Universidade Federal de Goiás
Instituto de Informática

Thiago Borges de Oliveira

Efficient Processing of Multiway Spatial Join Queries in Distributed Systems

Goiânia, GO
2017

sibi sistema de bibliotecas ufg

PRPG
PRÓ-REITORIA DE PÓS-GRADUAÇÃO

UFG

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TESES E DISSERTAÇÕES NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou *download*, a título de divulgação da produção científica brasileira, a partir desta data.

1. Identificação do material bibliográfico: **[ ] Dissertação** **[X] Tese**

2. Identificação da Tese ou Dissertação:

Nome completo do autor: Thiago Borges de Oliveira

Título do trabalho: Efficient Processing of Multiway Spatial Join Queries in Distributed Systems

3. Informações de acesso ao documento:

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF da tese ou dissertação.

Assinatura do(a) autor(a)[2]

Ciente e de acordo:

Assinatura do(a) orientador(a)[2]

Data: 25/09/2017.

[1]Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.

Casos de embargo:
- Solicitação de registro de patente
- Submissão de artigo em revista científica
- Publicação como capítulo de livro
- Publicação da dissertação/tese em livro

[2]A assinatura deve ser escaneada.

Thiago Borges de Oliveira

Efficient Processing of Multiway Spatial Join Queries in Distributed Systems

Thesis presented to the postgraduate program of Instituto de Informática of Universidade Federal de Goiás, as a partial fulfillment of the requirements for the Ph.D. degree in Computer Science.

Concentration Area: Computer Science.

Advisor: Prof. Fábio Moreira Costa, Ph.D.

Co-Advisors: Prof. Leslie Richard Foulds, Ph.D.
Prof. Vagner José do S. Rodrigues, Ph.D.

Goiânia, GO
2017

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

de Oliveira, Thiago Borges
Efficient Processing of Multiway Spatial Join Queries in Distributed Systems [manuscrito] / Thiago Borges de Oliveira. - 2017.
156 f.: il.

Orientador: Prof. Dr. Fábio Moreira Costa; co-orientador Dr. Leslie Richard Foulds; co-orientador Dr. Vagner José do Sacramento Rodrigues.
Tese (Doutorado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação em rede (UFG/UFMS), Goiânia, 2017.
Bibliografia. Apêndice.
Inclui tabelas, algoritmos, lista de figuras, lista de tabelas.

1. Distributed Multiway Spatial Join. 2. Cost-based Optimizer. 3. Query Scheduling. 4. Histograms. I. Costa, Fábio Moreira, orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

UFMS

Ata de Defesa de Tese de Doutorado

Aos vinte e nove dias do mês de novembro de dois mil e dezessete, no horário das catorze horas, foi realizada, nas dependências do Instituto de Informática da UFG, a defesa pública da Tese de Doutorado do aluno Thiago Borges de Oliveira, matrícula no. 2013 0499, intitulada **"Efficient Processing of Multiway Spatial Join Queries in Distributed Systems"**.

A Banca Examinadora, constituída pelos professores:

Prof. Dr. Fábio Moreira Costa – INF/UFG - orientador

Prof. Dr. Leslie Richard Foulds – INF/UFG - coorientador

Prof. Dr. Vagner José do Sacramento Rodrigues – INF/UFG - coorientador

Prof. Dr. Kelly Rosa Braghetto – DCC/IME/USP

Prof. Dr. Claudio Nogueira de Meneses – CMCC/UFABC

Prof. Dr. Wellington Santos Martins – INF/UFG

Prof. Dr. Kleber Vieira Cardoso - INF/UFG

emitiu o resultado:

(X) Aprovado

( ) Aprovado com revisão

(A Banca Examinadora deve definir as exigências a serem cumpridas pelo aluno na revisão, ficando o orientador responsável pela verificação do cumprimento das mesmas.)

( ) Reprovado

com o seguinte parecer: _____

Prof. Dr. Fábio Moreira Costa

Prof. Dr. Leslie Richard Foulds

Prof. Dr. Vagner José do Sacramento Rodrigues

Profa. Dra. Kelly Rosa Braghetto

Prof. Dr. Claudio Nogueira de Meneses

Prof. Dr. Wellington Santos Martins

Prof. Dr. Kleber Vieira Cardoso

Thiago Borges de Oliveira

Received his master's in Computer Science with emphasis in Distributed Systems from UFG - Universidade Federal de Goiás in 2010. He is currently a professor in the computer science bachelor course at Universidade Federal de Goiás, Regional Jataí. He worked previously as a software engineer for several years in the agribusiness industry. His current research interests are the design and development of large-scale distributed systems for spatial data processing.

To Beatriz, my beloved daughter. I hope you always find a matter to wisely use your immense creativity and intelligence.

Acknowledgements

First and foremost, I would like to express my sincere thanks to my advisor, Prof. Fabio M. Costa, for all his guidance, support, trust along these years, and for the flexibility when I decided to move to Jataí.

I am deeply indebted to my co-advisor, Prof. Leslie R. Foulds, for all his guidance and patience in our meetings, especially when the idiom became a challenge from my side. Thank you for gently holding the throttle to wait for me.

I am especially grateful to my co-advisor Prof. Vagner José do Sacramento Rodrigues, for his support and guidance in the development of the main subject of this thesis.

My sincere thanks to Prof. Humberto José Longo, for helping and sharing his experience with the optimization methods, for the interest in the early drafts of the integer model, and for sharing them with Prof. Leslie. I'm proud of the development of this subject in this thesis, and his help was fundamental.

My sincere and special thanks to the thesis committee, Prof. Kelly R. Braghetto, Prof. Claudio N. de Meneses, Prof. Wellington Santos Martins, and Prof. Kleber Vieira Cardoso, for the invaluable suggestions and feedback.

I am fortunate to have the commitment and help of my wife, Cristiane de Sateles Valente, during all these years since I proposed myself this challenge. Without her attention and support with our family, this work would be indescribably challenging. Now we can enjoy the benefits together!

I am also grateful to my friends and work colleagues that supported and encouraged me in concluding this thesis. Thank you all for the support and for pushing me up.

I am also indebted to the two cloud providers that provided the necessary environment to perform the experiments presented in this work. Microsoft Azure supported this work through the Azure4Research program until 2015. Amazon supported the final round of experiments (2016-2017), providing the necessary computational environment through the AWS Cloud Credits for Research program.

I learned to walk; since then have I let myself run. I learned to fly; since then I do not need pushing in order to move from a spot.

Friedrich Nietzsche,
Thus Spoke Zarathustra, Part 1, Reading and Writing.

Abstract

de Oliveira, Thiago Borges. **Efficient Processing of Multiway Spatial Join Queries in Distributed Systems**. Goiânia, GO, 2017. 156p. PhD Thesis. Instituto de Informática, Universidade Federal de Goiás.

Multiway spatial join is an important type of query in spatial data processing, and its efficient execution is a requirement to move spatial data analysis to scalable platforms as has already happened with relational and unstructured data. In this thesis, we provide a set of comprehensive models and methods to efficiently execute multiway spatial join queries in distributed systems. We introduce a cost-based optimizer that is able to select a good execution plan for processing such queries in distributed systems taking into account: the partitioning of data based on the spatial attributes of datasets; the intra-operator level of parallelism, which enables high scalability; and the economy of cluster resources by appropriately scheduling the queries before execution. We propose a cost model based on relevant metadata about the spatial datasets and the data distribution, which identifies the pattern of costs incurred when processing a query in this environment. We formalized the distributed multiway spatial join plan scheduling problem as a bi-objective linear integer model, considering the minimization of both the makespan and the communication cost as objectives. Three methods are proposed to compute schedules based on this model that significantly reduce the resource consumption required to process a query. Although targeting multiway spatial join query scheduling, these methods can be applied to other kinds of problems in distributed systems, notably problems that require both the alignment of data partitions and the assignment of jobs to machines. Additionally, we propose a method to control the usage of resources and increase system throughput in the presence of constraints on the network or processing capacity. The proposed cost-based optimizer was able to select good execution plans for all queries in our experiments, using public datasets with a significant range of sizes and complex spatial objects. We also present an execution engine that is capable of performing the queries with near-linear scalability with respect to execution time.

Keywords

Distributed Multiway Spatial Join, Cost-based Optimizer, Job Scheduling, Histograms

Resumo

de Oliveira, Thiago Borges. **Processamento Eficiente de Consultas de Multi-Junção Espacial em Sistemas Distribuídos**. Goiânia, GO, 2017. 156p. Tese de Doutorado. Instituto de Informática, Universidade Federal de Goiás.

A multi-junção espacial é um tipo importante de consulta usada no processamento de dados espaciais e sua execução eficiente é um requisito para mover a análise de dados espaciais para plataformas escaláveis, assim como aconteceu com dados relacionais e não estruturados. Nesta tese, propomos um conjunto de modelos e métodos para executar eficientemente consultas de multi-junção espacial em sistemas distribuídos. Apresentamos um otimizador baseado em custos que seleciona um bom plano de execução levando em consideração: o particionamento de dados com base nos atributos espaciais dos datasets; o nível de paralelismo intra-operador que proporciona alta escalabilidade; e o escalonamento das consultas antes da execução que resulta em economia de recursos computacionais. Propomos um modelo de custo baseado em metadados dos datasets e da distribuição de dados, que identifica o padrão de custos incorridos no processamento de uma consulta neste ambiente. Formalizamos o problema de escalonamento de planos de execução da multi-junção espacial distribuída como um modelo linear inteiro bi-objetivo, que minimiza tanto o custo de processamento quanto o custo de comunicação. Propomos três métodos para gerar escalonamentos a partir deste modelo, os quais reduzem significativamente o consumo de recursos no processamento das consultas. Embora projetados para o escalonamento da multi-junção espacial, esses métodos podem também ser aplicados a outros tipos de problemas em sistemas distribuídos, que necessitam do alinhamento de partições de dados e da distribuição de tarefas a máquinas de forma balanceada. Além disso, propomos um método para controlar o uso de recursos e aumentar a vazão do sistema na presença de restrições nas capacidades da rede ou de processamento. O otimizador proposto foi capaz de selecionar bons planos de execução para todas as consultas em nossos experimentos, as quais usaram datasets públicos com uma variedade significativa de tamanhos e de objetos espaciais complexos. Apresentamos também uma máquina de execução, capaz de executar as consultas com escalabilidade próxima de linear em relação ao tempo de execução.

Palavras–chave

Multi-junção Espacial Distribuída, Otimizador Baseado em Custos, Escalonamento de Tarefas, Histogramas

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols[1]

$\bowtie$	Spatial join operator.
θ	Spatial predicate, such as intersects, distance.
d	Number of dimensions in a spatial dataset.
$\bar{\bar{a}}$, $\bar{\bar{b}}$	Cardinality of a, b.
l_{xk}	Average length of x in dimension k, when x is a histogram cell, or length of x in dimension k, otherwise.
f^q	A weight to specify the desired emphasis on a balanced execution while scheduling a query q, $0 \leq f^q \leq 1$.
$\bar{c}_{ib}$	Communication cost incurred if cell b is moved to machine i.
$\hat{c}_{ia}$	Communication cost incurred if cell a is moved to machine i.
w_j	Processing cost of job j.
σ^2	Standard deviation.
f	A weight to specify the desired emphasis on a balanced execution while scheduling a step of an execution plan for a multiway spatial join query.
m	Number of machines (or servers) used to schedule or execute a query.
$\boldsymbol{J}$	Set of jobs to be scheduled in a step of an execution plan.
n	Number of jobs in a step of an execution plan.
u_i	Residual load in machine i that reduces its processing capacity.
$\hat{y}_{ia}$	Boolean variable that indicates if the cell a is processed on machine i.
$\bar{y}_{ib}$	Boolean variable that indicates if the cell b is processed on machine i.
x_{ij}	Boolean variable that indicates if the job j is processed on machine i.
x_0	Variable that represents the completion time of the latest processed job by any machine, i.e., the makespan.
c_{ij}	Communication cost incurred when processing job j on machine i.
μ	Vector of Lagrangian multipliers, defined for each job j, μ_j.
f'	Upper bound on f for a given query scheduling instance.

[1] Global symbols used in the text, in order of their appearance.

CHAPTER 1

Introduction

The amount of spatial data has significantly increased with the popularization of GPS-enabled devices. Spatial data, such as geotagged images, IoT (Internet of Things) and Smart Cities sensor data, open data and census data, are continuously collected and organized in thematic datasets[1] to support decision-making and to improve market intelligence and logistics efficiency in both the private and public sectors.

Spatial data is a type of multidimensional data, a complex data type that is handled by Relational Database Management Systems (RDBMS) through the use of queries with spatial predicates [74]. An important type of query is the spatial join, which finds correlated objects in two or more datasets by applying some predicate like intersection or proximity [10]. Spatial join can be classified as "simple" when only two datasets are processed, or multiway ("complex") when processing more than two datasets in the same query [57].

Multiway spatial join queries are important in several application fields, including geography (e.g., to find animal species living in preservation areas that were damaged by fire), VLSI (e.g., to find circuits that formulate a particular topological configuration) [57] and digital pathology imaging [2] (e.g., to analyze topological images of the brain in order to check whether cancer is present).

The processing of multiway spatial join is usually significantly more complex than the processing of simple spatial join because a query can be executed in many different ways, called execution plans. An execution plan represents both the order in which datasets are combined and the algorithms that are used to find the final result of a query. The number of equivalent execution plans for a query is exponential in the number of datasets [57]. Further, a good execution plan for a query is, in general, orders of magnitude better than a bad plan, regarding its processing cost. Thus, a great amount of effort has been dedicated to proposing cost-based optimizers that can select a good execution plan for a query based on its estimated computational costs [58]. However, those efforts are mainly dedicated to non-distributed databases or distributed databases for non-spatial data. Spatial

[1] We use the word "dataset" in this thesis as a synonym for the term "data set".

data imposes particular challenges to join processing and, in general, algorithms designed for scalar data in relational databases do not apply to spatial data due to the absence of total ordering in multidimensional data [46].

Another particular challenge of spatial data analysis is the complexity of computational geometry algorithms – which evaluate the predicates over the spatial data. Even queries on small datasets have a high processing cost due to the complexity of predicate algorithms and the different types of spatial objects (e.g., points, lines, polygons). Processing spatial queries in distributed systems by partitioning the datasets using spatial columns has often been used to reduce the query execution time (or makespan) [46]. However, the skewed nature of spatial datasets imposes significant challenges to their partitioning using spatial columns, as skewness may cause unbalanced query execution.

In a distributed system, besides considering the local CPU and I/O costs, the selection of execution plans must take into account the approach used to perform the data partitioning and communication between the machines. A proper balance of the query has to be considered, choosing plans that divide the work evenly among machines, regarding both the bandwidth limit on the network interface and the load on the CPU. Furthermore, if we consider a query in a non-isolated scenario – i.e., as a complete solution for data processing of a distributed spatial database – other important issues arise. For example, the data must be loaded and partitioned before query execution. However, queries on different datasets have different data distribution, so, a fixed data distribution cannot be effective for all queries. The system needs to identify a schedule for query execution which adapts the data distribution and determines the machine that will process each data partition. For instance, in an analytics scenario, where a single query runs many times to discover correlations in the data, a little improvement in query scheduling can significantly reduce the execution time of long running jobs.

Selecting execution plans and identifying query schedules to efficiently process multiway spatial join queries in distributed systems are steps towards moving spatial data analysis to scalable platforms, as has already been done with relational and unstructured data. However, new methods and algorithms to estimate the cost of an execution plan need to be specified, taking into consideration the specifics of spatial data and the characteristics of distributed systems. Processing spatial data analysis in such environments can significantly improve the capabilities of spatial data processing, especially in today's scenario of cloud computing platforms, taking advantage of scalability, elasticity, and pay-as-you-go offers.

A large effort in the literature is dedicated to the distributed processing of spatial join queries (e.g., [14, 20, 46, 70]). However, the focus is on the handling of simple spatial join. Mamoulis and Papadias [57] studied the processing of multiway spatial join queries. Their work presents an in-depth study of algorithms to process spatial join, covering

techniques to enumerate the set of execution plans, to estimate their cost, and to select a good plan. However, their work applies only to sequential query processing on non-distributed systems.

Recently, various frameworks for distributed data processing have emerged, such as the MapReduce framework [24] and the Spark engine [92]. In these frameworks, data is split into partitions that are distributed to a number of commodity machines. The predicate of a query, i.e., an algorithm, processes local data on each machine and produces partial result sets that are later combined to produce final result. Techniques to process multiway spatial join in these frameworks were recently proposed (e.g., [3, 27]). However, these techniques did not consider the selection nor scheduling of execution plans, a well-established strategy used to process multiway queries in traditional database systems. In turn, the kind of parallelism implemented by these frameworks, known as intra-operator level of parallelism [65], is a fundamental design principle responsible for the high scalability achieved and, generally, is not implemented in traditional distributed database systems as they focuses mainly in intra-query parallelism [65]. Recently, the lack of attention to the database theory in the mentioned emerging frameworks was criticised (e.g., [71, 80]) and some surveys proposed the integration of query optimizers for plan selection as future work (e.g., [26]).

In this thesis, we investigated how to address the previously mentioned issues by proposing a cost-based query optimizer for distributed multiway spatial join at the intra-operator level of parallelism. We studied how to design and implement it by mixing a set of data and query processing phases, some of them mostly investigated in the parallel and distributed systems literature, e.g., data partitioning, data distribution, and load-balancing mechanisms, and others mostly studied in database literature, e.g., query planning that is comprised of execution plan selection, plan enumeration, and plan costs estimation. In the next section, we present the scope of this thesis by illustrating this set of phases.

1.1 Thesis Scope

When considering a cost-based query optimizer at the intra-operator level of parallelism, the efficient processing of multiway spatial join queries in distributed systems can be addressed by a set of phases as illustrated in Figure 1.1. The cost model is a central component in this design and supports the cost estimation in all query planning phases. It is designed for the distributed environment, considering the computational costs of the algorithms used to process spatial data. The first two phases, data partitioning and data distribution, split the data into partitions and distribute them into the system, capturing the metadata required by the cost model. The discontinuity shown in the figure between data distribution and plan enumeration implies that the first two phases can occur irrespective

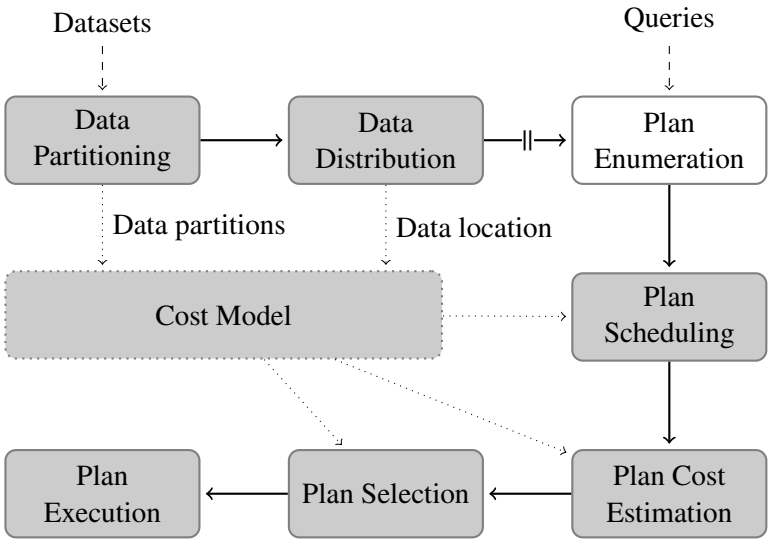


Figure 1.1: *Distributed query processing phases and their interaction with the cost model. Bold arrows indicate the flow of the query processing. Dashed arrows indicate input and dotted arrows the interactions between the phases. The shaded boxes indicate the scope of this thesis.*

of the next ones. After the data is loaded into the system, the submission of queries starts the next phases (query planning and execution), as described next.

The first phase of query planning is plan enumeration (Figure 1.1), which identifies the alternative execution plans for query execution. Each alternative plan has an associated computational cost that needs to be determined by the query scheduling and plan cost estimation phases. Query scheduling determines where (i.e., in which machine) each data partition will be processed. Based on the computed schedule, it is possible to determine the computational costs of a query in the plan cost estimation phase. After computing the cost of the alternative plans, the plan selection phase selects the best plan to execute the query in the plan execution phase.

In this thesis, we present models and methods for each of the phases illustrated in Figure 1.1 except for plan enumeration. The reason for plan enumeration being outside our scope is that the work of Mamoulis and Papadias [57] presents a comprehensive algorithm for plan enumeration, which was explicitly designed for spatial data. Although it was designed for non-distributed query processing, the distributed environment does not require significant changes. In the next section we detail the primary contributions of this thesis.

1.2 Primary Contributions

The contribution of our work lies in providing a set of comprehensive models and methods that form a cost-based optimizer for multiway spatial join queries. The optimizer is able to select a good execution plan for distributed processing, taking into account *i*) the partitioning of data based on spatial attributes of datasets, *ii*) the intra-operator level of parallelism, which enables high scalability, and *iii*) the economy of cluster resources by appropriately scheduling the query before execution.

The major contributions of this thesis are:

- Identification of the characteristics of spatial datasets and data distribution that are relevant for the efficient processing of multiway spatial join queries in distributed systems;
- Definition of a multidimensional histogram data structure to organize these characteristics, observing the data skewness of real spatial datasets; this data structure is also used as a distributed hash index to access data partitions;
- Introduction of a cost model based on the multidimensional histogram and associated costs of the distributed environment, which improves the accuracy of query cost estimates; although targeting an estimate of the cost of multiway spatial join queries, the cost model can also be used to estimate the cost of window and simple spatial join queries;
- Formalization of the distributed multiway spatial join plan scheduling problem as a multi-objective linear model, considering the minimization of both the makespan and the communication cost as objectives, and the introduction of three methods to compute schedules based on this model: a greedy algorithm (GR), and two algorithms based on combinatorial methods: the well-known Linear Relaxation [85] (LP) method and the more sophisticated Lagrangian Relaxation [32] (LR) method; although targeting multiway spatial join query scheduling, these methods can be applied to other distributed systems as well, which require both data partitions alignment and assignment of jobs to machines;
- Investigation and introduction of methods to control the resource consumption of query schedules regarding the trade-off between a well-balanced query execution that minimizes the query execution time, and a somewhat unbalanced execution that minimizes the incurred communication cost due to the movement of data partitions; and
- Introduction of a method to select good execution plans to process multiway spatial join queries in distributed systems.

1.3 Thesis Outline

The remainder of this thesis is organized as follows.

Chapter 2 provides background concepts about spatial data and spatial query processing, focusing on spatial join and multiway spatial join queries. We present an overview of the steps involved in the planning of multiway spatial join queries and highlight the additional challenges that a distributed environment imposes on them.

