to each query. In conclusion, we improved the performance of our system and laid the bases for future research on similar optimizations that will make processing of genomic data faster, hoping it will help the bioinformatics community achieve new goals in the near future.

Bibliography

[1] J. Shendure, and H. Ji, "Next-generation DNA sequencing," *Nat. Biotechnol.*, vol. 26, no. 10, pp. 1135-1145, 2008.

[2] S. C. Schuster, "Next-generation sequencing transforms today's biology," *Nat. Methods.*, vol. 5, no. 1, pp. 16-18, 2008.

[3] NIH National Human Genome Research Institute, "DNA Sequencing Costs." `http://www.genome.gov/sequencingcosts/`

[4] ENCODE Project Consortium, "An integrated encyclopedia of DNA elements in the human genome," *Nature*, vol. 489, no. 7414, pp. 57-74, 2012.

[5] Cancer Genome Atlas Research Network, J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, and J. M. Stuart, "The Cancer Genome Atlas pancancer analysis project," *Nat. Genet.*, vol. 45, no. 10, pp. 1113-1120, 2013.

[6] 1000 Genomes Project Consortium, G. R. Abecasis, D. Altshuler, A. Auton, L. D. Brooks, R. M. Durbin, R. A. Gibbs, M. E. Hurles, and G. A. McVean, "A map of human genome variation from population-scale sequencing," *Nature*, vol. 467, no. 7319, pp. 1061-1073, 2010.

[7] C. E. Romanoski, C. K. Glass, H. G. Stunnenberg, L. Wilson, and G. Almouzni, "Epigenomics: Roadmap for regulation," *Nature*, vol. 518, no. 7539, pp. 314-316, 2015.

[8] Jim Gray. Jim gray on escience: A transformed scientific method. *The fourth paradigm: Data-intensive scientific discovery*, pages xvii-xxxi, 2009.

[9] H. Gunadhi and A. Segev, "Query processing algorithms for temporal intersection joins," in *Proc. IEEE ICDE*, 1991, pp. 336-344.

[10] W.J. Kent, The human genome browser at UCSC. *Genome Res.*, 2002 Jun;12(6):996-1006.

[11] F. Afrati and J. Ullman, Bounds for Overlapping Interval Join of Map Reduce. *Workshop Proceedings, EDBT/ICDT*, 2015.

[12] Kaitoua Abdulrahman, 2017, "Scalable data management and processing for genomic computing", Politesi, `http://hdl.handle.net/10589/132065`

[13] Pinoli Pietro, 2017, "Modeling and querying genomic data", Politesi, `http://hdl.handle.net/10589/132099`

[14] Cattani Simone, 2016, "Genomic Computing with SciDB, a Data Management System for Scientific Applications", IEEE/ACM Trans Comput Biol Bioinform. 2016 , Politesi, `http://hdl.handle.net/10589/123101`

[15] Bertoni Michele, 2015, "Querying the DNA : genomic computing with Apache flink", Politesi, `http://hdl.handle.net/10589/112545`

[16] Ceri S, Kaitoua A, Masseroli M, Pinoli P, Venco F., "Data Management for Heterogeneous Genomic Datasets."

[17] Bio-Informatics Group, DEIB, Politecnico di Milano, "Specification of GMQL Version 2", `http://www.bioinformatics.deib.polimi.it/genomic_computing/GMQL/doc/GMQL_V2_manual.pdf`

[18] Marco Masseroli, Pietro Pinoli, Francesco Venco, Abdulrahman Kaitoua, Vahid Jalili, Fernando Palluzzi, Heiko Muller, and Stefano Ceri. Genometric query language: a novel approach to large-scale genomic data management. *Bioinformatics* 31(12):1881-1888,2015.

[19] A. Kaitoua, P. Pinoli, M. Bertoni and S. Ceri, "Framework for Supporting Genomic Operations," in IEEE Transactions on Computers, vol. 66, no. 3, pp. 443-457, March 1 2017.

[20] Masseroli M, Kaitoua A, Pinoli P, Ceri S., "Modeling and interoperability of heterogeneous genomic big data for integrative processing and querying." Methods, 2016.

[21] Ceri S, Kaitoua A, Masseroli M, Pinoli P, Venco F., "Data management for heterogeneous genomic datasets.", IEEE/ACM Transactions on Computational Biology and Bioinformatics 2016.

[22] Fernandez JD, Lenzerini M, Masseroli M, Venco F, Ceri S., "Ontology-based search of genomic metadata".

[23] Montanari P, Bartolini I, Ciaccia P, Patella M, Ceri S, Masseroli M., "IEEE Transactions on Knowledge and Data Engineering 2016"

[24] Bertoni M, Ceri S, Kaitoua A, Pinoli P., "Evaluating Cloud Frameworks on Genomic Applications."

[25] Cattani S, Ceri S, Kaitoua A, Pinoli P., "Bi-Dimensional Binning for Big Genomic Datasets." Proc. Beyond Map Reduce Workshop, co-located with ACM-Sigmod, May 2017; Boston.

[26] Kaitoua A, Gulino A, Masseroli M, Pinoli P, Ceri S, "Scalable Genomic Data Management System on the Cloud. Int. Symp. Big Data Principles, Architectures and Applications (BDAA), July 2017; Genova."