Chapter 3 proposes and evaluates a model to estimate the cost of distributed multiway spatial join queries. We show how to build intermediate histograms using the predicted intermediate results and how to use them to calculate the cost of a multiway spatial join. Furthermore, we present a set of new statistical formulae that provide more accurate estimation of join selectivity, i.e., the size of intermediate join results, for real spatial datasets, involving complex lines and polygons. This chapter includes and revises material from previously published work [21, 22].

In Chapter 4, we consider the scheduling of query plans, i.e., the problem of assigning jobs, which are defined by pairs of data partitions from two datasets and aligned by a spatial predicate, to a set of machines in a distributed system. We propose and evaluate a multi-objective linear integer model for this problem and propose methods to compute approximate solutions for it.

In Chapter 5, we study and propose a method to adapt the scheduling behavior concerning the use of computing resources. We show that attempting to minimize both CPU and network consumption when executing distributed queries creates conflicting objectives, in the sense that to achieve a better balance in the query execution (and consequently a reduced makespan) an extra cost is incurred to transfer partitions to machines that are underloaded. The method is based on Parametric Analysis (PA), which is a type of post-optimality analysis of integer linear programs [36].

Chapter 6 introduces some methods for the final query optimizer proposed in this thesis and presents a query execution engine that can execute multiway spatial join queries following the schedules provided by the optimizer. This chapter revises material from [22].

Chapter 7 presents a literature survey on the processing of multiway spatial join queries and compares the reported solutions with the corresponding ones proposed in this thesis.

Finally, we present our conclusions in Chapter 8, reflecting on our contributions, the limitations of our approach, and opportunities for future work.

There are also two appendices: Appendix A, with the complete experiment results and parameters used in the evaluation of the schedule methods in Chapter 4 and Appendix B, which describes a complete example of the application of the method proposed in Chapter 5.

CHAPTER 2

Multiway Spatial Join Processing

In this chapter, we provide background concepts about spatial data and spatial queries processing, focusing on spatial join and multiway spatial join queries. We present an overview of the steps involved in the planning of multiway spatial join queries and highlight the additional challenges that a distributed environment imposes on them.

We first explain the nature of spatial data (Section 2.1) and then present the main types of spatial queries (Section 2.2). Section 2.3 covers the simple spatial join query, one of the most-used spatial queries, and an algorithm used to process it in distributed systems, known as Clone Join.

Section 2.4 presents the definition of multiway spatial join queries, which is the focus of this thesis. We review the steps involved in planning and executing general distributed multiway queries: plan enumeration (Section 2.4.1), plan cost estimation (Section 2.4.2), plan scheduling (Section 2.4.3), plan selection (Section 2.4.4), and plan execution (Section 2.4.5). In each of these sections, we discuss the challenges imposed by spatial data and by the distributed environment.

2.1 Spatial Data

Spatial data is data about positions, attributes, and relations of entities in space [59]. It is a specialized data type, handled with particular techniques by *spatial databases* [74]. The term is often used as a synonym for *geographical data* or *geospatial data*, which refers to data about objects on the surface or near-surface of the Earth [77].

In this thesis, we use the term spatial data, instead of geospatial data, to make explicit that all methods proposed here can be applied irrespective to the spatial reference. For instance, the spatial reference can be another planet surface, the cosmos, the place of electronic components on a circuit board, or data about the human body captured by medical images. In each of these spatial references, a particular projection is used to flatten the data onto a planar representation. This projection is called a *spatial reference system*.

A *spatial object* is a representation of an entity that is present in space [74], composed of alphanumeric attributes that describe the entity itself, and by a spatial

component that describes the geometry (location and shape) and sometimes the topology (spatial relationships, such as overlapping, adjacency).

A *layer* or *dataset* is a set of spatial objects that are of the same type, i.e., have the same structure [74]. For example, a river dataset composed of a set of spatial objects that represent water courses in a particular region. A spatial object represents each river and has a set of descriptive alphanumeric attributes such as name, length, and topographical classification, and a spatial attribute that embodies its geometric aspects.

A *spatial data model* is used to represent the spatial attribute of a spatial object. A spatial data model is a set of rules to describe and represent aspects of real spatial entities [12]. There are two major spatial data models: the raster data model and the vector data model. Both models have advantages and disadvantages. Which one to use depends upon the purpose of the application.

The *raster data model* divides the space into cells arranged in rows and columns of equal size. These cells annotate spatial properties or attributes, similar to what a digital image does with visual information [54].

The *vector data model* uses a set of points in a coordinate system as vertices of a spatial entity contour. It provides a high precision representation of spatial objects and has inherent topology, two characteristics that simplify spatial analysis [12].

The vector data model has three fundamental data types:

1. *point*, a zero-dimensional object with a single coordinate pair, used to represent the location of something in space;
2. *line*, a one-dimensional object with a set of interconnected points, used to represent roads, rivers, boundaries, and so forth; and
3. *polygon*, a two-dimensional object with one or more lines arranged in a closed loop, with the first point being equal to the last one. Polygons are used to represent objects with areas, such as city contours, and lakes [12]. When considering volume or time, a third and fourth dimension can be introduced in this model [74].

The task of storing and querying a set of spatial datasets is performed by specialized Database Management Systems (DBMS) [65]. A *Spatial Database Management System* (SDBMS) provides comprehensive technology to represent spatial objects, access methods for fast retrieval, query languages, and algorithms from related disciplines such as computational geometry [74]. Due to the high volume and the complexity of spatial data, a Distributed Database Management Systems (DDBMS) [65] specialized with spatial data capabilities should be used to perform spatial queries fastly.

Spatial data and SDBMS is a broad field of study. In this thesis, we are particularly interested in spatial data represented in the vector data model, and we focus on distributed processing of multiway spatial join queries. The following sections describe spatial analysis and spatial queries, and introduce the spatial join query.

2.2 Spatial Analysis

Spatial Analysis is the manipulation of spatial data for the purpose of adding value to it, supporting decisions, or even revealing information that is not immediately apparent. In other words, spatial analysis is the process of transforming raw spatial data into useful spatial information [54].

Spatial Queries are the fundamental operations employed in spatial analysis. They enable an in-depth study of topological and geometric attributes of spatial objects from a single or multiple datasets [12].

Three important types of spatial queries are:

1. *Window query*, a geometric selection applied to a single dataset. The result of this query is a set of objects that overlap with a given region or *window*, usually rectangular;
2. *Spatial Join*, an overlay operation, applied to two datasets. In this query, a spatial object from one dataset is *joined* with an object of the other dataset if their geometries satisfy a spatial predicate, such as intersection or coverage; and
3. *Multiway Spatial Join*, a spatial join that involves an arbitrary number of datasets, each pair of datasets having a particular spatial predicate.

As they are of fundamental importance in this thesis, spatial join and multiway spatial join queries are discussed in more detail in the following sections.

2.3 Spatial Join

A spatial join operation consists of a combination of objects from two spatial datasets in pairs that satisfy some spatial predicate, θ, such as intersection or coverage. The result of a spatial join of datasets A and B, denoted as $A \bowtie B$, consists of all pairs of objects $\{a, b\}$, $a \in A$ and $b \in B$, which fulfill $a\ \theta\ b$ [10].

Consider two spatial datasets with a spatial attribute representing city limits and rivers of a country. A spatial join can identify all combinations of rivers and cities that intersect with each other, applying an algorithm for predicate-checking to verify intersection between each pair of spatial attributes (e.g., comparing the set of points and line segments that define each river and city object).

A spatial join is processed in two distinct steps: (*i*) *filtering step* and (*ii*) *refinement step* [46]. In the filtering step, candidate pairs are identified using simplified polygons, such as the *minimum bounding rectangle* (MBR), an approximation of the original polygon formed by a bounding rectangle with extreme corners defined by the minimum and maximum coordinates in each dimension of the original object. Due to the use of

simplified polygons, these candidate pairs can be false hits. The refinement step applies a geometry algorithm to the filtered pairs to check the predicate θ over the real objects, in order to identify the correct results of the operation.

Performing spatial joins using indexes like R-Trees or another recursive index can reduce the join execution time [9]. For datasets with specific characteristics, like low cardinality or high selectivity, however, some hash-based algorithms can perform better [53]. The next section discusses the issues involved in the distributed processing of spatial joins.

2.3.1 Distributed Processing of Spatial Join

As the availability of large spatial datasets has recently increased, many algorithms to process spatial join queries in parallel or distributed computational environments have been proposed. Most of these algorithms use a data partitioning (declustering) strategy to split dataset objects into groups, called *data partitions* or *cells*. Prior to or during the join execution, a routine assigns a set of partitions to a particular server or processor that will perform the query on the set. There are two principal categories of methods for spatial data partitioning:

1. Disjoint space partitioning – uses a grid of disjoint cells to divide the space extent of the dataset. Each cell of the grid groups the spatial objects according to their intersection with the cells. This partitioning replicates objects that intersect at more than one cell; and
2. Non-disjoint space partitioning – partitions can overlap each other to accommodate the extent of the objects that intersect them and does not require object replication. An example of this type of partitioning is the set of MBRs on a given level of an R-Tree index.

The type of partitioning guides the development of a spatial join algorithm. For non-disjoint space partitioning, some associated distributed algorithms are: *Replicated Semi-packed Parallel R-Tree* (RSPR) [61], *Distributed Synchronous Traversal* (DST) [18], and *Proximity Area Spatial Join* (PASJ) [20]. All of them use distributed R-Tree indices to process the spatial join. For disjoint space partitioning, some associated algorithms are: *Clone Join* (CJ) [70], *Shadow Join* (SJ) [70], and *Non-blocking Parallel Spatial Join* (NPSJ) [62].

As shown by Patel and DeWitt [70], the non-disjoint division of the space of an R-Tree causes significant communication overhead during join filtering and refinement. For this reason, in this thesis, we consider only algorithms designed for disjoint space partitioning.

CJ is the simplest algorithm that uses disjoint space partitioning. It uses a fixed grid to partition datasets and replicate spatial objects that intersect more than one grid cell. The difference between CJ and SJ is that the latter uses a more sophisticated technique to reduce object replication. The NPSJ algorithm also uses a fixed grid and replicates objects as does CJ. However, it is a non-blocking algorithm that produces results early in the execution. Another difference between NPSJ and the others is that it creates local R-Trees on each data partition and identifies the results using an *R-Tree Join* (RJ) [9], while CJ and SJ use *Partition-based Spatial-Merge Join* (PBSM) [69].

Another issue in distributed spatial join processing is the distribution of the data partitions. The simplest and most-used distribution method is round-robin. It distributes the partitions evenly between the servers, promoting load balance in the cluster. Other methods that favor the co-location of spatial data have also been proposed, such as the Proximity Area method [20] for non-disjoint partitioning, which distributes the partitions based on their location, favoring reduction of network bandwidth usage during spatial join execution.

As the co-location of spatial extent is not considered in a round-robin data distribution, often data partitions from two distinct datasets with overlapping geographic region will be assigned to distinct servers in the cluster. The join algorithm thus needs to replicate the overlapping partitions on a particular server in order to execute the join. Although the round-robin distribution can be thought as the worst distribution to use, there seems to be a trade-off between data assignment and load balancing that favors a round-robin distribution on faster local networks. As discussed by de Oliveira et al. [20], distributing the partitions based on their location will reduce network bandwidth usage at the expense of compromising the load balance of spatial join execution.

Following the results reported in [70], in this thesis we use a disjoint space partitioning. Each grid cell was used to group and assign the spatial objects to form data partitions. For join execution, we used the CJ algorithm with the duplication avoidance technique, based on the Reference Point Method. The CJ algorithm is simpler than NPSJ and duplication avoidance also transforms it into a non-blocking algorithm. The next section covers the CJ algorithm and the Reference Point Method in more detail.

2.3.2 Clone Join

Clone Join (CJ) [70] is a distributed spatial join algorithm for joining two non-indexed datasets. The data partitioning used by the algorithm is a grid of disjoint cells, each cell representing a small part of the spatial extent. The algorithm assigns spatial objects to each cell based on the intersection between their MBRs and the cell boundaries, and replicates objects that intersect more than one cell. A set of cells and the corresponding

spatial objects produce a data partition that is assigned to one server using a round-robin distribution or a similar hash function applied to the cell number. In each server, the join operation is performed in parallel, using a local *Partition-based Spatial-Merge Join* (PBSM) [69], which is a spatial join algorithm for shared-memory parallel systems.

Figure 2.1 illustrates a data partition for an example dataset. The spatial objects (gray rectangles in the figure) are positioned on a grid of sixteen cells. The cells are distributed among four servers indicated by the number on each cell. Objects that intersect more than one cell are cloned for each of them.

Patel and DeWitt [70] assume that both datasets are partitioned and distributed using the same grid, at the beginning of the join execution. Although this is an interesting scenario for query cost estimation, due to the absence of partial cell overlapping, this assumption is not valid for a database system. In a database, we do not know which datasets will be joined until a query is submitted to the system. Thus, when loading the data into the system, we can not choose what grid to use for which dataset based on future joins. Furthermore, it seems that for a more precise query costs estimation, each dataset requires a particular partitioning scheme based on its geographic extent and spatial objects sizes. We will investigate this issue in Chapter 3.

A more realistic scenario would be to previously load the datasets in the database and assign a customized grid to each one, considering its spatial extent and the distribution and size of its spatial objects. During query evaluation, the pairs of cells from each dataset are identified using the system catalog (metadata) and the join algorithm copies and assigns the necessary partitions to the servers to perform the join.

Due to object replication, an additional step is required to eliminate duplicate results generated at the refinement step. The algorithm reports a result more than once whenever two spatial objects that intersect each other are present in two or more cells that were previously assigned to different servers. Each server will individually report the

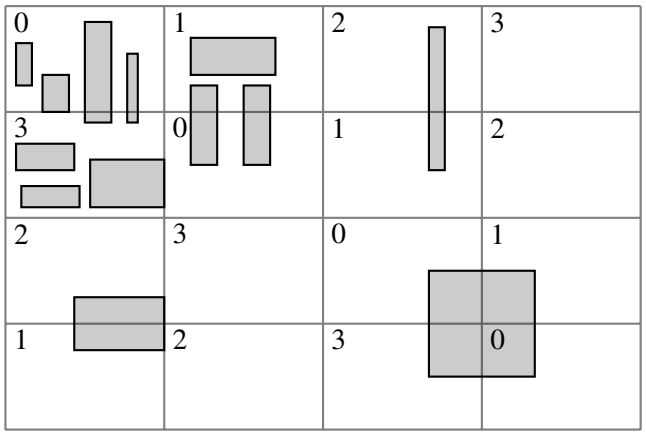


Figure 2.1: *Illustration of the Clone Join Partitioning Method. Adapted from [70].*

intersection between the objects. Patel and DeWitt [70] utilized a distinct operator at the end of the algorithm to eliminate duplicate results. However, this is a costly operator in distributed systems due to its network-bound behavior. Objects with large spatial extents (such as lines and polygons) increase the number of replicated objects. Consequently, the consumption of resources (notably, bandwidth and execution time) increases and partitioning needs to be defined according to object replication.

2.3.3 Reference Point Method

Dittrich and Seeger [25] proposed a solution for the result duplication problem called the Reference Point Method (RPM). The method consists of identifying the possible cells that will report duplicate results and allowing only one of them to do so.

Figure 2.2 shows an illustration of the method. There are four partitions: $P_0, \ldots, P_3$, and two spatial objects, r and s. The two objects are from distinct datasets R and S. The four partitions have all objects replicated in them, due to their spatial extent. If each partition ends up assigned to a different server in the cluster, each server will report the pair (r, s) as a join result. The reference point in the figure, rp, can be obtained by the equation $rp = (max(r.xl, s.xl), min(r.yh, s.yh))$, where xl is the leftmost x coordinate and yh is the highest y coordinate [25]. Although not mentioned by Dittrich and Seeger [25], the case for which rp falls exactly at the intersection of the four regions, or indeed, in any of the segments that divide any two regions, can be handled by considering these coordinates to pertain to the rightmost bottom partition. Thus, as the reference point can overlap only one partition, this fact is used to specify which server will report the result.

Patel and DeWitt [69] proposed the RPM originally for the PBSM algorithm, a parallel spatial join algorithm for shared memory parallel systems. Naughton and Ellmann [62] later adapted it to the NPSJ algorithm, a spatial join algorithm for distributed systems. Similarly, we can apply the reference point method as a verification subroutine in the filtering step of CJ.

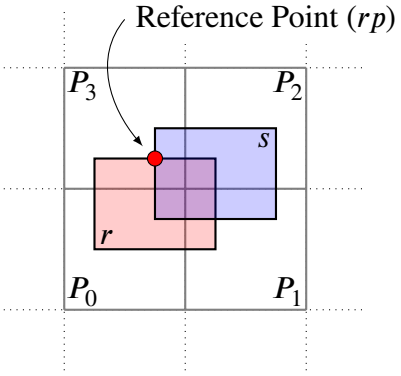


Figure 2.2: *Illustration of the Reference Point Method with two rectangular objects, r and s.*

2.4 Multiway Spatial Join

Multiway spatial join is a spatial join query with an arbitrary number n of input datasets, $n > 2$ [68]. An example of multiway spatial join query is: "Find all animal species living in preservation areas that were damaged by fire at a riverside." Four spatial datasets: animals, preservation areas, fire propagation and rivers need to be combined to compute the query results.

A multiway spatial join can be represented as a graph $G = (V, E)$ with node set V and edge set E, where each node represents a distinct dataset, and the edges represent the join predicates [57, 68]. Figure 2.3 illustrates three examples of query graphs. Each subfigure represents a different type of query, classified according to the characteristics of the graph. Figure 2.3a shows a tree or chain query, a common type of query on relational spatial data processing that has all datasets combined in pairs, without repetition. Figure 2.3b shows a cycle query, in which there is a cycle in the graph. Finally, Figure 2.3c shows a case where all datasets need to be verified against each other to check the predicate.

Each query in Figure 2.3 can be divided into steps and processed in different orders. Mamoulis and Papadias [57] investigated the number of ways a query can be processed for serial processing (non-parallel, non-distributed), and showed that it is a function of the query type, the number of input datasets, and the number of different join algorithms that can be used at each step. For example, there are almost 100 combinations for five input datasets with a clique query graph, considering three join algorithms [57]. Each of these combinations is a different execution plan for a query.

Figure 2.4 illustrates some alternative plans to process a chain query graph (2.4a). In Figure 2.4b, the datasets are pairwise joined in a first step, producing two intermediate results, which are joined by the second step. In Figure 2.4c three datasets are joined in a single step and the second step joins the generated intermediate result with the animals dataset. In Figure 2.4d , the first step joins fire and riversides, and the subsequent steps join intermediate results with the other two datasets, one at a time.

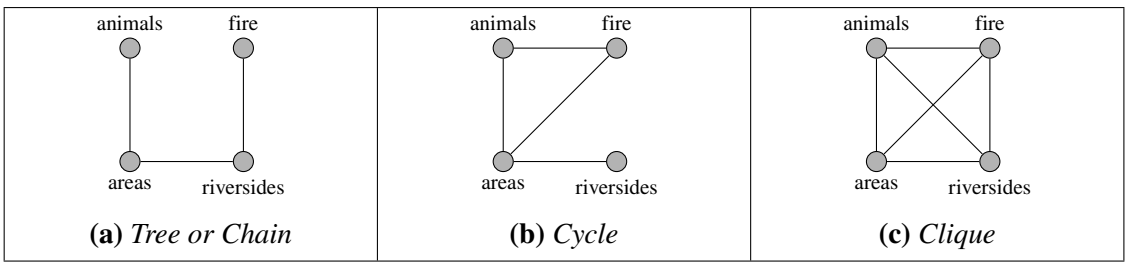
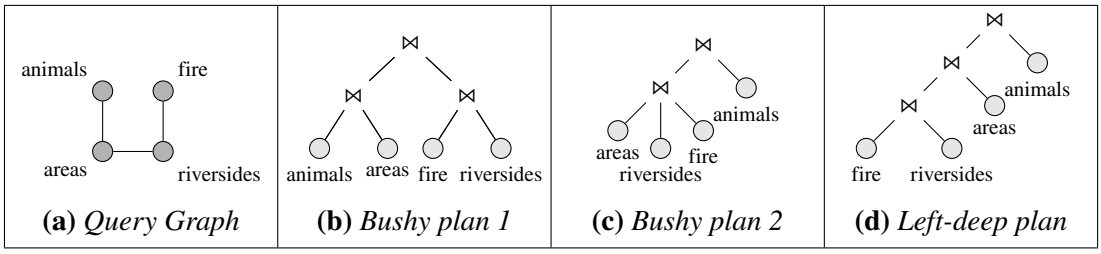


Figure 2.3: *Multiway spatial join query types: a) fires that intersect riversides, but not animals nor areas, b) fires that intersect preservation areas and animals, and c) all items that intersect with each other.*



(a) *Query Graph* **(b)** *Bushy plan 1* **(c)** *Bushy plan 2* **(d)** *Left-deep plan*

Figure 2.4: *Alternate execution plans to process a multiway spatial join query.*

All the execution plans for a given query preserve the same query semantics but may take different amounts of time to produce the final result. A query optimizer, or plan selection algorithm, uses as input the graph that represents a query and, after enumerating the possible plans, selects one of them to perform the query. The optimizer considers some aspects of the datasets and the associated costs of the algorithms to select an execution plan that determines: *i*) how the datasets will be combined, *ii*) the processing order of the datasets, and *iii*) which algorithms to use in each step.

Despite the name, a query optimizer does not search for the optimal plan to perform a query, since the optimal execution plan is difficult to find with a large number of datasets [44]. In general, the optimizer uses heuristic algorithms to choose a plan that has a reasonable execution time.

A query optimizer can be of two types: a rule-based optimizer (RBO) and a cost-based optimizer (CBO). The rule-based optimizer uses fixed rules to select a good execution plan [33]. Rule-based optimizers are simpler to construct and have the advantage of quickly selecting an execution plan, but are less flexible concerning the addition of new join algorithms and have limited sensibility to dataset properties not considered in their design.

In contrast, a cost-based optimizer estimates a cost for each plan based on dataset properties, as well as the I/O and CPU costs of the join algorithms. This type of optimizer is more adaptive to dataset properties, and a join algorithm can be easily integrated as long as it provides a cost estimation function. However, the optimizer needs to estimate the cost of many plans. It also needs to collect metadata from the dataset, a priori. For instance, the optimizer proposed by Fornari et al. [34] and Mamoulis and Papadias [57] are examples of this type of optimizer.

2.4.1 Plan Enumeration

In multiway spatial join queries, the space of possible plans to select from is nonlinear with respect to the number of datasets. Mamoulis and Papadias [57] studied the number of possible plans considering three different algorithms to process spatial join queries in a non-distributed system. The recurrence in (2-1) gives the number of plans,

$P(n)$, for a chain query with n datasets [57]. In asymptotic terms, $P(n) = \Omega(2^n)$. Cycle and clique queries have even more execution plans. As a result, if a query involves a sufficiently large number of datasets, a query optimizer can take more time planning – by having to enumerate all possible plans to estimate their costs – than the execution engine will take by executing the query. For instance, a chain query with ten datasets results in 8,944 plans to evaluate, according to the equation.

$$\begin{aligned} P(2) &= 1 \\ P(n) &= 1 + 2P(n-1) + \sum_{2 \leq k < n-1} P(k)P(n-k) \end{aligned} \tag{2-1}$$

To quickly identify good execution plans while searching only a small fraction of the search space, Mamoulis and Papadias [57] proposed a heuristic that randomly transforms a seed plan using a set of pre-defined rules, such as associativity, commutativity, and others specific to spatial data predicates. This transformation was applied as a subroutine within iterative improvement and simulated annealing methods and was able to find plans only slightly more expensive than an optimal execution plan found by an exhaustive method. The proposed algorithm is a good strategy for queries with a large number of datasets and can be integrated into a cost-based optimizer for multiway spatial joins.

The enumeration of plans is a required component of a cost-based query optimizer, whose input is the query to be processed, and the output is a set of candidate plans. This process has been extensively studied in the context of relational databases [45]. Mamoulis and Papadias [57] discussed how to adapt it for multiway spatial join queries.

The investigation of a plan enumeration algorithm is outside the scope of this thesis because the distributed environment does not impose significant modifications on it. Indeed, the evaluation of our proposed methods benefits from using an exhaustive algorithm, to compare the costs and the selection of a good plan over the entire set of plans. For practical applications, where the exhaustive search is prohibitively expensive, the non-distributed approach proposed by Mamoulis and Papadias [57] can be used without modification.

2.4.2 Estimating the Cost of Execution Plans

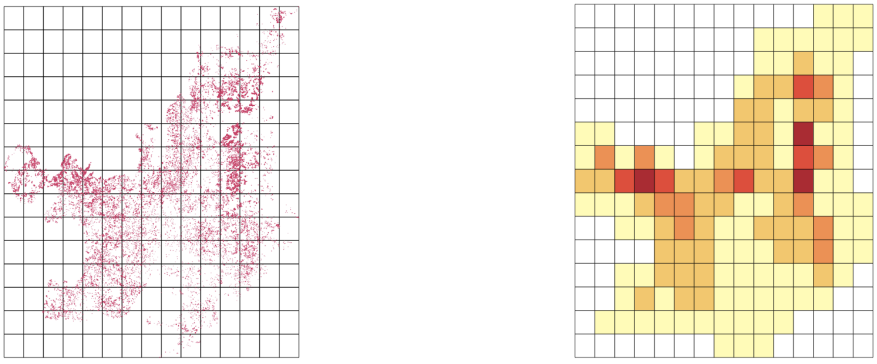
Despite the complexity of estimating the cost of execution plans for spatial data, some authors have proposed expressions that predict the cost of spatial join queries, such as [34, 75, 79], as well as methods to combine them to predict the cost of multiway spatial join queries [56]. Such a set of equations and methods is called a *cost model* in the database literature.

Those equations predict the cost of an execution plan by calculating the I/O and CPU costs of the join algorithm, assuming that spatial objects fill the spatial extent uniformly. The same authors later studied the difficulty of using them on real datasets [57], and conclude that, when used on real spatial datasets, the equations can induce bad execution plans, especially in the presence of dataset skewness.

Mamoulis and Papadias [56] studied the applicability of the equations in small regions of the dataset. They propose the use of a uniform histogram that divides the spatial extent of the dataset in a map with fixed-size cells, each of them mapping the density of small regions, as illustrated in Figure 2.5b. In Figure 2.5b, a grid of 225 cells (15×15) divides the spatial dataset of fire warnings for the Brazilian cerrado biome (Figure 2.5a).

The main advantages of uniform histograms are *i*) simplified construction, *ii*) time efficiency to estimate queries, and *iii*) incremental maintenance for non-static datasets [16]. Furthermore, grid histograms provide a natural way of partitioning spatial data, a required step in distributed systems processing. A deficiency of uniform histograms is the large estimation error for skewed data. Although increasing the number of cells reduces the error, it also increases the amount of memory needed to store the histogram structure [16]. There are other types of histograms for spatial data, such as [1, 73, 75], that can be used to improve the estimation of skewed datasets. However, estimating query costs with those histograms is more complex, and their recursive nature creates significant drawbacks for incremental maintenance and data partitioning.

A major challenge when building multidimensional histograms for spatial data is how to account for spatial objects in the histogram cells. We refer to this process as *hashing*. Mamoulis and Papadias [56] defined the cardinality value of each cell based on the number of spatial objects that have the center of their minimum bounding rectangle (MBR) within the cell limits. Although this method is accurate for point data, other types of spatial objects, such as lines and polygons, have a spatial extent that brings additional



(a) **(b)**

Figure 2.5: *Fire warnings dataset for the Brazilian cerrado biome, splitted by a histogram grid (a) and the* 15×15 *histogram cells distinguished by the density of objects (b).*

challenges to histogram building. A single line or polygon object can overlap more than one histogram cell, and the use of the MBR center accounts the object to only one histogram cell, leading to errors when estimating the query costs.

The definition of the number of cells, or grid resolution, is another issue in grid histograms. In general, a higher number of cells (or buckets, for non-grid histograms) can provide additional precision in cost estimation. However, Mamoulis and Papadias [57] identified that increasing the number of cells, besides reducing the cell size, introduces boundary errors due to the increase in the number of objects that intersect more than one cell. How to define the grid resolution, based on dataset metadata is still an open question for spatial data. Furthermore, in distributed systems, when we use the standard approach of transforming each cell in a data partition, a small number of partitions can interfere with the load balance. A higher number of data partitions can be used as a strategy to reduce the size of data partitions and the effect of data skewness in the load balance. Moreover, a small number of data partitions can decrease the degree of parallelism of query execution, affecting system scalability.

To further improve the precision in cost estimation with grid histograms, Mamoulis and Papadias [56] proposed the use of the average length of objects as an additional metadata in each histogram cell. This metadata is defined for each dimension based on the average of the lengths of the MBR of all spatial objects hashed in a given histogram cell. For objects that overlap more than one histogram cell, only the area inside the cell boundaries is considered. The restated equations (2-2), originally presented in [82], and (2-3), originally presented in [83] (both improved in [56]), show how to use the average length to determine the output cardinality of windows ($O^{\bar{w}}$) and join queries (O^{j}) as explained in the following. We use these equations in the development of the cost model in this thesis.

Given a cell a from histogram H_A, the output cardinality $O^{\bar{w}}$ of a window query $\bar{w}$ can be estimated by (2-2), where $\bar{\bar{a}}$ is the cardinality of a, d is the number of data dimensions, l_{ak} represents the average length of the set of objects in a in dimension k, $l_{\bar{w}k}$ is the length of $\bar{w}$ in dimension k, and l_{uk} is the length of a in dimension k, $l_{uk} \neq 0$. The rationale behind this equation is that the probability of intersection between a random MBR of an object from a with the query $\bar{w}$ at some dimension k equals the sum of projections l_{ak} and $l_{\bar{w}k}$ on that dimension normalized to the workspace (l_{uk}). The product in the equation is due to the intersection predicate itself, i.e., all dimensions must intersect simultaneously.

$$O^{\bar{w}}(a, \bar{w}) = \bar{\bar{a}} \cdot \prod_{k=1}^{d} min\left(1, \frac{l_{ak} + l_{\bar{w}k}}{l_{uk}}\right) \tag{2-2}$$

For spatial joins, given a pair of overlapping cells $\{a, b\}$, from histograms H_A and H_B, generated from datasets A and B, the output cardinality of the spatial join O^j for the cell pair can be estimated by (2-3), where i is the intersection of the MBRs of a and b, l_{ik} is the length of i in dimension k, $l_{ik} \neq 0$, and the other terms are defined according to the previous equation (2-2). This is an extension of (2-2), where the cardinality of the two datasets are limited by their common area (i) and the expected number of intersections is reduced by applying window queries of size l_{bk} on a. The workspace, used to normalize the lengths, is also limited by the common area, i.e., determined by the length of i (l_{ik}).

$$O^j(a,b) = O^{\bar{w}}(a,i) \cdot O^{\bar{w}}(b,i) \cdot \prod_{k=1}^{d} min\left(1, \frac{l_{ak} + l_{bk}}{l_{ik}}\right) \tag{2-3}$$

Mamoulis and Papadias [57] showed that by applying (2-3) to each pair of cells that obeys the join predicate in a plan step leads to more precise estimates of the output cardinality for join queries. Furthermore, by looping through all pairs of intersecting cells from H_A and H_B and applying (2-3), we can build an intermediate histogram H_R with the resulting J_c for each pair. Therefore, the estimation (to be used later in the plan) can use this intermediate histogram as input.

The plan format has an impact on the precision of the estimated cost of a query due to error propagation across the plan steps. Left-deep plans have a histogram created from a dataset at each step that contributes positively to error reduction. On the contrary, bushy plans have steps composed of two intermediate results, both being estimated histograms, which may contribute to error propagation.

Although left-deep plans do not exhibit a high level of parallelism [65], due to the dependency between two plan steps in sequence, a more aggressive data partitioning helps to solve this problem, while also increasing histogram precision. For instance, by creating a data partition for each histogram cell, one can take each data partition as a query fragment that can be performed in parallel.

In distributed systems, other parameters are also relevant to compute the cost of a plan, such as the size of objects (given by the number of points of spatial objects), the overhead of messages, and the location (server or processor) of data partitions. Although these new parameters bring additional challenges to histogram building, including how to compute these values from datasets and how to estimate their values in intermediate histograms, they are useful when estimating the query communication cost.

These open issues mean that to efficiently estimate the total cost of a multiway spatial join query in a distributed system, new algorithms and formulae need to be proposed. We address these issues in Chapter 3.

2.4.3 Plan Scheduling

In a distributed system, the optimizer must also specify on which server each query fragment will be executed [48]. Consequently, this decision also specifies where each data partition should be copied from, observing that data partitions are distributed a priori.

Different strategies exist to specify the server where to execute a query fragment. Some of them are: *i*) place smaller query fragments on the servers that handle the largest partitions, to reduce data communication, *ii*) place query fragments in such a way that network contention does not cause processors sleep, and *iii*) place query fragments in a way that maintains cluster balance. We refer to this decision process as *plan scheduling*. It is also referred to as copy selection and sub-query allocation in the literature for multi-database systems [65].

The query optimizer can generate the schedule for a plan at different stages in the optimization process. An initial proposal is to view each possible schedule for a plan as a distinct sub-plan, further increasing the search space in plan enumeration and plan selection. According to Özsu and Valduriez [65], this may result in higher query startup times. A better approach is to use hybrid optimization techniques [13, 31]. They split the execution plan into two parts: *i*) a static plan, built at compile time, to determine the access methods to use and the order of dataset processing, and *ii*) an execution plan, generated at runtime, to determine the copy and site selection.

The existing methods proposed for plan scheduling employ heuristics that use as input the network topology, the size and location of data partitions, and a general objective – such as minimizing the response time (increasing the degree of parallelism) or reducing the consumption of resources. In addition, they are designed considering a modest number of data fragments and sites, and only search for locally optimal strategies, rather than global ones [65].

Nowadays, the availability of cloud environments with a large number of machines and the large size of spatial datasets creates a demand for specific algorithms, specially designed for the combinatorial nature of the problem. In this thesis, we address the shortcomings of existing plan scheduling approaches by proposing new algorithms based on the theory of combinatorial optimization. We dedicate Chapter 4 to deal with plan scheduling, from the perspective of single and multiway spatial join queries.

2.4.4 Plan Selection

The primary purpose of a query optimizer is the selection of the plan that will be used to execute the query, considering the set of possible plans and the estimated cost for each one.

In distributed systems, at least two main strategies of selection exist:

1. select the plan that minimizes the total query time, i.e., the sum of the times the query takes in each server; reducing the total query time results in an improvement of resource utilization, increasing system throughput, and
2. select the plan that minimizes the query response time, considering only the time of the server that stays for longer executing the query, also referred to as the makespan of the query.

Reducing response time sometimes can increase resource consumption. For example, raising the degree of parallelism increases the use of network bandwidth, but divides the load on the servers [65]. In practice, the optimizer establishes a compromise between these two strategies.

The query optimizer searches through the space of equivalent plans enumerated by the plan enumeration method. For queries involving a small number of datasets, this space is commonly searched using a dynamic programming algorithm. For larger queries, however, randomized algorithms are employed. Furthermore, the optimizer needs to estimate the cost of each enumerated plan, using a cost model that predicts the use of computational resources [65].

The plan selection can be thought as a final objective, after considering the computational resources consumption in each step of the query optimization. Thus, the optimization can be resumed as the following procedure: (1) while enumerating the plans for a query, (2) the cost model estimations is used to (3) define a schedule of the execution and compute the final costs incurred, and finally, by comparing the costs for enumerated plans, we (4) select a good plan to execute the query. As we mentioned, (1) is a well-studied problem and in this thesis we focus on (2), (3), and (4) in later chapters.

2.4.5 Query Execution

In this step the query is finally executed, according to the computed execution plan. The query execution engine dispatches query fragments according to a pre-computed schedule, and reports the query results. Moreover, the execution engine also performs the copy of partitions, according to the pre-defined data distribution.

The execution of multiway spatial queries can be implemented using algorithms for distributed spatial join, i.e., it does not require specific multiway algorithms, as the steps can be combined in a pairwise fashion. Furthermore, it can greatly benefit from non-blocking algorithms, due to the fact that plan steps are executed in a pipeline, reducing query execution time and the need for storing intermediate results.

Query execution is a relevant problem to be considered in a distributed system. Recent research has reported relevant engines for distributed query execution, such as

the MapReduce framework [24], not specifically targeted for spatial query execution but with relevant results reported (e.g., [39]), and its in-memory counterpart and more query focused, Spark engine [92], also with relevant results reported (e.g., [27]).

In this thesis, we made an effort to specify the minimal set of operations that an execution engine must implement to perform a selected plan following its respective schedule. We also implemented this specification to compute the result set for a multiway spatial join query and to report relevant measurements in our experiments. However, as we focused on planning the query execution, i.e., selecting good execution plans with respect to computational resource consumption, a complete execution engine is outside the scope of this thesis. Despite this decision, we show that our results are applicable to the previously-mentioned execution engines as well, when the computational cost can be estimated a priori.

2.5 Final Considerations

In this chapter, we presented the relevant concepts and structures proposed for non-distributed processing of multiway spatial join queries and discussed some open issues related to the distributed processing of this type of spatial query. We also presented concepts of distributed query planning, such as the components of a distributed query optimizer. A summary of the open issues addressed in this thesis is given in the following.

First, in Chapter 3, we propose a cost model to estimate the resource consumption of simple and multiway spatial join queries. To improve the accuracy of estimates, we study the hashing of complex spatial objects, i.e., objects with spatial extent (lines, polygons), and introduce a method to perform the histogram construction. Also, we propose new algorithms to compute intermediate histograms, taking into account the inherent properties of the distributed environment. An example is the communication cost, which is mainly determined by the data distribution, rather than by the properties of the spatial dataset itself.

In Chapter 4, we study how to define a schedule for the execution of a query plan in a distributed system. We propose and evaluate a multi-objective linear integer model for this problem and apply combinatorial methods to solve it. To the best of our knowledge, the formal definition of models and the application of combinatorial techniques were little explored in distributed database literature.

Next, in Chapter 5, we study how to control the usage of computational resources, i.e., processing power and network capacity. In general, to achieve a better balance in the query execution (and consequently a reduced makespan), an extra cost is incurred to transfer partitions to machines that are underloaded. We study how to control the scheduling behavior in this respect.

Finally, in Chapter 6, we show how to integrate all the proposed pieces in the method for plan selection, defining the complete query optimizer. We also present the execution engine specification and the evaluation of the query optimizer.

In the next chapter, we present the mentioned cost model for distributed multiway spatial join queries that supports the plan cost estimation.

CHAPTER 3

A Cost Model for Distributed Multiway Spatial Join Queries

Chapter 2 introduced the main components of a distributed query optimizer and the challenges that distributed systems and spatial data bring to the cost estimation of multiway spatial join. In a query optimizer, the cost model deals with the prediction of resource consumption (cost), before query execution. In this chapter, we propose and evaluate a new model to estimate the cost of distributed multiway spatial join queries.

As well as being designed for multiway spatial join, the cost model proposed here comprises a set of steps that also address the estimation of other kinds of spatial queries, such as the already-presented windows and simple spatial join. Thus, we start by introducing methods to estimate the cost of these simpler queries and then show how to build intermediate histograms for more complex queries using the predicted intermediate results. We also show how to use these intermediate histograms to compute the cost of a multiway spatial join query. Furthermore, we present a set of statistical formulae that provide more accurate estimation of join selectivity for real spatial datasets having complex lines and polygons.

In Section 3.1, we discuss relevant metadata to support cost estimation and propose a new multidimensional histogram data structure to store it. We also present new methods to improve histogram construction and the precision of estimates. Section 3.2 presents a new method to calculate the number of histogram cells, i.e., the number of subdivisions used to split each data dimension. Section 3.3 proposes a method to gather metadata for histogram cells and a new strategy to determine the histogram cells in which a spatial object needs to be represented. We introduce a new algorithm to construct histograms for estimated intermediate results in Section 3.4, and show how to use these histograms to compute the cost of an execution plan for a multiway spatial join in Section 3.5. Section 3.6 presents the results of a set of experiments that were designed to evaluate the proposed methods, and finally, Section 3.7 concludes with our final considerations.

3.1 Multidimensional Grid Histograms

Besides the traditional use of cardinality, which is common in centralized systems, in this section we present the relevant metadata that needs to be captured from spatial datasets to perform the cost estimation of multiway spatial join queries in distributed systems.

The cardinality of a dataset, or of parts of it, is key to predict the size of the join output, also referred to as *join selectivity* or *join cardinality*. However, the use of the cardinality alone can lead to imprecise estimates. For instance, let us consider two spatial datasets, A and B, whose spatial extents overlap with each other, but with no spatial objects intersecting each other. Alone, the cardinality value of the two datasets would lead to the conclusion that an intersection join will have $|A| \times |B|$ results, i.e., the Cartesian product. To improve this estimation, the length of spatial objects in each dimension of the data is also relevant, as shown in [56]. If all objects are points, their average length in each dimension will be zero and we can predict a better (still approximated) value for the join output. In general, as the average length of objects increases, the probability of occurring an intersection increases as well and this is also true for more complex spatial objects, such as lines and polygons.

Regarding processing costs, predicate check algorithms from computational geometry, such as intersections, containments, and overlaps, often are time consuming to process due to the high numbers of objects and their number of line segments [19]. Thus, the cardinality and the total number of points[1] of the objects are relevant in the estimation of computational time.

Regarding communication cost, we considered the number of points, the size of spatial objects, the size and location of data partitions and replicas, and the costs with the data copy protocol as the key components in the estimation of communication costs when transferring data partitions in the network.

Other informations associated with the dataset are also relevant in the definition and refinement of data structure sizes, such as the dataset cardinality, the average length of all spatial objects, and the associated standard deviation for each data dimension. For instance, this information is relevant in the definition of the extent of data partitions.

Let d be the number of data dimensions, r the maximum number of replicas permitted for a cell, and $e_{1..d}$ the number of cells in each dimension d. The data structure that stores the metadata information for a dataset is illustrated in the pseudocode in Algorithm 3.1. Each histogram cell is represented by a Histogram-Cell structure that stores:

- the MBR, i.e., boundaries of the cell (line 6),

[1] The number of line segments of an spatial object can be determined based on the number of points.

- the cell cardinality, i.e., the number of objects within its boundaries (line 7),
- the proportional cardinality (line 8), explained in Section 3.3,
- the size of the objects in the cell (line 9), i.e., their total number of points,
- the total area of the objects that are polygons (line 10),
- a list of server identifiers that indicates the storage location of the cell (line 11), and
- the average length of the objects in the cell, for each data dimension (line 12).

Furthermore, global dataset metadata is stored as: the type of objects in the dataset (line 15), the dataset spatial extent or MBR (line 16), the dataset cardinality (line 17), the average length of all objects in each dimension (line 18), the standard deviation of the average length (line 19), and the d dimensional matrix with the histogram cells (line 20).

The space complexity of the DATASET-METADATA structure in Algorithm 3.1 depends on the number of data dimensions, d, the number of cells in each of them, e_k, $1 \leq k \leq d$, and the number of replicas r. Considering the constant size of primary types (int, double, and char) as occupying one unit of storage space, the space complexity is:

$$1 + 2d + 1 + d + d + (2d + 1 + 1 + 1 + 1 + r + d) \prod_{k=1}^{d} e_k = \Theta\left(d + (r + d) \prod_{k=1}^{d} e_k\right).$$

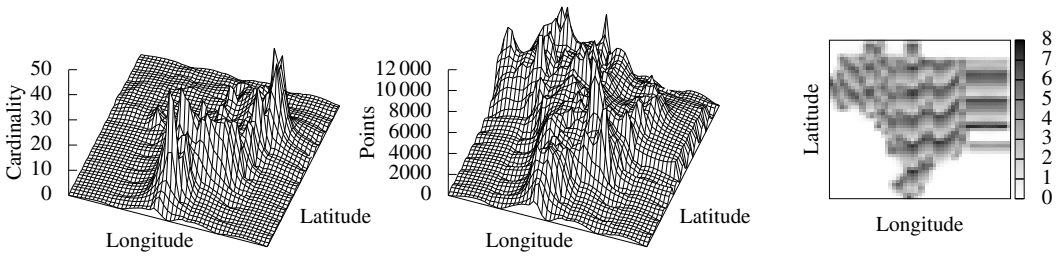
Algorithm 3.1: Data structures for dataset and histogram cell metadata.

```
1  struct MBR {
2      double min[d]
3      double max[d]
4  }
5  struct HISTOGRAM-CELL {
6      MBR mbr
7      double cardinality
8      double prop_cardinality
9      double obj_size
10     double areasum
11     int locations[r]
12     double avg-length[d]
13 }
14 struct DATASET-METADATA {
15     char type // Line or Polygon
16     MBR mbr
17     double cardinality
18     double avg-length[d]
19     double stddev-avglen[d]
20     HISTOGRAM-CELL histogram[e_1, ..., e_d] // A d-dimensional matrix
21 }
```

Figure 3.1 shows a partial illustration of a multidimensional histogram generated for a dataset that represents the political limits of Brazilian municipalities. The histogram captures important aspects of the dataset, as shown in the figure: at the top left corner of Figure 3.1a, the cardinality is low because this region corresponds to the Amazon Forest, where there are fewer municipalities with vast sizes, thus resulting in lower cardinalities. In contrast, Figure 3.1b shows the number of points for the same municipalities. The number of points is much higher than in other regions because the area (extent) of the objects is larger and more points are needed to represent their contour. Figure 3.1c, in turn, is a map of the location of cells.

We assume that the gathering of metadata occurs during the loading of the dataset into the system, and we use Welford's online algorithm [47] to compute standard deviations, in order to avoid multiple passes over data input. The load phase partitions the data according to the defined histogram grid, one partition for each histogram cell, and distributes the generated partitions to servers. We use a round-robin algorithm to distribute data partitions. The server id is set in the $locations$ field of the HISTOGRAM-CELL structure. This field reflects the initial distribution of the partitions and is updated when the join algorithm generates replicas during its execution.

We restricted our research to the cost model and have not provided a study of the initial data distribution. We chose the round-robin method due to the lack of a data location strategy, thus leaving room for future work to investigate strategies for data copy reduction. Most likely, better algorithms can be developed or used to perform the data distribution. Notwithstanding, all the methods proposed in this thesis work independently of the method used for data distribution.



(a) **(b)** **(c)**

Figure 3.1: *Multidimensional histogram for a two-dimensional dataset that represents the political limits of Brazilian municipalities. The graphs represent (a) the cardinality, (b) the number of points, and (c) the location of partitions (in gray tones). The scale on the right indicates the server, from 1 to 8, for which the partitions were assigned. A value of zero (white) represents an unassigned partition that contains no objects.*

3.2 Split Method

In this section, we present a method to define the number of cells in each dimension of a grid histogram, based on dataset metadata. It is based on the following reasoning:

- The number of cells determines the number of data partitions and consequently, a small number of data partitions limits the degree of parallelism in query execution;
- The division of the dataset spatial extent by the average length of objects in each dimension produces a number of cells that may improve the precision of the cost estimation, as a result of the reduction in boundary effects;
- We should establish an upper bound on the number of cells due to the impact on the query optimization process and on the overheads of processing and of moving many small data partitions within the system. Thus, when we have a small average length and a large spatial extent in the dataset, we establish an upper bound on the number of divisions in each data dimension; and
- A small number of cells should also be avoided, as it would generate skewed sized partitions, due to the skewed nature of spatial data.

We propose the following method to define the number of cells, as detailed in Algorithm 3.2, where M_A is the dataset's metadata structure, d is the number of data dimensions, mi is the minimum number of cells in each dimension, and ma is the

Algorithm 3.2: Procedure SPLIT-METHOD that defines the number of cells for multidimensional grid histograms.

SPLIT-METHOD(M_A, d, mi, ma)

1 // Let $M_A.mbr_k$ be the length of dataset extent in dimension k
2 **for** $k = 1$ **to** d
3 $\quad objspan = M_A.avg\text{-}length[k] + M_A.stddev\text{-}avglen[k]$
4 $\quad cells[k] = \frac{M_A.mbr_k}{objspan}$
5
6 // At least mi cells in each dimension
7 **for** $k = 1$ **to** d
8 $\quad cells[k] = \max(mi, cells[k])$
9
10 // At most ma cells in the histogram
11 $adjust = \sqrt[d]{\frac{\prod_{k=1}^{d} cells[k]}{ma}}$
12 **if** $adjust > 1$
13 $\quad$ **for** $k = 1$ **to** d
14 $\quad\quad cells[k]$ /= $adjust$
15 **return** $cells$

maximum number of cells for a histogram. Lines 2–4 define an initial number of cells for each dimension, based on the spatial extent of the dataset divided by the average length plus the standard deviation of all objects. Lines 7–8 ensure the generation of at least a feasible minimum number of cells and lines 11–14 proportionally reduce the number of cells when it reaches the specified maximum.

3.3 Gathering Metadata for Histogram Cells

When building a histogram, each object in a dataset needs to be hashed into the histogram grid to find the cell, or the set of cells, that it intersects with. In this section, we propose a new method of hashing a spatial object, to improve the accuracy of cost estimation. Furthermore, we show how to fill the metadata fields of each histogram cell.

In a similar way as with global dataset metadata, some metadata fields are trivially gathered. In a single pass over the dataset objects we can calculate:

- The $cardinality$ field, which produces the number of spatial objects that overlap the cell boundaries, whether they are entirely contained within the cell or not;
- The obj_size field, obtained by summing up the number of points of each object that intersects a cell, including the points outside the cell if any; and
- The $avg\text{-}length$ field, which stores the average length of the objects that overlap with the cell boundaries in each dimension. When objects overlap more than one cell, this field considers only the length of the MBR of objects within the cell.

To calculate $prop\_cardinality$, we increment the value stored in each cell based on the proportion of the area of intersection between the spatial object's MBR and the histogram cell boundaries, and the total area of the spatial object's MBR. Formally, let M_A be the metadata structure for the dataset A, and let $o \in A$ be an object of the dataset. The proportional cardinality of a cell $h \in M_A$ is given by (3-1). We refer to this method as the Proportional Overlap (PO) method.

$$h.prop\_cardinality = \sum_{o \in A} \frac{area(o \cap h.mbr)}{area(o)} \tag{3-1}$$

Figure 3.2 illustrates the hash process of a spatial object in a histogram. In the MBR center method (left), the center of the object's MBR identifies the histogram cell to which the object is assigned. The cardinality of that cell is thus incremented by one. In contrast, the Proportional Overlap method (right) increments the cardinality of each cell that intersects the object's MBR by adding the proportion obtained in (3-1).

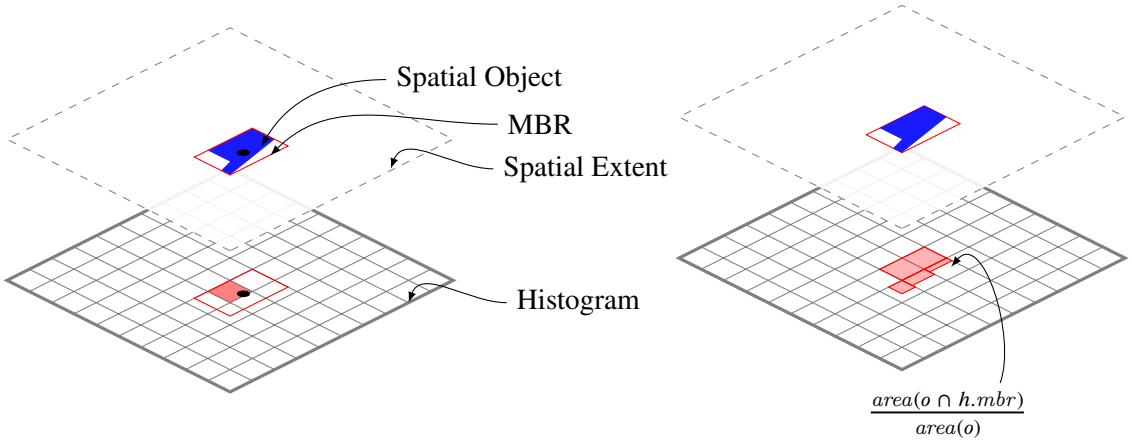


Figure 3.2: *A spatial object hashed using the MBR Center method (left) and the Proportional Overlap method (right).*

3.4 Building Intermediate Histograms

In this section, we introduce algorithms to build an intermediate histogram, based on the original histograms for two given datasets. The intermediate histogram will be a part in the process of estimating the cost of execution plans for multiway spatial join queries, as one of the inputs for intermediate steps.

Here we focus on the intersection predicate, the most frequent predicate used for spatial join queries [12]. Nonetheless, our methods have applicability to similar predicates, such as "contains", "within", "touches", and "disjoint", if we switch the explicit usage of intersection by a matching predicate algorithm and an estimation formula. Other predicates, such as "distance", may require more work due to significant differences in the structure of estimation [17].

Procedure BUILD-INTERMED-HISTOGRAM in Algorithm 3.3 defines the process of building an intermediate histogram. Let M_A and M_B be the metadata for two given datasets, and let M_R be the resulting estimated metadata structure that will be generated by the procedure. Lines 1–9 initialize the M_R structure, identifying the area of interest (line 2) and copying the limits of the reference cells and average lengths. To improve the estimate precision, the method uses as a reference the dataset (A or B) specified in the predicate of the plan step that follows the step being estimated, relabelling it as M_N to simplify the algorithm (line 3). Lines 11–17 loop through each intersecting cell pair h_A, h_B, identifying the respective g cell on M_R, calling the auxiliary procedure ESTIMATE-CARDINALITY-WITH-AVGLENGTHFIX to estimate the proportional cardinality (line 14). The proportional cardinality is used to compute the cardinality (line 16) and object size (line 17), based on the average of previous values multiplied by the estimated new cardinality for g.

Algorithm 3.3: Procedure BUILD-INTERMED-HISTOGRAM that generates an intermediate metadata for a spatial join between datasets A and B.

```
BUILD-INTERMED-HISTOGRAM(M_A, M_B)
 1  M_R = new DATASET-METADATA
 2  M_R.mbr = MBR-INTERSECTION(M_A.mbr, M_B.mbr)
 3  Let M_N be M_A or M_B, according to the next step predicate
 4
 5  for each h ∈ M_N.histogram | h.mbr ∩ M_R.mbr ≠ ∅
 6      g = new HISTOGRAM-CELL
 7      g.mbr = h.mbr
 8      g.avg-length = h.avg-length
 9      append(g, M_R.histogram)
10
11  for each a ∈ M_A.histogram, b ∈ M_B.histogram | a.mbr ∩ b.mbr ≠ ∅
12      Let g be the respective cell of the pair a, b on M_R
13      Let h be a or b, according to the choice on line 3
14      c = ESTIMATE-CARDINALITY-WITH-AVGLENGTHFIX(a, b)
15      g.prop_cardinality += c
16      g.cardinality += (h.prop_cardinality / h.cardinality) * c
17      g.obj_size = (h.obj_size / h.cardinality) * g.cardinality
18
19  return M_R
```

Before explaining the procedure ESTIMATE-CARDINALITY-WITH-AVGLENGTHFIX, we introduce improved estimates to predict intersections between spatial objects, replacing the more general ones given in (2-2) and (2-3).

The equation we propose next (3-2) addresses the imprecision that the MBR induces in the average length field, regarding complex spatial objects (lines and polygons). For instance, let us consider object o in Figure 3.3. The average length of o in both dimensions (x and y) are the same as if it is a square occupying all of the cell space. However, the probability of intersection of o with another object remains at 0.5, due to its format. This issue also arises for line objects not parallel to the sides of a histogram cell. Indeed, spatial objects of the same dataset frequently represent the same natural phenomenon and do not overlap with each other. In other words, in each dimension, the sum of the length of all objects is less or equal to the cell length. However, the MBR approximation can lead to an average length that distort this behavior, i.e., the average length multiplied by the cell cardinality is larger than the cell length, causing a larger probability of intersection. Thus, we propose that the largest average length for all dimensions be proportionally reduced considering the second largest average length, according to (3-2), where h is a histogram cell, $h.mbr_i$ is the length of cell h in dimension i, $h.mbr_j$ is the length of cell h in dimension j, l is the average length vector (i.e.,

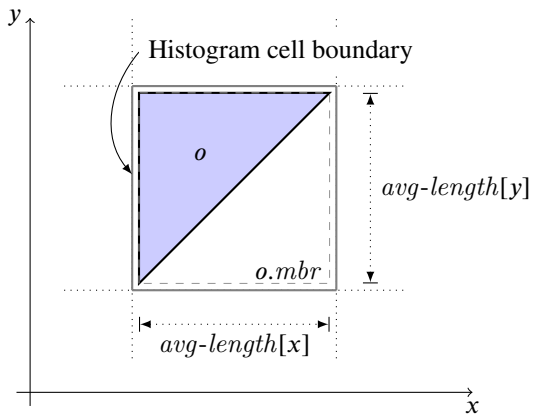


Figure 3.3: *Illustration of the error introduced by MBR on the average length for a 2d object (o).*

$h.avg\text{-}length$), and i and j are the indices of the dimensions with largest and second largest average length.

$$fixavgl(h,i,j) = \min\left(l[i], \frac{h.mbr_i}{(l[j]\cdot h.cardinality)/h.mbr_j}\right) \tag{3-2}$$

The second set of equations we propose, (3-3) to (3-5), is used to estimate intersections between two datasets, one with objects of type line and the other with objects of type polygon. These equations determine the cardinality (J_{lp}) of a join between the objects of two histogram cells (a and b), where mbr_k is the cell length in dimension k, l_{ak} and l_{bk} are the average length for a and b in dimension k, respectively, $\bar{\bar{a}}$ and $\bar{\bar{b}}$ are the cardinalities of the cells, and γ indicates the factor by which we have to increase the average length of polygons in each dimension to fill the entire cell space. The formulae works by finding how many times a line of length l_{ak} intersects a scaled polygon of length $l_{bk} * \gamma$. It is used to reduce the returned value proportionally to the density ρ of polygons in b, calculated using their area $b.areasum$.

$$\rho = \frac{b.areasum}{\prod_{k=1}^{d} b.mbr_k} \tag{3-3}$$

$$\gamma = \sqrt[d]{\frac{\prod_{k=1}^{d} b.mbr_k}{\bar{\bar{b}}\prod_{k=1}^{d} l_{bk}}} \tag{3-4}$$

$$J_{lp}(a,b) = \bar{\bar{a}}\cdot\rho\cdot\prod_{k=1}^{d}\max\left(1.0, \frac{l_{ak}}{l_{bk}\cdot\gamma}\right) \tag{3-5}$$

Finally, we introduce relations (3-6) to (3-8) to estimate the number of intersections between two datasets with line objects. They can be used to determine the join cardinality (J_{ll}) between two histogram cells a and b, where $h[x]$ is the length of a line in cell x, based on the average length, assuming that it is a straight line, and l_{abk} is the length of

the intersection between a and b in dimension k. The other variables are defined as above. $0 \leq \eta \leq 1$ is a coefficient of line intersection used to reduce the number of intersections, observing the minimum of the probability that two arbitrary line segments intersects each other and the ratio between the average line lengths for the two cells.

The constant $^{133}/_{432}$ was used as an upper bound limit approximation coming from the following observation. We are interested in the probability of intersection between two line objects. Suppose that they are two line segments and note that there are three ways of pairing them: horizontally aligned, vertically aligned and intersecting each other (similar to an X or T shape). Out of these three ways of pairing them, we are interested in the latter, i.e., $^1/_3$. Additionally, it is more likely that an intersection between two line objects from distinct datasets occurs when they cross each other (i.e., similar to an X shape) instead of in a perpendicular format (similar to a T shape). Thus, the intersection defines the two diagonals of a convex quadrilateral. To form a convex quadrilateral, from the four points that defines the quadrilateral (the extreme points of the two segments) one needs to fall outside the triangle defined by the other three. As the probability of the point falling within the triangle is known to be $^{11}/_{144}$ [84], the probability of it falling outside is $^{133}/_{144}$ and hence the required probability is $^1/_3 \cdot {}^{133}/_{144} = {}^{133}/_{432}$.

$$h[x] = \sqrt{\sum_{k=1}^{d} (l_{xk})^2} \tag{3-6}$$

$$\eta = \min\left(\frac{133}{432}, \frac{\min(h[a], h[b])}{\max(h[a], h[b])}\right) \tag{3-7}$$

$$J_{ll}(a,b) = \bar{\bar{a}} \cdot \bar{\bar{b}} \cdot \eta \cdot \prod_{k=1}^{d} \min\left(1.0, \frac{l_{ak} + l_{bk}}{l_{abk}}\right) \tag{3-8}$$

For two datasets with polygon objects, we use (2-3), but fix the average length vector by (3-2). We refer to it below as J_{pp}.

The auxiliary procedure Estimate-Cardinality-With-AvgLengthFix, Algorithm 3.4, uses these equations to calculate the proportional cardinality ($proc\text{-}cardinality$ field of Histogram-Cell). It receives two histogram cells as input (a and b) and returns the estimated cardinality of the set of intersections between the objects in the two cells. Lines 4–10 apply the average length reduction, the first proposed modification (3-2). Line 12 identifies the intersection, $inter$, between a and b, due to the boundaries of the two histogram cells being different, i.e., they may partially overlap each other because each dataset has particular grid cell limits. The algorithm then estimates the cardinality that the window query, defined by $inter$, will return for a (line 14) and b (line 15). These cardinalities ($\bar{\bar{a}}$ and $\bar{\bar{b}}$) are used when computing the join cardinality (line 16) by using the adequate equation (J_{lp}, J_{ll}, or J_{pp}) according to the type of the objects in the datasets.

Algorithm 3.4: Procedure ESTIMATE-CARDINALITY-WITH-AVGLENGTHFIX to estimate the resulting cardinality for an intersection between two histogram cells a and b.

ESTIMATE-CARDINALITY-WITH-AVGLENGTHFIX(a, b)

1 Let $a.mbr_k$ be the length of $a.mbr$ in dimension k
2 Let $b.mbr_k$ be the length of $b.mbr$ in dimension k
3
4 $l_a = a.avg\text{-}length$
5 $i, j = argmax_{k=1..d}(a.avg\text{-}length[k])$
6 $l_{ai} = fixavgl(a, i, j)$
7
8 $l_b = b.avg\text{-}length$
9 $i, j = argmax_{k=1..d}(b.avg\text{-}length[k])$
10 $l_{bi} = fixavgl(b, i, j)$
11
12 $inter = a.mbr \cap b.mbr$
13 Let $inter_k$ be the length of $inter$ in dimension k
14 $\bar{\bar{a}} = a.prop\_cardinality * \prod_{k=1}^{d} \min\left(1, \frac{l_{ak}+inter_k}{a.mbr_k}\right)$
15 $\bar{\bar{b}} = b.prop\_cardinality * \prod_{k=1}^{d} \min\left(1, \frac{l_{bk}+inter_k}{b.mbr_k}\right)$
16 **return**
$$\begin{cases} J_{lp}(a,b), & \text{if } a \text{ is from a line dataset and } b \text{ from a polygon one} \\ J_{lp}(b,a), & \text{if } b \text{ is from a line dataset and } a \text{ from a polygon one} \\ J_{ll}(a,b), & \text{if } a \text{ and } b \text{ are both from line datasets} \\ J_{pp}(a,b), & \text{otherwise.} \end{cases}$$

3.5 Estimating the Cost of an Execution Plan

In this section, we present the procedure ESTIMATE-PLAN-COST (Algorithm 3.5), that estimates the cost of an execution plan for a multiway spatial join query.

Naturally, the costs associated with the query execution need to be estimated based on the protocols used to transfer data through distributed computers and observing the time complexity of algorithms that compute the spatial predicate. Due to the diversity of existing algorithms to perform predicate checks, there is also a diversity of options to specify or estimate the cost. What we propose next is a method that conforms to our algorithm and environment choice. Nonetheless, any given cost formula can be integrated within the procedure and we recall that this is the main characteristic of a cost optimizer. Most relevant is the precision of the cost estimate as it imposes additional challenges due to the impact it has on the balance of the query execution.

The size of data partitions determines the cost of communication. It can be estimated based on the size and location of data partitions. As we have limited our research to in-memory data processing, we assume that there is no local I/O cost.

The cost of predicate checking is estimated based on the time complexity of the algorithms involved. It is composed of three parts: *i*) an initial step that prepares the spatial object geometries from one partition[2], *ii*) the filtering step, i.e., the number of MBR intersection checks, defined by the Cartesian product of objects in the two partitions, and *iii*) the refinement step, that performs the predicate algorithm in each filtered result.

The execution plan is represented by a binary tree, with root node P, whose leaves contain a pointer to the dataset metadata and the intermediate nodes have two pointers to their two sub-trees. ESTIMATE-PLAN-COST (Algorithm 3.5) is called for the root node P of the plan and recursively calls itself until it finds a step of the plan composed of two leaf nodes (lines 1–4). The procedure estimates M_R for the join between datasets M_A and M_B (line 7) using the procedure BUILD-INTERMED-HISTOGRAM, previously presented. Next, the costs that will be used to schedule the tasks are estimated: $\hat{c}$, $\bar{c}$ and w. A loop estimates $\bar{c}$

Algorithm 3.5: Procedure ESTIMATE-PLAN-COST to estimate the costs of an execution plan P.

ESTIMATE-PLAN-COST(P, f^q, $plan\_costs$)

```
1  if P.left is intermediate
2      P.left = ESTIMATE-PLAN-COST(P.left, f^q, plan_costs)
3  if P.right is intermediate
4      P.right = ESTIMATE-PLAN-COST(P.right, f^q, plan_costs)
5  M_A = P.left
6  M_B = P.right
7  M_R = BUILD-INTERMED-HISTOGRAM(M_A, M_B)
8  pairs = {a, b | a ∈ M_A.histogram, b ∈ M_B.histogram, a.mbr ∩ b.mbr ≠ ∅}
9  for each distinct b ∈ pairs
10     for i = 1 to m
11         c̄_ib += i ∉ b.locations ? b.obj_size : 0
12 for each distinct a ∈ pairs
13     for i = 1 to m
14         ĉ_ia += i ∉ a.locations ? a.obj_size : 0
15     for each b that forms a pair with a
16         for i = 1 to m
17             w_a += a.cardinality * b.cardinality // MBR check
18             w_a += b.obj_size // point in polygon tests [43]
19     w_a += prep_cost(a) // Cost to prepare a geometries
20 s = SCHEDULE-PLAN-STEP(M_R, stepf(f^q), ĉ, c̄, w)
21 AGGREGATE-COSTS(plan_costs, ĉ, c̄, w, s)
22 return M_R
```

[2]Preparation of geometries is a technique used in spatial libraries to speed up the execution of repeated intersection checks of target geometries. The preparation consists of building a small index with the line segments of the geometry and caching it for reuse. The cost of the index building is amortized during the predicate check (https://trac.osgeo.org/geos/wiki/PreparedGeometry).

with the cost of moving each object b to each server i (lines 9 to 11). The same estimate is computed for a and $\hat{c}$ (lines 12 to 14). To estimate $\hat{c}$ and $\bar{c}$, the procedure considers the actual locations of each partition, its replicas, and which server i to assign them to, $1 \leq i \leq m$, where m is the number of servers used to execute the query. Lines 15 to 19 loop through intersecting pairs of cells to estimate the load or CPU cost, w, conforming with the three previously mentioned costs for predicate checking. The procedure calls a routine to perform the scheduling of intermediate results (line 20), i.e., where each pair defined by a (and its accompanying b's) will be processed. This procedure uses a parameter f^q and the function $stepf$, both defined later in Chapter 6. Finally, the procedure uses the returned schedule s to perform an aggregation of the step costs into the global plan cost (line 21) – a simple aggregation of costs according to s.

Due to the relevance of the procedure SCHEDULE-PLAN-STEP in this thesis, we dedicate Chapter 4 to introduce it and evaluate its implementation. This procedure defines a schedule for the plan, i.e., where each resulting cell of M_R is to be processed, and consequently, the query processing balance and the communication cost incurred.

3.6 Cost Model Evaluation

This section covers the methodology and experiments designed to evaluate the cost model methods proposed earlier in this chapter.

Each proposed method has an impact on the overall precision of the cost model. We attempted to isolate the error produced by each of them. For this reason, some methods were evaluated using simple window queries. We recall from Section 3.4 that window query estimation techniques are used to compose the more complex estimate of join selectivity and thus, it is reasonable to check its performance. The methods designed to build intermediate histograms for two datasets are evaluated using spatial join queries. To evaluate the overall precision of the cost model, as well as to evaluate the estimation of the data communication cost, the scheduling of execution plans is necessary. We present this evaluation in Chapter 6, after we introduce scheduling methods in Chapter 4.

We chose a set of real spatial datasets, obtained from the Brazilian Institute of Geography and Statistics[3] (IBGE), from the LAPIG Laboratory[4] of the Institute of Social and Environmental Studies (IESA) at UFG and from the Digital Chart of the World[5] (DCW). Table 3.1 shows the selected datasets and their characteristics. All datasets have 2-dimensional objects, which represent geospatial objects on the Earth's surface. We downloaded each dataset from these sources in the well-known Shapefile format (SHP)

[3] www.ibge.gov.br

[4] Image Processing and Geoprocessing Laboratory: www.lapig.iesa.ufg.br/lapig/

[5] http://gis-lab.info/qa/vmap0-eng.html

and used the GDAL[6] and GEOS[7] libraries to extract and process the geometry of each spatial object inside them.

In the experiments using window queries, a set of randomly positioned windows is generated for each dataset. They have different sizes, varying from 0.1% to 30% of the dataset spatial extent. We also indicate other parameters used in each experiment. Table 3.2 presents the join queries used in the experiments. There are 20 queries assembled using the datasets listed in Table 3.1, providing an all-to-all combination of the first five datasets and also an all-to-all combination of the last five datasets. This set has queries joining all types of spatial objects, i.e., line ⋈ line, line ⋈ polygon, and polygon ⋈ polygon, as well as distinct dataset sizes. The join predicate used is intersects. We refer to these queries in the text and figures by their number, ranging from J_1 to J_{20}.

We ran all experiments in a m4.4xlarge Amazon EC2 machine, with an Intel(R) Xeon(R) CPU, E5-2686 v4 model, running at 2.30GHz and with 64GB of RAM. Each dataset is loaded from disk to memory before the execution. The experiments presented

Table 3.1: *Datasets used in experiments.*

Name	Abrev.	Type	Cardinality	SHP File Size (MB)
Brazilian datasets (IBGE and LAPIG)				
Fire alerts	A	Polygons	32,578	11.2
Hydrography	H	Lines	226,963	64.5
Roads	R	Lines	51,646	15.2
Counties	C	Polygons	5,564	38.8
Vegetation	V	Polygons	2,140	4.7
World-wide datasets (DCW)				
Rivers	RI	Lines	943,638	243.2
Rails	RA	Lines	194,261	28.7
Hydrography Inland	HI	Polygons	338,860	136.7
Elevation Contour	EC	Lines	703,574	572.5
Crops	CR	Polygons	123,746	69.3

Table 3.2: *Spatial Join queries used in experiments.*

Name	Query	Join Cardinality	Name	Query	Join Cardinality
J_1	A ⋈ H	4,868	J_{11}	RI ⋈ RA	58,885
J_2	A ⋈ R	3,395	J_{12}	RI ⋈ HI	530,782
J_3	A ⋈ C	34,261	J_{13}	RI ⋈ EC	449,309
J_4	A ⋈ V	34,672	J_{14}	RI ⋈ CR	269,301
J_5	H ⋈ R	55,766	J_{15}	RA ⋈ HI	5,975
J_6	H ⋈ C	268,369	J_{16}	RA ⋈ EC	47,106
J_7	H ⋈ V	252,830	J_{17}	RA ⋈ CR	121,007
J_8	R ⋈ C	70,304	J_{18}	HI ⋈ EC	22,128
J_9	R ⋈ V	63,339	J_{19}	HI ⋈ CR	79,002
J_{10}	C ⋈ V	15,678	J_{20}	EC ⋈ CR	234,900

[6]Geospatial Data Abstraction Library: www.gdal.org

[7]Geometry Engine, Open Source: https://trac.osgeo.org/geos/

next do not require a distributed environment to run, as they capture only the estimated cardinality (or selectivity) and compare it with the real cardinality obtained by performing the spatial query over the dataset.

3.6.1 Evaluation of the Hash Method

This section presents the evaluation of the Proportional Overlap method (PO), which computes the cardinality and number of points for multidimensional histogram cells, as described in Section 3.3. The experiment consists of creating, for each dataset in Table 3.1, two sets of multidimensional histograms: the set A, using the PO method to hash spatial objects over the grid cells and the set B, using the MBR Center method (MBRC). Each set has histograms of sizes $r \times r$, $r \in R = \{25, 50, 100, 200, 300, 400, 500\}$. For each histogram set $t \in \{A, B\}$ and histogram size r, a set $\mathcal{Q}_{rs}^{t}$ of window queries with sides length equal to $s\%$ of the dataset spatial extent in each dimension, $s \in S = \{0.1, 0.2, 0.5, 1, 2.5, 5, 10, 15, 20, 25, 30\}$, were randomly generated. A total of 100 queries for each histogram and query size were randomly generated, i.e., $|\mathcal{Q}_{rs}^{t}| = 100 \; \forall \; r \in R, s \in S$. The estimated selectivity of each query, e_i, is obtained using (2-2). The real selectivity r_i is gathered executing the query over the respective dataset. We summarized query precision for each histogram type t, with size r and query size s, using the relative error sum (RES), shown in (3-9). Finally, the RES for PO is compared with the RES for MBRC to obtain the PO improvement: $improvement = (RES(B, r, s) - RES(A, r, s))/RES(A, r, s)$.

$$RES(t, r, s) = \sum_{q_i \in \mathcal{Q}_{rs}^{t}} |r_i - e_i| \tag{3-9}$$

Besides evaluating a significantly large set of window query sizes, we are particularly interested in window queries that are smaller than the histogram cell, for example, $s < 4$ for $r = 25 \times 25$. The rationale for that is because when estimating spatial join selectivity (Algorithm 3.4, lines 6 and 10), the usage of distinct grid sizes for each dataset requires the estimation of the number of objects in the intersection of two grid cells. As the overlap of two grid cells of distinct datasets is always smaller than the grid cell itself, a window selection smaller than the size of the grid cell is performed.

Figure 3.4 presents the results of the experiment. The chart shows the average improvement of the PO over the MBRC method, for all query sizes, while estimating the selectivity of window queries for a set of datasets and histogram sizes. For all datasets and almost all histogram sizes, the PO method produced improved estimates compared to MBRC. For each dataset, there is a histogram size for which the improvement of PO achieves its maximum and we observe three distinct behaviors in this respect. For the first two datasets, counties (C) and vegeta (V), having polygonal objects with large object

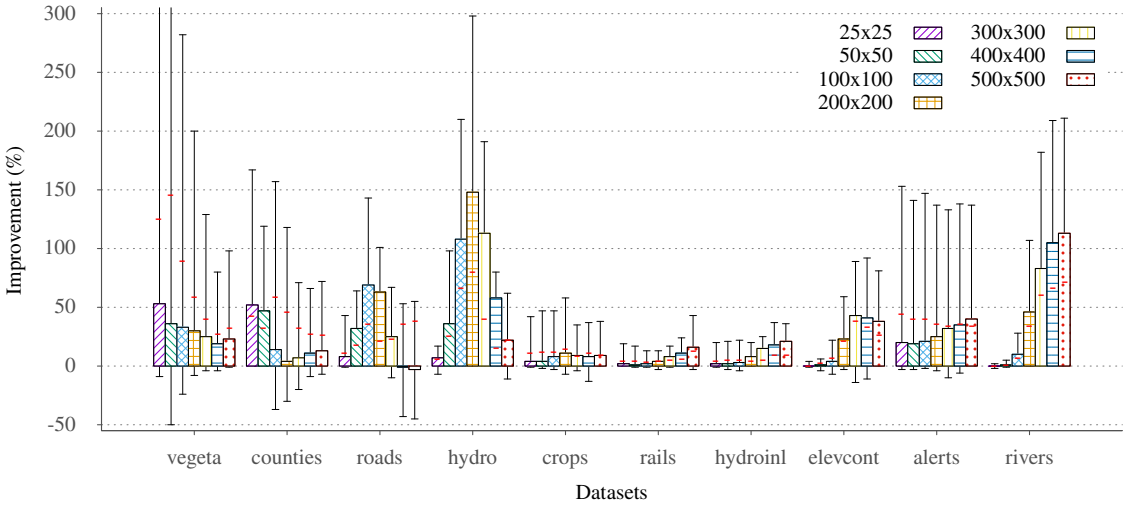


Figure 3.4: *Improvement of the selectivity estimation using the PO hashing method.*

extents, the improvement of PO decreases as the histogram size increases, showing that smaller histograms are better for these datasets – an expected scenario due to dataset characteristics that cause high boundary effects in the histograms. The second group of datasets, roads (R) and hydro (H), presents an improvement in a bell curve format that indicates that a mid-range histogram size will provide a better improvement in the estimated selectivity. Finally, for the third group of datasets, from crops (CR) to rivers (RI), the improvement follows the increase in the histogram size.

The chart in Figure 3.4 also shows error bars indicating the maximum and minimum improvement for all query sizes, and the red ticks at each bar represent the standard deviation. For all datasets, there is a histogram size for which PO significantly improves the selectivity estimation, with a small or nonexistent negative minimum. A negative minimum indicates that MBRC performs better than PO for some specific query size. To further illustrate what happens in these scenarios, Figure 3.5 presents the improvement for all histogram and query sizes, for two contrasting datasets: counties and rivers.

In the counties dataset (Figure 3.5(a)), PO improves the selectivity estimation for histograms of size 25×25 and 50×50 in all query sizes. It also produces improvement for all other histograms in query sizes from 0.1% to 2.5%. For query sizes from 5% to 30% and histograms sizes greater than or equal to 100×100, MBRC outperforms PO. The cause of this behavior is that PO underestimates the selectivity for larger queries, with cell sizes smaller than the length of objects, due to the extreme boundary effect that induces objects to be hashed in a high number of cells. MBRC produces a more precise estimation due to the large size of the query, which mitigates the error caused by the accounting of the spatial object in only one cell.

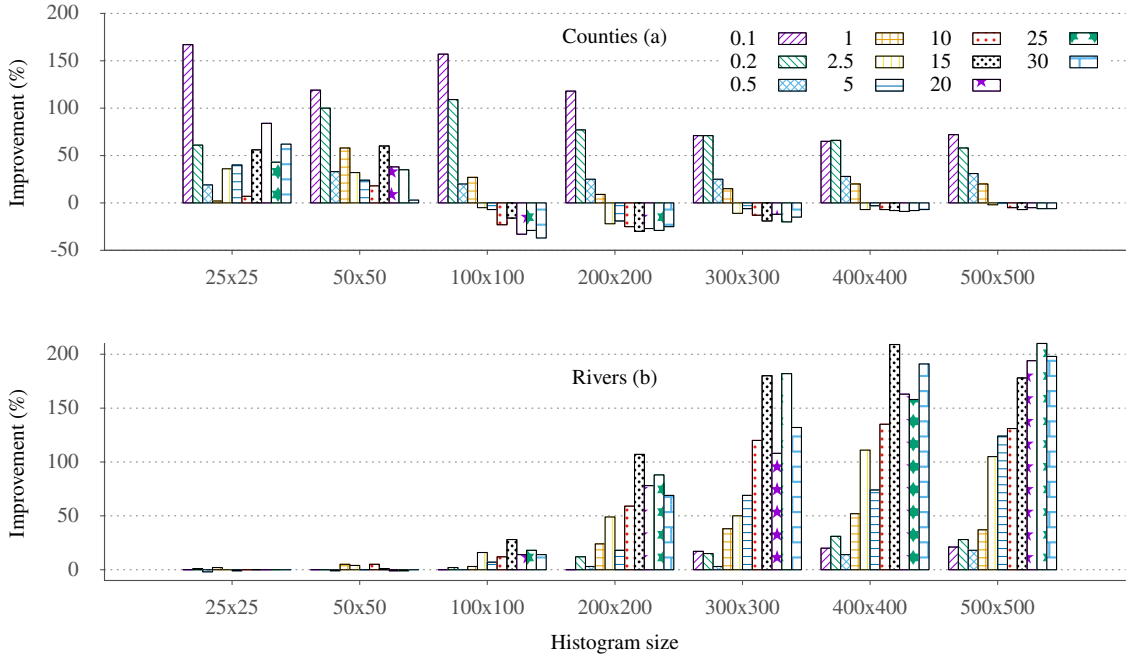


Figure 3.5: *Improvement of selectivity estimation for datasets Counties and Rivers on all histogram and query sizes.*

In the rivers dataset (Figure 3.5(b)), which has a world-wide spatial extent and spatial objects with smaller average lengths, the improvement of PO increases as the histogram size increases. There is no significant improvement for histograms of size 25×25 and 50×50. For small histograms, due to the smaller boundary effect caused by the large size of cells, PO acts in almost the same way as MBRC.

Provided that an adequate histogram size is chosen based on the average length of dataset objects (Split Method), our conclusion is that PO increases the range of possible sizes for histograms and significantly improves the precision of estimated selectivity of window queries. This may provide a higher number of partitions for a dataset, resulting in a possibly higher degree of parallelism in distributed systems.

3.6.2 Evaluation of the Split Method

To evaluate the proposed split method, defined in Algorithm 3.2, we generated a histogram with the size returned by procedure Split-Method and using the PO method, for each dataset listed in Table 3.1. We set the parameter $mi = 5$, that produces a histogram of at least 25 cells and $ma = 29.127$. Larger histogram sizes can be used for some datasets, but we keep this maximum size to limit the size of histogram structure in memory to at most ≈2Mb, considering the size of the histogram cell data structure in our implementation (72 bytes). We also measured the average number of wrongly estimated objects per query (WEO), when estimating window queries with the sizes used in the previous section (S set).

Table 3.3 presents the resulting histogram sizes and the WEO for each dataset. Comparing the resulting histogram sizes with the respective histogram size in Figure 3.4, we can see that the suggested size is nearly always close to the point of largest improvement. Even histograms limited by the ma limit exhibit considerable improvement. For instance, the Hydrography histogram (126×130) lies in between 100×100 and 200×200 in Figure 3.4, the two histograms with the largest improvement for this dataset.

With respect to the estimation error WEO, the result presents a small number of wrong objects when $s = 0.1$. Although WEO increases with the query size, the larger value has less significance in the result set of the query, because, in general, large queries have larger result sets. In summary, the hash and split methods provide a considerable precision when estimating window queries selectivity. The next section will evaluate the estimation of spatial join selectivity.

Table 3.3: *Histogram size determined by* Split-Method *and the number of wrong estimated objects (WEO) per dataset, in average, for* $s = 0.1$ *and for* $s = 30$.

Dataset	**Hist. size**	**WEO avg**	**WEO** $s = 0.1$	**WEO** $s = 30$
Vegetation	7×7	7.98	0.14	19.01
Alerts	133×219	4.92	0.58	7.68
Counties	14×13	12.12	0.28	25.18
Roads	77×75	10.82	0.76	20.23
Hydrography	126×130	17.30	0.99	38.83
Crops	203×78	16.83	1.17	36.63
Rails	249×118	22.76	2.61	40.86
Hydr. Inland	214×137	25.72	1.86	45.23
Elev. Countour	181×161	45.52	2.72	103.45
Rivers	233×125	52.04	2.79	101.00

3.6.3 Evaluation of Join Selectivity

This section presents the evaluation of the intermediate histogram generated by the procedure Build-Intermed-Histogram, in Algorithm 3.3, referred to as the IHWAF method (an acronym for Intermediate Histogram With Average Length Fix). We made the estimate for each join query in Table 3.2 and captured the cardinality of the resulting histogram (M_R) to compare it with the real cardinality obtained by performing the join query over the datasets. Besides providing an estimate for a simple join query with only two datasets, precise estimates provide better intermediate metadata, which can be used later to estimate multiway spatial join queries.

Table 3.4 presents the results of the experiment. They are grouped by the types of spatial objects in the query (L $\bowtie$ L for two datasets with line objects, P $\bowtie$ P for two datasets with polygon objects, and L $\bowtie$ P or P $\bowtie$ L for datasets with line and polygon objects), ordered by the IHWAF relative error. The first three columns indicate the query,

Table 3.4: *Estimated Cardinality Results for Join Queries using the IHWAF and MP methods.*

Query	Type	Cardinality	Estimated Cardinality		Relative Error %	
			IHWAF	MP	IHWAF	MP
J_{13}	L ⋈ L	449,309	443,249	1,843,256	1.3	310.2
J_{11}	L ⋈ L	58,885	57,553	241,473	2.3	310.1
J_5	L ⋈ L	55,766	67,781	398,869	21.5	615.3
J_{16}	L ⋈ L	47,106	61,722	237,809	31.0	404.8
J_4	P ⋈ P	34,672	35,281	63,746	1.8	83.9
J_3	P ⋈ P	34,261	32,837	82,560	4.2	141.0
J_{10}	P ⋈ P	15,678	17,442	31,707	11.3	102.2
J_{19}	P ⋈ P	79,002	53,255	42,986	32.6	45.6
J_{17}	L ⋈ P	121,007	126,257	66,298	4.3	45.2
J_7	L ⋈ P	252,830	264,011	534,048	4.4	111.2
J_8	L ⋈ P	70,304	65,310	250,912	7.1	256.9
J_9	L ⋈ P	63,339	58,505	186,602	7.6	194.6
J_6	L ⋈ P	268,369	290,029	819,124	8.1	205.2
J_{14}	L ⋈ P	269,301	215,062	282,484	20.1	4.9
J_{15}	P ⋈ L	5,981	7,535	40,272	26.0	573.3
J_{20}	L ⋈ P	234,900	172,173	269,542	26.7	14.7
J_{18}	L ⋈ P	22,128	28,374	277,125	28.2	1,152.4
J_2	P ⋈ L	3,395	2,274	60,287	33.0	1,675.8
J_1	P ⋈ L	4,868	2,048	69,045	57.9	1,318.3
J_{12}	L ⋈ P	531,269	63,287	414,137	88.1	22.0
Average					20.9	379.4
Standard deviation					21.1	461.5

the type of spatial objects, and the real cardinality. Columns IHWAF and MP indicate the estimated cardinality and the relative error, calculated when using our proposed method (IHWAF) and the methods proposed by Mamoulis and Papadias [56] (MP). For IHWAF we used histograms constructed with the PO method and the estimation was carried out by procedure Estimate-Cardinality-With-AvgLengthFix. For MP we used histograms generated with the MBRC method and related estimations using (2-2) and (2-3).

In the first four lines in Table 3.4, for two datasets with line objects, the maximum error for IHWAF is 31.0% in the J_{16} query, compared to 404.8% for the MP method. For two datasets with polygon objects (the next four lines), the maximum error is for query J_{19}, where the IHWAF method underestimates the cardinality. This underestimation occurs due to the characteristic of the two datasets used, crops and hydrography, two events that have an inherent proximity due to the frequent usage of fertile soil at riversides. The same underestimation occurs for J_1, also for two commonly co-located events (rivers and deforestation) and for J_{12}, two datasets representing the same natural phenomenon (hydrography and inland water), one using polygons and other using lines. As the method assumes uniformity in each histogram cell, when the dataset data distribution diverges from this assumption the estimate presents an error. In summary, the table shows that

our proposed method (IHWAF) outperforms the baseline method (MP), showing more consistent results with smaller errors (except for J_1 and J_{12}, all other datasets produced a relative error of no more than 33.0%). The average error of IHWAF for all queries is 20.9%, with standard deviation $\sigma^2 = 21.1\%$, while the average of MP is 379.4%, with standard deviation $\sigma^2 = 461.5\%$.

3.6.4 Evaluation of Join Selectivity per Histogram Cell

We also investigated the precision of the estimated cardinality for each resulting histogram cell, $g.cardinality$, as computed by procedure BUILD-INTERMED-HISTOGRAM in Algorithm 3.3. The estimated cardinality of each cell is a key parameter when estimating the load on a server in a distributed environment. Thus, it has a direct impact on the definition of query balance. We captured the estimated cardinality for each histogram cell in all join queries (J_1 to J_{20}) and compared it with the real values obtained when running the query over the dataset.

The results are shown in Table 3.5. As we have thousands of results for each query, we computed the error E for each of them and present statistical values about the E population. There are columns for the average, minimum, maximum, and standard deviation (σ^2), as well as the percentage of resulting histogram cells for which the error fits in a particular range (the last four columns). The minimum error is always equals to zero as

Table 3.5: *Statistics for Estimated Cardinality Results per Histogram Cell for Join Queries.*

	Global statistics				% of cells with error E			
Query	**Min**	**Max**	σ^2	**Average**	≤ 5	≤ 10	≤ 20	≤ 50
J_1	0	24	1.1	0.6	99.1	99.9	100.0	100.0
J_2	0	66	1.5	0.5	98.9	99.7	100.0	100.0
J_4	0	24	1.4	1.0	97.7	99.6	100.0	100.0
J_3	0	47	1.6	1.1	97.6	99.6	99.9	100.0
J_{15}	0	123	5.2	3.6	93.5	97.6	99.2	99.7
J_{18}	0	425	9.1	3.2	84.4	92.3	97.0	99.5
J_{11}	0	89	5.1	3.4	76.9	91.8	98.6	99.9
J_5	0	33	4.4	3.6	73.7	90.9	99.0	100.0
J_{19}	0	199	15.2	7.1	70.0	81.4	90.4	97.6
J_{16}	0	139	8.6	5.7	65.8	82.7	94.2	99.5
J_{17}	0	254	21.2	10.4	60.1	73.2	85.7	95.7
J_8	0	167	14.0	8.1	59.7	76.1	89.5	98.1
J_9	0	146	12.8	7.8	59.4	76.6	89.9	98.4
J_{20}	0	370	32.5	19.1	49.3	59.9	72.1	88.3
J_{14}	0	424	22.0	14.3	48.6	61.2	75.6	93.6
J_7	0	144	9.5	9.0	41.9	66.2	89.3	99.4
J_6	0	126	10.2	9.8	40.0	62.4	86.8	99.4
J_{13}	0	500	26.8	17.7	37.2	53.5	72.1	91.9
J_{10}	0	177	41.6	35.2	31.6	36.8	54.4	74.6
J_{12}	0	959	85.9	45.7	30.7	42.7	58.2	76.9

the method correctly estimate the cardinality of a set of cells in each query. The first eight queries in the table have the majority of cells correctly estimated with more than 90% of the cells presenting an error less than or equal to ten objects. After that, the error increases slowly until the end of the table. The values in Table 3.5 also permit us to make an in-depth study of the estimates presented in the previous section. Query J_{12} exhibits consistent behavior, i.e., it is the worst-estimated query in both experiments. A different behavior occurs for query J_1. While it is the second worst query in Table 3.4, here it is the best one, with less than one percent of the histogram cells presenting an error ≥ 5. The reasoning behind this is the small cardinality of the query, for which even a smaller number of wrong estimates or a large number of smaller errors causes a substantial error when summing up the values to obtain the query cardinality. A markedly different behavior occurs for query J_{10}. Although it is not listed in the worst queries in the previous experiment (11.3% relative error), here it is one of the two worst-estimated queries, together with J_{12}. The case of J_{10} is related to the compensation that occurs when the estimation oscillates between high and low, compared with the real cardinality for the cell.

3.7 Final Considerations

In this chapter, we introduced a model to estimate the cost of multiway spatial join queries. Although it is designed for the estimation of multiway spatial join queries, it can also estimate the selectivity of window and simple spatial join queries, as we have shown during the evaluation of some of its features.

We presented improvements to overcome the errors caused by the MBR simplification of spatial objects and provided specific formulae to estimate join selectivity when the two datasets have line objects or when the objects of one dataset are of a line type and the objects of the other are of a polygon type. The experimental evaluation showed that these new methods significantly improve a previous well-known cost model for spatial queries, while dealing with complex spatial objects and real spatial datasets.

According to our evaluation, there are two main sources of error in the model: (*a*) natural co-located events and (*b*) the divergence between the uniformity assumption in the proposed formulae and the spatial data itself. For (*a*), another metadata field indicating the expected probability of intersection can be of help. However, there are two drawbacks with this. Firstly, one needs to have prior knowledge about the dataset, and secondly, as it is a particular hint about the join itself – not related to each dataset in particular –, we need to know what datasets will be joined in the system[8]. For (*b*), other metadata fields

[8]DBMSs for scalar data use per-query hints to improve cost estimation [11].

to indicate the divergence from uniformity or even a complete redesign of the estimation formulae may be necessary.

Other sources of error exist, such as the mis-handling of boundary effects by the PO method. For this case, improved histogram techniques should be employed. However, it is necessary to adapt such techniques to perform data partitioning in distributed systems. Two related techniques which we are aware of are the MinSkew histogram [1] and the improved version of Euler Histograms proposed in [81]. We leave the implementation and evaluation of these suggestions as future work.

The cost model presented here provides the parameters for the scheduling of spatial join queries in distributed systems presented in Chapter 4. We complete the evaluation of the proposed cost model in Chapter 6, presenting an overall evaluation of estimates for multiway spatial join queries.

CHAPTER 4

Scheduling Multiway Spatial Joins Queries

In this chapter, we consider the problem of assigning jobs, defined by pairs of data partitions from two datasets and aligned by a spatial predicate, to a set of servers or machines in a distributed system. Together with the cost model presented in the previous chapter, the defined assignment, or schedule, establishes the communication cost and the load balance of execution plans for distributed multiway spatial join queries.

The scheduling of a query plan occurs in a critical stage inside the query optimizer: the plan selection algorithm. In the previous chapter, we showed that a schedule is generated for each step of a query and for a number of execution plans. The objective is to compare the cost of plans and select the best one. This is illustrated in the call to SCHEDULE-PLAN-STEP inside the procedure ESTIMATE-PLAN-COST (Algorithm 3.5). Since the number of plans is exponential in the number of datasets, even with a pruned set of plans, the time to select an execution plan needs to be considered from the perspective of the total runtime of the query and the system throughput. For *ad-hoc* queries and focusing better query runtime, the optimizer should take less time than the query execution time itself using a non-optimal plan. Otherwise, we may just run the query directly without optimization. For repetitive queries or focusing better system throughput, this restriction is relaxed as the query will be executed many times and we can amortize the cost of query optimization.

We propose and evaluate a bi-objective linear integer model for this problem and apply two combinatorial methods to solve it: the well-known Linear Relaxation and the more sophisticated Lagrangian Relaxation [32]. To the best of our knowledge, the formal definition of models and the application of these techniques have not been extensively explored in the distributed database literature. We will present scenarios for which these tools can be considered, observing the economy of computational resources due to the fact that approximated solutions are very close to the optimum, in contrast to the difficulty of solving the integer model using exact methods.

The problem studied here relates to the copy selection and sub-query allocation problem described in the distributed database literature, that deals with the allocation and copy of entire relations or fragments of horizontally partitioned relations [65]. Early

solutions for this problem suggest the usage of exhaustive enumeration or heuristics to cope with the NP-hard nature of the problem [90], and due to that, in general, solutions assume a controlled number of disjoint relation fragments, and a small number of replicas.

We considered a generalization of this problem, for which even a single step of a multiway query has a large number of data partitions to handle and the fragments (or data partitions) are non-disjoint by nature, due to the intrinsics of spatial data. Also, due to its time complexity, the predicate checking needs to be split into a number of servers in a distributed system. This problem conforms to modern ways of processing data in distributed systems, such as the MapReduce framework [24] and the Spark engine [92]. Besides the recent criticism of the general applicability of these frameworks (e.g., [71, 80]) due to the lack of attention to the existing distributed database literature, here we attempt to contribute to reducing this gap, presenting methods that consider both the optimization of queries and more flexible ways of data partitioning. This approach has been identified as a future research direction in the literature [26].

Throughout the text, we assume a previous understanding of Linear Programming (LP) and some properties of LP models. We try to achieve a balance between formalism and application and present a brief introduction to Linear Programming and Lagrangian Relaxation in Sections 4.1 and 4.2, respectively. For a more comprehensive introduction, we refer the reader to a helpful introductory textbook on this subject [5] and a more advanced one in approximation algorithms [85, Chapter 12].

The remainder of the chapter is organized as follows. In Section 4.3, we formalize the problem of scheduling individual steps of a distributed multiway spatial join query through a linear integer model. In Section 4.4 we describe related problems found in the literature and discuss the complexity of solving the proposed integer model. Section 4.5 presents a simplified version of the model, which has a particular structure exploitable by Linear Relaxation (Section 4.6) and Lagrangian Relaxation (Section 4.7). Next, in Section 4.8, we introduce a heuristic to repair a partial schedule provided by Linear and Lagrangian Relaxations. This heuristic can also be used as a greedy algorithm to compute schedules for the simplified model (Section 4.9). After discussing the complexity of the proposed methods (Section 4.10), we present their evaluation in Section 4.11. We conclude by showing the broader applicability of the methods (Section 4.12) as well as our final considerations, conclusions, and future directions (Section 4.13).

4.1 Linear Programming Background

Linear Programming (LP) is a set of methods to deal with the minimization or maximization of a linear objective function, subject to a set of constraints expressed as linear inequalities. The objective function and the set of constraints, together, form a

linear programming model. The problem of solving this model, i.e., finding a solution that minimizes or maximizes the value of the objective function subject to the set of constraints, is called a linear program [5].

A linear programming model may contain integral constraints on its variables, i.e., they may be constrained to assume integer or Boolean values. When it has integral constraints, the program is called a linear integer program, or simply, an Integer Program (IP) [15]. Formally, a linear integer program is a combinatorial optimization problem that, in the standard form is:

$$Z_{IP} = \text{Min } \mathbf{cx} \tag{4-1}$$

$$\text{s.t. } \mathbf{Ax} = \mathbf{b}, \tag{4-2}$$

$$\mathbf{x} \geq 0 \text{ and integral.} \tag{4-3}$$

where $\mathbf{c} \in \mathbb{R}^n$ is a row vector, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{x}$ is the vector of unknowns, (4-1) is the objective function, (4-2) is the set of constraints, and the vector inequality in (4-3) means that each component of $\mathbf{x}$ is nonnegative and integral. Variations of this problem, such as $\mathbf{Ax} \geq \mathbf{b}$, $\mathbf{Ax} \leq \mathbf{b}$, or maximizing instead of minimizing, can be rewritten in this standard form [5]. A number of solutions can exist such that (4-2) and (4-3) are satisfied. They are called *feasible* solutions. A feasible solution that minimizes the value of (4-1) is said to be *optimal* and we denote it as x^* and as Z^*_{IP} its value.

Many NP-hard discrete optimization problems can be modeled as integer programs [85]. However, when dealing with NP-hard problems, the existing exact methods to solve integer programs, such as branch-and-bound, cutting-plane, and others, do not efficiently solve large enough instances [87]. A commonly employed approach to cope with this limitation is to relax the requirement of finding an optimal solution, and searching for a "good enough" approximate solution [87].

Besides the intuitively appealing greedy heuristics, another method used to find approximate solutions for an integer program is termed Linear Relaxation. Linear Relaxation consists of relaxing the integer constraints, i.e., removing the integrality constraints on variables, and solving the resulting linear program [85]. When the integrality constraints are removed, the solution that results often has fractional variable values. Thus, to determine a feasible integer solution to the original problem we need to repair the partial solution, rounding the values of the fractional variables (LP-rounding), or using a heuristic algorithm. Of course, repairing the partial solution will not always result in an optimal or near-optimal solution. However, some integer programs may have a bounded number of fractional variables in the relaxed solution and good approximations can be found with this method. Another more sophisticated method in this matter is the Lagrangian Relaxation [32]. The next section gives a brief introduction to it.

4.2 Lagrangian Relaxation

Lagrangian Relaxation is a relaxation method to obtain an approximate problem for an integer program when its constraints can be divided into two groups: a set of simpler constraints, chosen in a way that the problem becomes relative "easy" to solve if it has only these constraints, and a set of difficult constraints. To obtain the Lagrangian problem, we dualize the set of difficult constraints, i.e., move it to the objective function, weighted by a vector of Lagrangian multipliers. The optimal solution to the Lagrangian problem provides a lower (upper) bound to the original minimization (maximization) problem [32].

Consider the following integer program:

$$Z = \text{Min } \mathbf{cx} \tag{4-4}$$

$$\text{s.t. } \mathbf{Ax} = \mathbf{b}, \tag{4-5}$$

$$\mathbf{Dx} \leq \mathbf{e}, \tag{4-6}$$

$$\mathbf{x} \geq 0 \text{ and integral}, \tag{4-7}$$

where (4-6) is the set of easy constraints and (4-5) the set of difficult ones. Moving the difficult constraints to the objective function Z, weighted by a row vector of Lagrange multipliers $\mu \in \mathbb{R}^m$, leads to the following Lagrangian problem:

$$Z_D(\mu) = \text{Min } \mathbf{cx} + \mu(\mathbf{Ax} - \mathbf{b}) \tag{4-8}$$

$$\text{s.t. } \mathbf{Dx} \leq \mathbf{e}, \tag{4-9}$$

$$\mathbf{x} \geq 0 \text{ and integral}. \tag{4-10}$$

This leaves us with the problem of finding a value for μ for which $Z_D(\mu)$ is equal to or nearly equal to Z. A method used to determine μ is the Subgradient Optimization Method, proposed by Held and Karp [42]. The method starts with a vector of multipliers μ^0 and iteratively computes a sequence of $\{\mu^k\}$ using the rule in (4-11), where x^k is an optimal solution to $Z_D(\mu^k)$ and t_k is a positive scalar step size. Indeed, t_k can be determined in practice by using (4-12), where λ_k is a scalar satisfying $0 \leq \lambda_k \leq 2$ and Z^U is an upper bound on Z.

$$\mu^{k+1} = \mu^k + t_k(Ax^k - b) \tag{4-11}$$

$$t_k = \frac{\lambda_k(Z^U - Z_D(\mu^k))}{||Ax^k - b||^2} \tag{4-12}$$

The method runs for a specified arbitrary number of iterations. The sequence of λ_k is obtained by setting $\lambda_0 = 2$ on the first iteration and halving it whenever $Z_D(\mu)$ fails to increase in some arbitrary number of iterations. Additionally, setting $\mu^0 = 0$ is a natural choice, but for some problems, we can choose a μ^0 that accelerates the convergence, based

on the problem structure. The method does not guarantee that μ can be found such that $Z_D(\mu) = Z$, i.e., such that the optimal solution for the Lagrangian problem is equal to the optimal solution for the original problem. Furthermore, there is no way of proving optimality unless a μ^k is obtained for which $Z_D(\mu^k)$ equals the cost of a known feasible solution [32].

It is possible to obtain feasible solutions to the original problem while solving Z_D, however, the occurrence of them is rare [32]. Nonetheless, often a solution for $Z_D(\mu)$ is nearly feasible and can be made feasible by using a repairing heuristic that explores the problem structure.

Besides the intrinsics of the method, Fisher [32] showed a list of NP-hard problems for which LR provided substantially improved solutions. The main advantages of LR are the fast convergence to nearly optimal or optimal solutions, and its use as a replacement for the Linear Relaxation in branch-and-bound algorithms [32].

A comprehensive introduction to Lagrangian Relaxation is given by Fisher [32], including alternative methods to compute μ.

4.3 Problem Formulation

In this section, we introduce a formal integer model of the problem of assigning jobs to a set of servers or machines in a distributed system. Each job constitutes a pair of data partitions that are aligned by a spatial predicate for a join between two spatial datasets. To conform to the related literature, we will use the term *machine* as a synonym of *a server* in a distributed system.

We consider the arrays provided by procedure Estimate-Plan-Cost (Algorithm 3.5), when it calls Schedule-Plan-Step. The call provides the estimated and summed communication cost for each machine and partition, $\hat{c}$ and $\bar{c}$, as well as the processing cost for each job w.

The communication cost of a job is defined based on the data transference that is incurred when moving a data partition from the machine where it is currently located to the machine where it is assigned to be processed. The data partition is copied only once for each machine in which it is used and no communication cost incurs if it is processed in a machine to which it has already been assigned.

The processing cost is defined based on the amount of processing time required to finish a job. We assume that the processing cost of a job is the same on any machine. We also consider a residual load for a machine, arising from a prior unbalanced query execution or other particularity of a system. This residual load is useful when scheduling multiway queries, to consider the unbalance of a prior step and get a better balance of the entire query.

The main objective of the scheduling is to perform the job allocation in such a way that the query load is evenly distributed among the machines and the communication cost incurred is minimized. However, these are two conflicting objectives in the sense that to achieve a better balance in the query execution we incur in extra costs to transfer data partitions to idle machines. To this end, we introduce a parameter termed f, to specify the desired emphasis on a balanced schedule or on a low usage of network capacity.

Let A and B represent the set of cells in the two histograms used in a step of a multiway spatial join query plan. Let $p = |A|$ and $q = |B|$, and m be the number of machines, $m \leq \min(p, q)$. The set of jobs J to be processed is composed of pairs $\{a, b\}$, where $a \in A$, $b \in B$, $J \subseteq A \times B$, and $n = |J|$. Each job $j \in J$ has an associated weight, $w_j \in \mathbb{Z}^+$, that specifies its processing cost. Each a and b has an associated cost, $\hat{c}_{ia} \in \mathbb{Z}^+$ and $\bar{c}_{ib} \in \mathbb{Z}^+$, respectively, which specifies the incurred communication cost when it is processed on machine i, $1 \leq i \leq m$. Each machine has a residual load u_i given in the same units as the processing costs, which represents a previous existing load that reduces its capacity.

There are three sets of binary decision variables:

- $\hat{y}_{ia} = 1$, if a is processed on machine i, or $\hat{y}_{ia} = 0$, otherwise, $\forall\ a \in A, i = 1, \ldots, m$;
- $\bar{y}_{ib} = 1$, if b is processed on machine i, or $\bar{y}_{ib} = 0$, otherwise, $\forall\ b \in B, i = 1, \ldots, m$;
- $x_{ij} = 1$, if the job j is processed on machine i, or $x_{ij} = 0$, otherwise, $\forall\ j \in J$, $i = 1, \ldots, m$.

Additionally, let x_0 be a decision variable that represents the completion time of the latest processed job by any machine, i.e., the makespan. The IP program, denoted by FM for short, to determine the best schedule to process a step of a multiway spatial join query which minimizes the sum of the weighted makespan and the total communication cost is given in (FM.1) – (FM.6).

$$Z_{FM} = \text{Min } f x_0 + \sum_{i=1}^{m} \left(\sum_{a=1}^{p} \hat{c}_{ia} \hat{y}_{ia} + \sum_{b=1}^{q} \bar{c}_{ib} \bar{y}_{ib} \right), \tag{FM.1}$$

$$s.t. \sum_{i=1}^{m} x_{ij} = 1, \qquad \forall\ j \in J \tag{FM.2}$$

$$\sum_{j \in J} w_j x_{ij} + u_i \leq x_0, \qquad i = 1, \ldots, m \tag{FM.3}$$

$$x_{ij} - \hat{y}_{ia} \leq 0, \qquad \forall\ j = \{a, b\} \in J,\ i = 1, \ldots, m \tag{FM.4}$$

$$x_{ij} - \bar{y}_{ib} \leq 0, \qquad \forall\ j = \{a, b\} \in J,\ i = 1, \ldots, m \tag{FM.5}$$

$$x_{ij},\ \hat{y}_{ia},\ \bar{y}_{ib} \in \{0, 1\}. \qquad \forall\ j = \{a, b\} \in J,\ i = 1, \ldots, m \tag{FM.6}$$

Function (FM.1) represents the objective of minimizing the makespan and the sum of the communication costs. Constraint family (FM.2) expresses the requirement that each job must be processed on exactly one machine. Constraint family (FM.3) is a set of logical inequalities arising from the need to minimize the makespan. Constraint families (FM.4) and (FM.5) are a set of logical inequalities to indicate whether or not each particular cell is processed by a particular machine. Constraint family (FM.6) represents the usual integrality constraints, indicating if a job is or is not processed by a particular machine (x_{ij}), and if a data partition is or is not used by a machine ($\hat{y}_{ia}, \bar{y}_{ib}$).

Following the standard nomenclature and classification of the scheduling theory and the three-field notation introduced by Lawler et al. [51], this problem is an extension of the unrelated[1] parallel machines scheduling problem with costs. To compare the FM model with the existing literature, let us briefly introduce a well-studied model called $R||C_{max}$.

$R||C_{max}$ is the problem in which n jobs are to be assigned to m unrelated machines without preemption, each job being assigned to exactly one machine. Job j ($j = 1, \ldots, n$) becomes available for processing at time zero and requires a processing time p_{ij} if assigned to machine i ($i = 1, \ldots, m$). The objective is to schedule the jobs so that the makespan, C_{max}, which is the completion time of the latest finished job, is minimized. Garey and Johnson [35] showed that $R||C_{max}$ is NP-hard in the strong sense, even for the special case of identical processors, $P||C_{max}$, where the processing time of each job does not depend on the processor to which it is assigned ($p_{ij} = p_j$).

Thus, the existence of a polynomial time algorithm to solve FM is remote, unless P=NP. Naturally, we want to know how fast and how well the optimum for FM can be approximated. In the next section, we expand the comparison of FM, highlighting its differences to related problems and discussing approximation algorithms proposed for similar problems.

4.4 Related Problems

As presented in the previous section, FM is an extension of $R||C_{max}$ also referred to as *Unrelated Parallel Machine Scheduling* (UPMS) problem, which is a known NP-hard problem [85] when $m \geq 2$. Lenstra et al. [52] proposed a $2-$approximation algorithm for UPMS, and also showed that there is no approximation better than a $\frac{3}{2}$-approximation unless P=NP. Shchepin and Vakhania [76] presented an LP-rounding procedure with a worst case performance ratio of $2-\frac{1}{m}$ and proved that it is the best approximation ratio that can be achieved using the rounding approach. FM differs from the UPMS problem as it

[1] Unrelated is used as defined by [51]. It means that we have individual costs for each job-machine pair, represented by $\hat{c}_{ia}$ and $\bar{c}_{ib}$ in the model.

considers the minimization of costs in the objective function as well as the makespan. Also, a set of additional restrictions was added to compute the total cost, (FM.4) and (FM.5).

Unrelated Parallel Machine Scheduling with Costs (UPMS-C) is a more similar problem to FM, proposed by Shmoys and Tardos [78], which considers both the makespan and an additional cost to process a job. The formal description is: find a schedule that minimizes the total cost and the makespan, given a set J of jobs, a set M of machines, the time w_{ij} to process job $j \in J$ on machine $i \in M$, a cost c_{ij} to process the job $j \in J$ on machine i, and a maximum limit of time T_i to use machine i. They presented an approximation algorithm that given C and T constants finds a schedule with cost at most C and makespan at most $2T$, if such a schedule with cost C and makespan T exists. Angel et al. [4] proposed a Fully Polynomial Time Approximation Scheme (FPTAS) for UPMS-C. They focus on the case where the number of machines is a fixed constant, i.e., not part of the input. The FPTAS finds a schedule for any $\epsilon > 0$ with a makespan at most $(1+\epsilon)T$ and cost at most $C_{opt}(T)$, in time $O(n(n/\epsilon)^m)$, given that a schedule of makespan T exists. Despite the similarity in the objective of FM and UPMS-C, they are still different. Again, FM presents additional constraints that are needed to compute the total costs, using each data partition that composes a job.

Vazirani [85] discusses an interesting property of a Linear Relaxation for the UPMS model. Given the number n of jobs and the number of machines m, any extreme point solution assigns at least $n - m$ jobs integrally to a machine. The remaining m jobs can be set fractionally. Thus, when m is small, only a limited number of jobs needs to be reallocated in the partial solution and a repairing procedure applied to it can generate a good feasible solution for some instances. However, in the general case, this procedure provides only a 2-approximation algorithm, as it is possible to provide tight instances for which it does not perform well [85]. Furthermore, the number of constraints in FM is greater than in UPMS. UPMS has $n + m$ constraints while FM has $2mn + n + m$, arising from the additional constraints to compute the cost. Compared to the number n of jobs, the number of constraints in FM is higher, and thus, Linear Relaxation for FM does not perform as well as for UPMS.

The absence of an approximation algorithm with a good performance ratio for these similar problems tells us about the difficulty to solve FM. To discover some structure in the problem that can be successfully explored by LP and LR relaxations, we present a simplified version of FM in the next section.

4.5 Simplified Model

The main complicating constraints in FM, when compared to related problems, are (FM.4) and (FM.5), which provide a means to compute the total cost. A possible

simplification to FM is to ignore the fact that a data partition should not be redundantly transferred in the network, assuming that it is copied every time it is used in a machine. Thus, instead of a per partition cost, we define a per job cost c_{ij}, summing up the costs of partitions a and b for a job $j = \{a, b\}$:

$$c_{ij} = \hat{c}_{ia} + \bar{c}_{ib}. \tag{4-13}$$

This simplification removes the need for additional variables and constraints to compute the communication cost and leads to a relatively easier model to solve. The new simplified model SM, obtained from FM by applying this modification, is:

$$Z_{SM} = \text{Min } f x_0 + \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}, \tag{SM.1}$$

$$\text{s.t.} \sum_{i=1}^{m} x_{ij} = 1, \qquad j = 1, \ldots, n \tag{SM.2}$$

$$\sum_{j=1}^{n} w_j x_{ij} + u_i \leq x_0, \qquad i = 1, \ldots, m \tag{SM.3}$$

$$x_{ij} \in \{0, 1\}. \qquad j = 1, \ldots, n;\ i = 1, \ldots, m \tag{SM.4}$$

SM preserves the properties of extreme point solutions for UPMS about the number of constraints, variables, and fractionally set jobs that are useful for a Linear Relaxation when m is small. We also show in Section 4.7 that this simplified problem presents an important structure for LR-relaxation. Furthermore, a feasible solution x to SM is feasible for FM as well, because: *i*) FM and SM share the same set x of variables, *ii*) none of the FM constraints are violated by x, and *iii*) it is possible to determine the additional variables $\hat{y}$ and $\bar{y}$ from x. We can obtain upper bounds for either Z_{SM} and Z_{FM} by substituting x in (SM.1) and (FM.1), respectively.

One drawback of this modification is that the resulting schedule may result in a lack of opportunities to reduce the overall communication cost due to the concentration of jobs involving the same partition on a given machine, to the detriment of improving the makespan. Informally, suppose an instance with three distinct jobs formed by a data partition with the same cost c for all machines (ignore for a moment the other side of the pair that composes the job). The cost of processing this instance in one or two machines is the same, $3c$, as SM assumes that it is transferred three times. Thus, as it will not incur an extra cost, SM can divide the load between the two machines to reduce the makespan.

Formally, the following describes a family of instances with this behavior. Let p partitions of A and q partitions of B compose n jobs $\{a, b\} = A \times B$, implying $n = pq$. For the sake of simplicity, assume $n = m$ and $f = 1$. Let the communication cost be the

same for all partitions and machines ($\hat{c}_{ia} = \bar{c}_{ib} = \kappa m$, for all $a = 1, \ldots, p$, $b = 1, \ldots, q$, and $i = 1, \ldots, m$), and the processing costs $w_j = \kappa$ for all $j \in J$. An optimal solution for SM will assign one job to each machine, with $x_0 = \kappa$ and communication cost $2\kappa mn = 2\kappa m^2$, resulting in $Z_{SM} = \kappa + 2\kappa m^2$ and $Z^*_{SM} = Z_{FM}$ since all partitions are copied to all machines. An optimal solution for FM, in turn, will assign all jobs to only one machine, with $x_0 = \kappa m$, communication cost $\kappa mpq = \kappa mn$, and $Z^*_{FM} = \kappa m + \kappa m^2$. Comparing Z_{FM} with Z^*_{FM}, the former provided by SM and the later provided as the optimum for FM, we have:

$$\kappa + 2\kappa m^2 = \kappa m(1/m + 2m) > \kappa m(1 + m) = \kappa m + \kappa m^2.$$

Notwithstanding, in the instances of spatial join queries with data partitions generated from spatial histograms cells, the re-use of data partitions in jobs is expected to be low because after all, the purpose of the histogram grid is to generate disjoint partitions that reduce the number of predicate checks. Thus, a solution to SM for these instances should provide a reasonable value for Z_{FM}. We study this behavior in the evaluation section.

4.6 Linear Programming Relaxation for SM

The Linear Relaxation for SM consists of removing the integrality constraints and letting x_{ij} assume fractional values in the solution. Besides (SM.4) also provides an upper limit, $x_{ij} \leq 1$, the same bound is also imposed by constraint family (SM.2).

The optimal solution for the Linear Relaxation can be computed by a LP method, such as the well-known Simplex algorithm [5]. The solution, however, will be infeasible for SM if it has fractionally set jobs, i.e., jobs partially scheduled on two or more machines. In this case, we use a repairing heuristic to fix their scheduling. We denote this procedure as LP in the following.

Both LP and the method resulting from the Lagrangian relaxation use the same repairing procedure (Repair-Partial-Solution), and we introduce it in Section 4.8.

4.7 Lagrangian Relaxation for SM

Recall from Section 4.2 that to make a Lagrangian relaxation for an IP we must split its constraints into two sets: a set of difficult constraints that will be dualized in the objective function, and another set of easier constraints, such that solving the IP with these later constraints is relatively easier compared to the original IP. For SM, a possible Lagrangian relaxation (LR) is obtained by dualizing constraints (SM.2) into the objective function using Lagrangian multipliers $\mu_j, \forall\ j \in J$. The resulting model is given in (LR.1).

$$
\begin{aligned}
Z_{LR}(\mu) &= \text{Min } f x_0 + \sum_{j \in J} \sum_{i=1}^{m} c_{ij} x_{ij} + \sum_{j \in J} \mu_j \left(\sum_{i=1}^{m} x_{ij} - 1 \right), \\
&= \text{Min } f x_0 + \sum_{j \in J} \sum_{i=1}^{m} \left(c_{ij} + \mu_j \right) x_{ij} - \sum_{j \in J} \mu_j, \qquad \text{(LR.1)} \\
&\text{s.t. (SM.3) and (SM.4).}
\end{aligned}
$$

An important property of LR is that it reduces to m 0-1 Knapsack Problems, one for each constraint in the family (SM.3). In the following we introduce a 0-1 Knapsack model and show how to use it to compute solutions to $Z_{LR}(\mu)$.

A 0-1 Knapsack is a problem of choosing a subset of $\hat{n}$ items, each with a profit p_j and weight $\hat{w}_j$, $j = 1, \ldots, \hat{n}$, such that the profit sum of the selected items is maximized and the sum of weights does not exceed the capacity v [85]. A 0-1 Knapsack model is given in (K.1)–(K.3) for reference. (K.1) is the objective function, (K.2) is a constraint family to restrict the selected items not to exceed a given knapsack capacity v, and (K.3) defines the integrality constraints for x_j to indicate the selected items:

$$
\begin{aligned}
Z_K &= \text{Max } \sum_{j=1}^{\hat{n}} p_j x_j, && \text{(K.1)} \\
&\text{s.t. } \sum_{j=1}^{\hat{n}} \hat{w}_j x_j \leq v, && \text{(K.2)} \\
& x_j \in \{0, 1\}, \qquad j = 1, \ldots, n. && \text{(K.3)}
\end{aligned}
$$

The m 0-1 Knapsack Problems for LR are obtained in the following way: Let $K_{LR}(i, \mu)$, $i = 1, \ldots, m$ be the i-Knapsack problem for LR. The number of items to select from is $\hat{n} = n$, one item for each $j \in J$. The weights $\hat{w}$ for each problem are obtained from w values, and profits p from c and μ. A lower bound for v can be determined by $\sum_{j \in J} w_j / m \leq x_0$. The complete model for a $K_{LR}(i, \mu)$ is defined by (KLR.1) to (KLR.3). The inversion of the signal for the sum of profits in (KLR.1) is due to Knapsack being a maximization problem, while SM is a minimization problem.

$$
\begin{aligned}
Z_{K_{LR}}(i, \mu) &= \text{Max } -\sum_{j=1}^{n} \left(c_{ij} + \mu_j \right) x_{ij}, && \text{(KLR.1)} \\
&\text{s.t. } \sum_{j=1}^{n} w_j x_{ij} \leq v - u_i, && \text{(KLR.2)} \\
& x_{ij} \in \{0, 1\}, \qquad j = 1, \ldots, n. && \text{(KLR.3)}
\end{aligned}
$$

Finally, we present a procedure to compute μ and how to obtain feasible solutions to SM, based on the iterative Subgradient Optimization Method. Algorithm 4.1 shows the steps of the method in pseudocode. After setting initial values in lines 1 and 2, we compute and set the initial upper bound on Z_{SM} (line 3), by calling a greedy algorithm that is presented next. The value of v (line 4) is set by using a best-fit heuristic that provides an upper bound by ordering the jobs by decreasing the makespan and allocating them to the least-used machine. The lower bound for v is set in line 5. In lines 7 and 8 the method iteratively solves the m K_{LR} knapsack problems. Next, in line 9, the vector of subgradients σ is computed. If the value of $Z_{LR}(\mu^k)$ increases above the upper bound limit Z^U, we reduce the value of v by an arbitrary small percentage (line 10) and return to line 7. Line 11 calls the procedure Repair-Partial-Solution, given next, to repair the partial solution x, transforming it into a feasible solution to SM. Next, based on a possibly improved upper bound Z^U (line 12), a new t_k and μ^{k+1} for the next iteration are computed (lines 13 and 14). Line 15 updates λ if a better Z_{LR} is not found in λ^i iterations. The method stops and returns the best feasible solution found ($\hat{x}$) if the condition in line 16 is satisfied at any iteration or if the last iteration t is reached. We refer to this procedure as the LR method.

Algorithm 4.1: Procedure Solve-LR-Relaxation to compute a feasible solution to SM through LR-relaxation.

Solve-LR-Relaxation(c, w, S)

1 $\mu^0 = 0$
2 $\lambda = 2$
3 Z^U is set with an upper bound on Z_{SM}
4 $v =$ best-fit UB for x_0
5 $v^{lb} = \sum_{j \in J} w_j/m$
6 **for** $k = 0$ **to** t
7 **for** $i = 1$ **to** m
8 Solve $K_{LR}(i, \mu^k)$ and partially set x, for i
9 $\sigma_j = \sum_{i=1}^{m} x_{ij} - 1, \forall j \in J$
10 reduce v and **goto** 7 **if** $\left(v > v^{lb}$ **and** $Z_{LR}(\mu^k) > Z^U\right)$
11 $\hat{x} =$ Repair-Partial-Solution(x)
12 $Z^U = Z^{\hat{x}}$ if $Z^U > Z^{\hat{x}}$
13 $t_k = \dfrac{\lambda\left(Z^U - Z_{LR}(\mu^k)\right)}{\|\sigma\|^2}$
14 $\mu^{k+1} = \mu^k + \sigma t_k$
15 $\lambda = \lambda/2$ if a better Z_{LR} is not found in λ^i iterations.
16 **stop if** $t_k < 1 \times 10^{-4}$ and $\lambda < 1 \times 10^{-4}$
17 **return** best $\hat{x}$

4.8 Repairing Heuristic

This section introduces a heuristic to repair a partial schedule provided by the LP or LR relaxations, transforming it into a feasible solution for both SM and FM. Let x be the partial solution provided by the LP or LR method, and let us partition the jobs into three sets defined by:

$$S_1 = \left\{ j \in J \mid \sum_{i=1}^{m} x_{ij} = 0 \right\},$$
$$S_2 = \left\{ j \in J \mid \sum_{i=1}^{m} x_{ij} = 1 \right\},$$
$$S_3 = \left\{ j \in J \mid \sum_{i=1}^{m} x_{ij} > 1 \right\},$$

where S_1 is the set of unassigned jobs, S_2 is the set of jobs that are correctly assigned, and S_3 is the set of jobs that were multiply assigned. All $j \in S_1 \cup S_3$ need to be repaired to transform x into a feasible solution.

Furthermore, let us introduce the concept of regret for a job. The regret r_j for a job j is defined based on the difference between the maximum and the minimum cost that may be incurred if the job was scheduled in the worst or the best possible machine, plus its load w_j weighted by f. Formally, r_j is defined by (4-14). We use this concept to sort the assignment of jobs, in a way that jobs that have a large load ($f w_j$) or a large regret are scheduled first.

$$r_j = f w_j + (max_{i=1}^{m} c_{ij} - min_{i=1}^{m} c_{ij}) \tag{4-14}$$

We repair x using the procedure Repair-Partial-Solution in Algorithm 4.2. Let $\hat{x}$ be the feasible solution under construction. The procedure starts by setting the residual load of previous steps u (line 1 and 2). Next it computes the number of machines for which each $j \in J$ is allocated (t) and uses it to build the sets S_1 and S_3 (lines 4 to 6). If j is correctly set ($t = 1$), $\hat{x}$ is set accordingly (line 9), and its load w_j is added to the array of loads for each machine i (line 10). The remaining jobs $j \in S_1 \cup S_3$, denoted as S_u, are assigned to machines by the procedure Schedule-Unassigned-Jobs (line 11). After this call, $\hat{x}$ has a feasible solution for SM. This solution is further improved by procedure Improve-Repaired-Solution (line 12). Next, we describe these two auxiliary procedures.

Procedure Schedule-Unassigned-Jobs, in Algorithm 4.3, starts by sorting the jobs in S_u by decreasing order of r_j (line 1). Next, for each item $j \in S_u$, the procedure finds the machine s for which the assignment of j least increases the cost (lines 2 to 10), assigns j to it (line 11), and updates the load on machine s (line 12) for the next iteration. The procedure stops when all jobs are assigned to machines. For the sake of simplicity, we

Algorithm 4.2: Procedure REPAIR-PARTIAL-SOLUTION to repair a partial solution x to SM.

REPAIR-PARTIAL-SOLUTION(x, w, c, u, f)

```
1  for i = 1 to m
2      load[i] = u[i]
3  for j ∈ J
4      t = Σ_{i=1}^{m} x_ij
5      S_1 = S_1 ∪ {j} if t = 0
6      S_3 = S_3 ∪ {j} if t > 1
7      if t = 1
8          Let i be the machine where j is allocated
9          x̂_ij = 1
10         load[i] = load[i] + w_j
11 SCHEDULE-UNASSIGNED-JOBS(load, S_1 ∪ S_3, x̂, w, c, f)
12 IMPROVE-REPAIRED-SOLUTION(load, x̂, w, c, f)
13 return x̂
```

Algorithm 4.3: Procedure SCHEDULE-UNASSIGNED-JOBS to schedule unassigned jobs in S_u.

SCHEDULE-UNASSIGNED-JOBS$(load, S_u, \hat{x}, w, c, f)$

```
1  Sort S_u by decreasing r_j
2  for j ∈ S_u
3      lowcost = ∞
4      for i = 1 to m
5          zinc = c_ij
6          mkspaninc = w_j − (x_0 − load[i])
7          zinc = zinc + f ∗ mkspaninc if mkspaninc > 0
8          if zinc < lowcost
9              lowcost = zinc
10             s = i
11     x̂_sj = 1
12     load[s] = load[s] + w_j
```

use x_0 to represent the makespan. In this context, it can be obtained from the maximum value of the $load$ array after line 1 and updated after line 12 if $load[s]$ surpasses the stored x_0 value.

After the scheduling of the jobs in S_u, we further search for jobs for which a machine exchange is worthwhile. Let x_0^i be the machine with the largest load, x_1 be the second largest load for all machines, and p_j be the index of the machine where j is assigned. Procedure IMPROVE-REPAIRED-SOLUTION, in Algorithm 4.4, searches for a new machine s where to schedule j, $j \in J$, such that the processing and communication costs reduce the

most (lines 1 to 12). If there exists such machine s (line 13), j is moved from p_j to s (lines 14 and 15) and the load for machines s and p_j are updated accordingly (lines 16 and 17). Note that the value of $zinc$ (line 9) is positive if moving j from p_j to i does not improve the solution, and negative otherwise.

Algorithm 4.4: Procedure IMPROVE-REPAIRED-SOLUTION to improve the feasible solution $\hat{x}$.

IMPROVE-REPAIRED-SOLUTION($load, \hat{x}, w, c, f$)

```
1  for j ∈ J
2      s = −1
3      lowcost = ∞
4      for i = 1 to m, i ≠ p_j
5          newx_0 = x_0
6          if s = x_0^i
7              newx_0 = newx_0 − min(x_0 − x_1, w_j)
8          newx_0 = max(newx_0, load[i] + w_j)
9          zinc = f ∗ (newx_0 − x_0) + (c_ij − c_{p_j j})
10         if zinc < lowcost
11             lowcost = zinc
12             s = i
13     if s ≠ −1
14         x̂_sj = 1
15         x̂_{p_j j} = 0
16         load[s] = load[s] + w_j
17         load[p_j] = load[p_j] − w_j
```

4.9 A Greedy Algorithm for SM

Besides being used to repair a partial solution to SM, the procedure SCHEDULE-UNASSIGNED-JOBS (followed by IMPROVE-REPAIRED-SOLUTION) can also be used to make a complete schedule, starting with no scheduled jobs in $\hat{x}$, $S_u = J$, and an empty $load$ array.

There are two purposes in using it this way: *i*) to compare the performance of the combinatorial methods (LP and LR) with the performance of an intuitively appealing but simple method, and *ii*) to use it when the limit of time imposed on the query optimization is critical, for example, for queries with small runtime.

We refer to this way of using these procedures as the GR method since it constitutes a greedy heuristic to provide solutions to SM.

4.10 Complexity of the Algorithms

We have proposed three distinct methods to provide feasible solutions to SM: a greedy algorithm GR, the method based on the Linear Relaxation (LP), and the method based on the Lagrangian Relaxation (LR). In this section, we discuss the time complexity of these methods.

Let $s = |S_u|$, $0 \leq s \leq n$. The initial ordering of S_u in Schedule-Unassigned-Jobs takes $\mathcal{O}(s\ lg\ s)$. The loop that follows (lines 2 to 13) iterates over s jobs and m machines for each of them, $\Theta(sm)$. Thus, the time complexity of Schedule-Unassigned-Jobs is $\Theta(sm) + \mathcal{O}(s\ lg\ s)$. The time complexity of Improve-Fixed-Solution is $\Theta(nm)$ as it iterates over all jobs and all machines. Observing that $S_u = J$ and $s = n$, the time complexity of GR is the sum of these two auxiliary procedures, $\mathcal{O}(n\ lg\ n) + \Theta(nm)$.

The time complexity of the LP method can be determined based on the complexity of the Simplex method executed to identify a partial solution plus the time needed to repair the partial solution, and the complexity of Repair-Partial-Solution. Repair-Partial-Solution has an additional time complexity of $\Theta(nm)$ to compute the load and identify the set of unscheduled jobs. Repecting the limit on the number of jobs to repair in LP, $s \leq m$, and adding the complexity of the two auxiliary procedures results in a time complexity of $\mathcal{O}(m\ lg\ m) + \Theta(nm)$. The time complexity of the LP method, however, will be dominated by the call to the Simplex algorithm that for practical purposes can be taken as polynomial in the number of constraints and variables [41]. Recalling that SM has $n + m$ constraints and $nm + 1$ variables, the time complexity of the LP method is thus: $\mathcal{O}(n + m + nm)$.

LR method is the most complex of the three algorithms studied, with a complexity determined by the solution of m 0-1 Knapsack problems in each iteration t, plus the call to Repair-Partial-Solution and the calculation of the Lagrangian multipliers. We also need an initial Z^U, that can be obtained using GR or LP. Updating Lagrangian multipliers takes $\Theta(nm)$ time. A 0-1 Knapsack Problem can be solved in pseudo-polynomial time [72], which means that its time complexity depends on the parameters given as input (e.g., v). The number of knapsack problems solved will dominate the time complexity. Although we can solve $\mathcal{O}(tm)$ problems for a tricky instance, t being the number of iterations, we expect a small t for a typical instance, as the method converges relatively fast.

4.11 Evaluation

We evaluate the three proposed methods using the set of join queries presented in Table 3.2 and a new set of multiway spatial join queries presented in Table 4.1. These new queries provide instances with intermediate results. We refer to multiway queries in

the text as $M_{i.j}$, where i is the query number and j is the step of the multiway query. For example, M_1 has three steps referred to as $M_{1.1}$, $M_{1.2}$, and $M_{1.3}$. We also present the number of jobs n for each query in Tables 4.1 and 4.2. The total number of tested instances is 36 and they were scheduled for $m = (4, 8, 16, 32, 64)$ machines, that is, 180 schedules for each method.

The parameters λ^i and t were set to $\lambda^i = 50$ and $t = 3000$, respectively, when executing the procedure Solve-LR-Relaxation (Algorithm 4.1). The maximum number of iterations t was reached for 11 instances. For the other, in general, it stayed between 140 and 2700. The initial Z^U was provided by the GR method.

All algorithms were coded in the C language and compiled using Clang[2] version 3.8.0, with optimization flags `-Ofast -march=native`. Pisinger's minknap algorithm[3] [72] was used to solve $K_{LR}(i, \mu^k)$ inside the LR algorithm. To find the extreme point solution for the LP method, we used an academic license of IBM ILOG CPLEX Optimization Studio[4], version 12.6.1. The model and its parameters are set into CPLEX optimization module through CPLEX C API calls, and the extreme point solution is captured after the optimization finishes.

The experiments were executed in a m4.4xlarge Amazon EC2 machine, with an Intel(R) Xeon(R) CPU, E5-2686 v4 model, running at 2.30GHz, and with 64GB of RAM. Experiments that display execution time were performed in a more controlled local

Table 4.1: *Additional multiway instances to provide intermediate results and the number of jobs n of each step.*

		Jobs n in each step		
Name	**Query**	$M_{i.1}$	$M_{i.2}$	$M_{i.3}$
M_1	$((A \bowtie RI) \bowtie RA) \bowtie CR$	148	69	69
M_2	$(RI \bowtie RA) \bowtie EC$	5,572	5,477	-
M_3	$(RI \bowtie HI) \bowtie RA$	10,298	5,544	-
M_4	$((R \bowtie RI) \bowtie RA) \bowtie EC$	581	263	229
M_5	$((A \bowtie HI) \bowtie CR) \bowtie C$	112	112	112
M_6	$((RI \bowtie EC) \bowtie HI) \bowtie RA$	10,019	9,870	5,450

Table 4.2: *Number of jobs for each query J.*

Name	**Jobs**	**Name**	**Jobs**	**Name**	**Jobs**	**Name**	**Jobs**
J_1	8,082	J_6	7,587	J_{11}	5,572	J_{16}	4,588
J_2	8,082	J_7	7,755	J_{12}	10,298	J_{17}	4,209
J_3	8,082	J_8	2,139	J_{13}	10,019	J_{18}	8,495
J_4	8,082	J_9	2,160	J_{14}	6,630	J_{19}	5,624
J_5	7,125	J_{10}	114	J_{15}	4,614	J_{20}	5,106

[2] http://clang.llvm.org

[3] http://www.diku.dk/~pisinger/codes.html

[4] http://www.ibm.com/software/commerce/optimization/cplex-optimizer/

environment, using an AMD Phenom(tm) II X6 1055T 2.8 GHz processor with 8G of RAM, with a Linux distribution using Linux Kernel version 3.18.1.

4.11.1 Instances Characterization and the Affect of f

In this section, we characterize the instances tested, with respect to their processing and communication cost, and show how the f parameter affects the generation of schedules by the three proposed methods.

For each query, we generated three schedules using GR, LP, and LR. The first schedule used $f = 0$, i.e., we ignored the makespan to search for the minimum communication cost. The second schedule used f=100,000, a sufficiently large value to make communication cost irrelevant in all tested instances and obtain a schedule with the minimum possible makespan. The third schedule used a particular f for each instance, in order to make the contributions of the makespan and the communication cost in Z_{SM} nearly equal ($Z_{SM}/2 \approx f x_0 \approx \sum c_{ij}$). This choice of value for f also makes the instances hard to solve and was empirically obtained through experimentation by determining the range of values for f, and selecting values for f in the middle of this range, such that $f x_0 \approx Z_{SM}/2$. Additionally, all schedules used $m = 64$, the larger m in our experiments. We selected some representative instances to show next. We provide the additional charts in Appendix A, as well as the list of f values used.

The axes in each scatter plot in Figure 4.1 are for the communication cost, in MB, and the makespan (x_0). The charts show the relationship between the two conflicting objectives. The common point in each chart for all methods (top-left corner) refers to the schedule with minimum communication cost and the largest makespan. To the contrary, the rightmost point for each method refers to the schedule with the smallest makespan (f=100,000). The methods showed different behaviors in this respect. In general, GR provided a schedule with a relatively low makespan but at the price of an increased communication cost. LP provided schedules with a larger makespan and a lower communication cost in general (e.g., J_{10}, $M_{4.1}$), but with some exceptions (e.g., J_3). LR, in turn, usually identified a schedule with the smallest makespan for all queries, sometimes with an increased communication cost (e.g., J_{10} and $M_{1.2}$).

Still, in Figure 4.1, the points near the bottom-left corner are of particular importance. They represent the schedules with a challenging f when we ask the methods to reduce resource consumption the most. GR presented a solution favoring a smaller makespan but with a high communication cost. LP presented almost the same schedule both for a challenging f and for f=100,000. This behavior occurred because LP ignores the integrality constraints and sets the jobs fractionally on machines, causing $x_0 = \sum w_j/m$ for any f value. Thus, the partial schedule returned is always the same and the small dif-

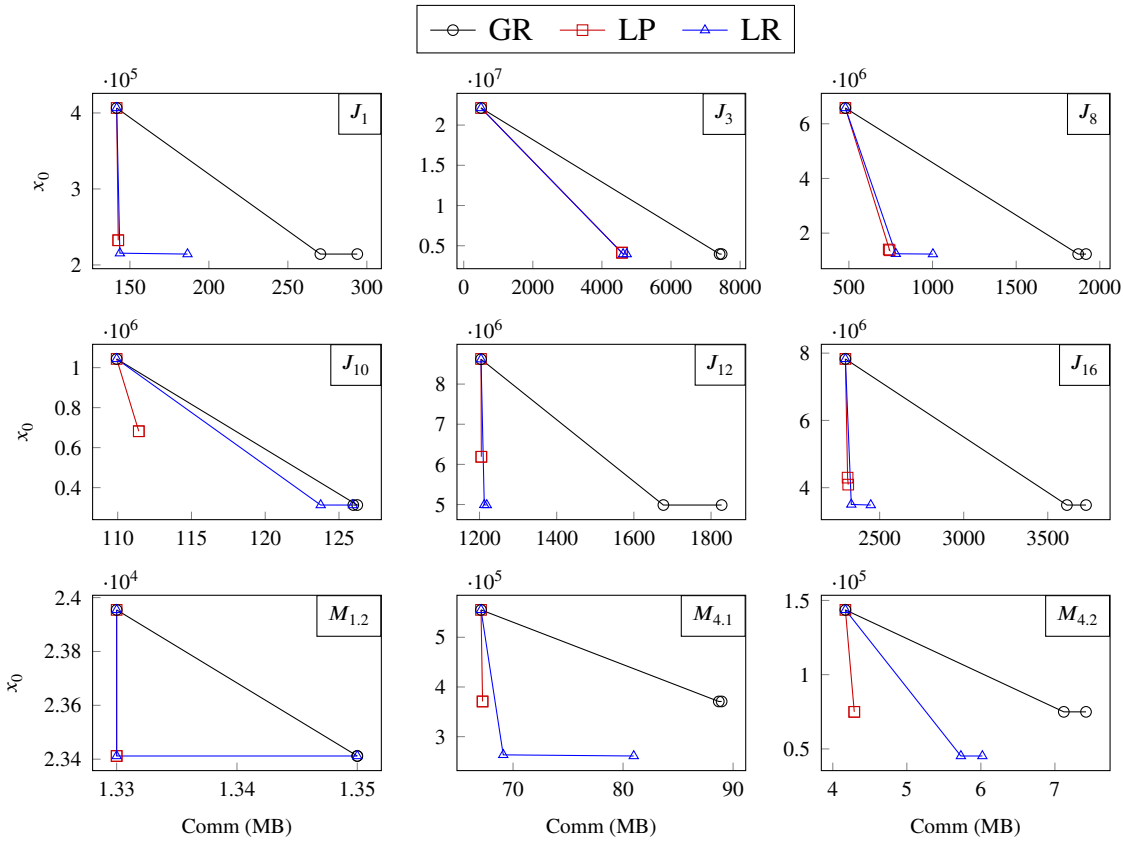


Figure 4.1: *Schedule makespan and communication cost for representative tested instances using distinct values of f.*

ference noticed in J_{16} (and somewhat blurred in J_8) is due to the repairing method, applied at the end. LR provided the best schedules with respect to resource consumption, with a makespan that is usually close to the minimum and a relevant improvement in the communication cost (compare the proximity in x_0 axis between the solution with minimum makespan, at right, and the left point near it in the same line).

4.11.2 Quality of Generated Schedules

In this section, we compare each schedule provided by GR, LP, and LR and show how close they are to a known lower bound (Z^{lb}_{SM}) for the optimal value Z^{*}_{SM}, i.e., how good they are with respect to the best possible schedule for each instance of SM. To get Z^{lb}_{SM}, we processed SM in CPLEX and left the MIP solver to run from the root node, applying all possible cuts. We present the distance between the proposed scheduled and the lower bound, computed as gap $= (Z^{+}_{SM} - Z^{lb}_{SM})/Z^{lb}_{SM}$, where the $+$ signal indicates the method used in each case, e.g., Z^{GR}_{SM}.

Figure 4.2 presents the results. There are five charts, one for each number of machines (m). The gap scale is logarithmic focusing on near-optimal schedules. There are three marks for each query, indicating the gap for GR, LP, and LR. A mark touching

the x axis indicates a gap $\leq$ 0.01%, i.e., a schedule that is very close to the optimum or even an optimal one. The gaps for GR are the largest ones, almost all fitting in the range $10\% - 100\%$. Although there exists an instance for which GR provided a good schedule ($M_{5.2}$ for $m = 64$), the average for all gaps is 25.07%, with a high standard deviation ($\sigma^2 = 31.71\%$). LP improved the GR schedule in almost all instances. Example instances for which it performed worst are M_1, M_2, and M_3, in nearly all steps and m's. The main observation for LP is that the schedules were worst when the number of machines m increased, going from 1.10% for $m = 4$ to 10.88% for $m = 64$. This occurs because the number of jobs that were fractionally set increased with the number of machines. We present some statistical values in Table 4.3 where it is possible to check this behavior. The average of all gaps for LP is 6.38% ($\sigma^2 = 10.74\%$).

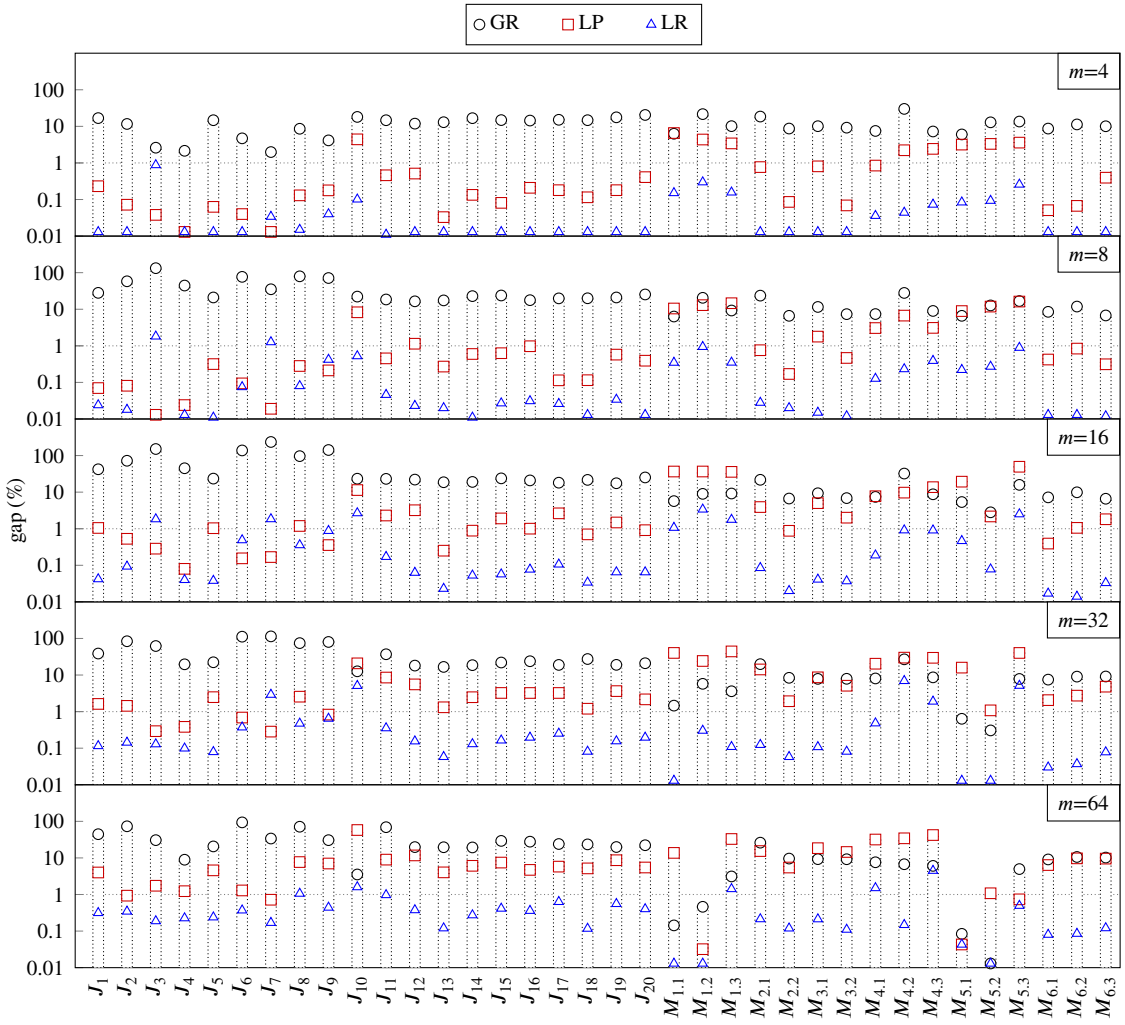


Figure 4.2: *Gap between each schedule provided by GR, LP, and LR and a know lower bound of Z^*_{SM}, for $m \in \{4, 8, 16, 32, 64\}$. y axis is logarithmic to emphasizes the near-optimal schedules.*

Table 4.3: *Average and standard deviation for gaps in Figure 4.2.*

	Average for m					Standard deviation for m				
Method	**4**	**8**	**16**	**32**	**64**	**4**	**8**	**16**	**32**	**64**
GR	12.0	26.8	37.3	27.1	22.2	5.9	26.1	51.0	29.6	22.4
LP	1.1	3.0	7.2	9.8	10.9	1.7	4.7	12.4	12.6	13.0
LR	0.1	0.2	0.6	0.8	0.5	0.2	0.4	0.9	1.6	0.8

LR presented the smallest gaps. Observing Figure 4.2, for $m = 4$, there are 21 out of 36 instances with gap $\leq 0.01\%$ and the other 15 have gap $\leq 1\%$ (check the dotted line at gap $= 1$). The gaps also increased when m increased, but in a small proportion (see LR line on Table 4.3). For $m = 8$, 34 instances still presented gap $\leq 1\%$ and two had a gap $> 1\%$ (≤ 1.8). For $m = 64$, 31 instances presented gap $\leq 1\%$ and the other five $1 \leq$ gap $\leq 4.5\%$. From all 180 runs, there are only 10 cases for which LR presented worse schedules than LP ($J_{\{3,7\}}$ for $m = 4$, $J_{\{3,7,9\}}$ for $m = 8$, $J_{\{3,6,7,9\}}$ for $m = 16$, and J_9 for $m = 32$). LR presented the best gap in all instances when $m = 64$ and also, the best gap for all M instances. The average for all gaps is 0.43% ($\sigma^2 = 0.94\%$).

4.11.3 Comparison of the Execution Times

This section presents the execution time for GR, LP, and LR. Despite the fact that we already provided the time complexity of each method, here we provide the computational experience in solving practical instances of the problem. In this experiment, CPLEX parallel execution option was disabled, i.e., the solver was limited to use only one OS thread. The other two methods, GR and LR, are implemented sequentially, and thus, all methods used only one thread. The time reported is wall clock time, obtained using the function `clock_gettime` with the argument `CLOCK_REALTIME`. Furthermore, we measured only the time taken in the optimization function, discarding initial dataset loading, jobs enumeration, and other final routines such as memory release.

Figure 4.3 presents the results. The y axis shows the execution time in seconds using a log scale. As the behavior is very specific for each instance, we present in (a), (b), and (c), the minimum, the average, and the maximum execution time, respectively, for all instances. The complete set of execution times is provided in Appendix A for further reference. As expected by the time complexity analysis, GR is the best, followed by LP and next by LR. The execution time increases for all methods when m increases. GR has an average time per query of 2.0 ms for $m = 4$, and 7.4 ms to $m = 64$. LP presented the second best time, with an average of 51.6 ms for $m = 4$, and 2.7 seconds for $m = 64$. LR, in turn, has an average of 1.1 seconds for $m = 4$, and 14.4 seconds for $m = 64$. The small dot for each bar in (b) indicates the standard deviation and it shows that LP execution time is less stable between instances than GR and LP (note the dot above average for LP when

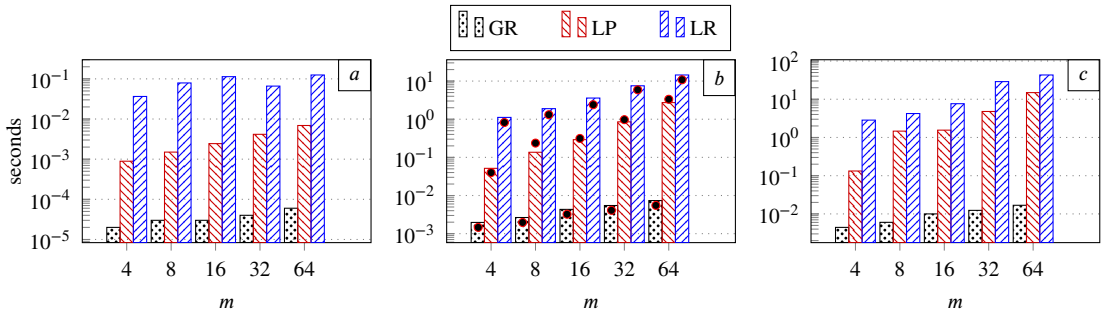


Figure 4.3: *Execution time for GR, LP, and LR. (a) shows the minimum execution time, (b) the average, and (c) the maximum execution time for all J and M queries.*

$m >= 8$, considering the log scale). The maximum execution time for $m = 64$ for GR, LP, and LR are 16.8 ms, 14.9 s, 43.1 s, respectively.

As described in the introduction of this chapter, for *ad-hoc* queries, the time to optimize the query has to be smaller than the time to execute it. In this scenario, the method of optimization can be chosen based on the expected runtime of the query, estimated based on its processing costs. For small *ad-hoc* queries the only reasonable option is GR.

If we focus on system throughput, LR appears as the most promising candidate, although a strategy to amortize its footprint may be worthwhile. One option is to cache and reuse the execution plan after the query optimization, a common strategy used for repetitive or stored queries in non-spatial DBMSs [37]. Other options to reduce the optimization time exist, considering that we can efficiently parallelize LR by splitting the execution of the m Knapsack instances in each iteration. The same applies only partially to LP. Although there exist parallel versions of the Simplex algorithm, their effectiveness is not always guaranteed as it relies on the problem structure [41].

4.11.4 Performance of SM Schedules in FM

This section evaluates how a solution for SM performs when considered as a solution for FM. We recall from Section 4.5 that a feasible solution for SM is also feasible for FM. The difference is that to compute the communication costs in FM, the objective function sums the cost $\hat{c}$ and $\bar{c}$ only once for each machine in which a job is executed. In this evaluation, we substitute the solution x given for SM in the FM objective function and compared the value obtained with a lower bound for FM.

Similar to what we did for the evaluation in Section 4.11.2, we computed the lower bound Z^{lb}_{FM} for the optimal value Z^{*}_{FM} by processing FM in CPLEX and leaving the MIP solver to run from the root node. The gap was computed using $(Z^{+}_{FM} - Z^{lb}_{FM})/Z^{lb}_{FM}$, where the $+$ signal indicates the method, e.g, Z^{GR}_{FM}.

Figure 4.4 shows five charts, for $m = (4, 8, 16, 32, 64)$. Each query has a bar with three marks, indicating the gap for GR, LP, and LR. Comparing the methods, LR presented

the smallest gaps of the three. For $m = 4$, gaps for LP and LR are similar (note the triangle inside the square). This behavior starts to degenerate for $m = 8$ and becomes marked after $m = 16$. GR presented the smallest gaps for J_3 when $m = (8, 16)$, and a gap better than LP for some queries (e.g., M_1 for $m >= 16$). For the other instances, however, the method provided the worst gaps. Average for gaps of GR ranged from 22.8% ($m = 4$) to 48.1% ($m = 64$) when considering all queries. The range for LP was from 12.8% ($m = 4$) to 30% ($m = 64$), and for LR from 11.5% ($m = 8$) to 16.9% ($m = 32$). The gaps for GR and LP increased with m, as can be seen in Table 4.4. LR gaps also increased but to a smaller extent.

The charts in Figure 4.4 show that, in general, the solutions provided by a method for SM were at least 11% more costly than a possible solution for FM. However, as we measured the gap with the lower bound, it does not mean that a solution with $Z^{lb}_{FM} \approx Z^{*}_{FM}$ exists. Computing the gap with Z^{*}_{FM} would be better, but it is also impracticable with a

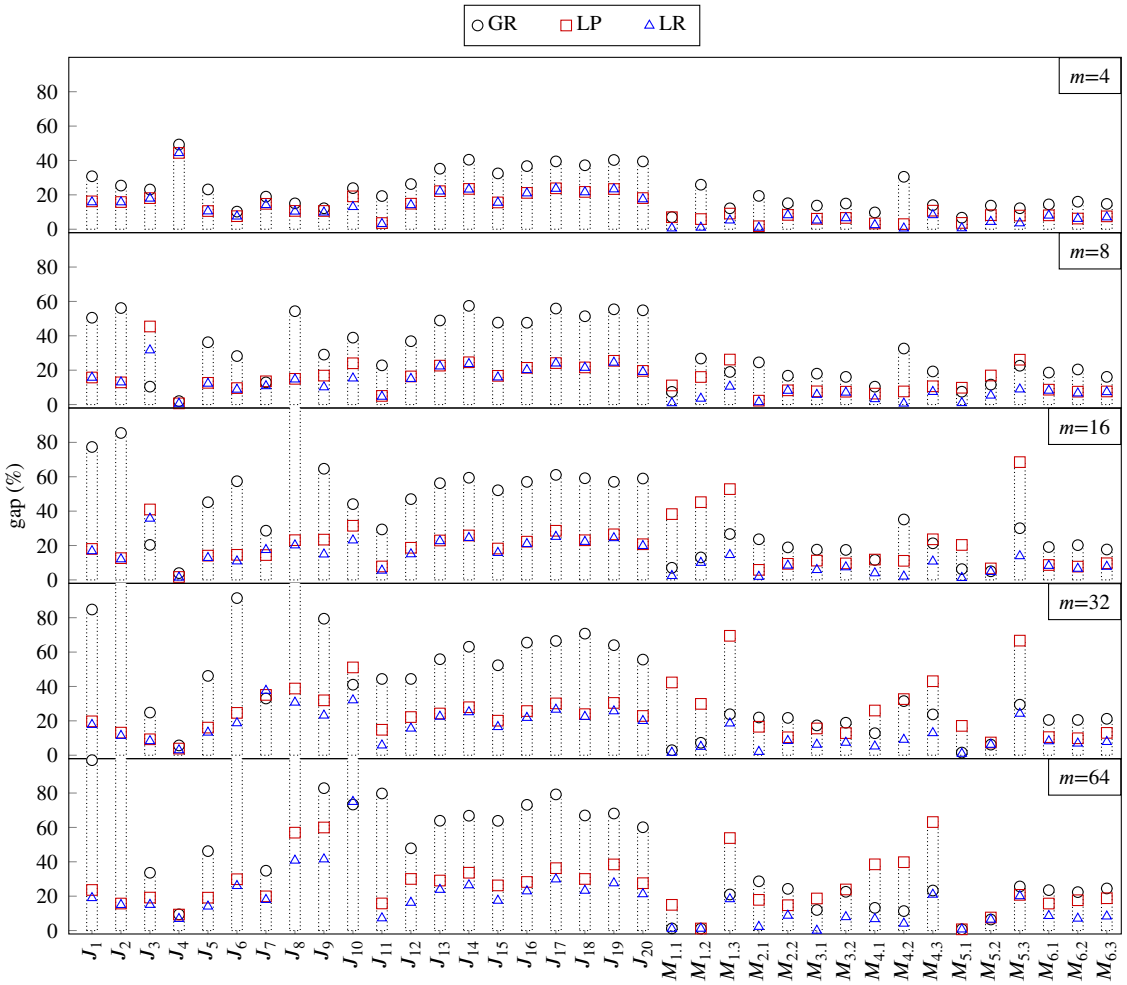


Figure 4.4: *Gap between each schedule provided by GR, LP, and LR and a know lower bound of* Z^{*}_{FM}*, for* $m \in \{4, 8, 16, 32, 64\}$.

Table 4.4: *Average and standard deviation for gaps in Figure 4.4.*

	Average for m					Standard deviation for m				
Method	**4**	**8**	**16**	**32**	**64**	**4**	**8**	**16**	**32**	**64**
GR	22.8	30.1	37.7	42.2	48.1	24.4	38.8	38.8	11.2	17.0
LP	12.8	15.1	20.8	25.3	30.0	14.0	26.9	26.9	8.6	8.8
LR	11.5	11.4	13.1	14.7	16.9	8.3	14.3	14.3	9.2	7.9

generic solver. For instance, when $m = 64$, $M_{6.1}$ and J_2 took more than four and five hours, respectively, only to compute the relaxation for the root node of the problem.

Let us illustrate the additional resource consumption that these gaps can cause for a query, by assuming that an optimal solution exists such that $Z^{lb}_{FM} = Z^{*}_{FM}$ and providing an upper bound on the improvement of communication cost. Although the makespan may change when we solve FM to optimality, it should remain somewhat stable for a typical instance. Thus, the communication cost can be reduced by at most $\delta \approx Z^{LR}_{FM} - Z^{lb}_{FM}$ in the optimal LR solution under these assumptions. As the communication cost is measured by the number of points and a point occupies 32 bytes, we can estimate the additional network traffic. For instance, $M_{6.1}$ has a gap of 20.5%, with $Z^{lb}_{FM} = 5.85904\text{e+}8$ and $Z^{LR}_{FM} = 6.34053\text{e+}8$, thus $\delta * 32/2^{30} \approx 1.43$ GB, a significant amount of additional traffic for only one step of a multiway query.

Finally, let us discuss the trade-off between running time and solution quality for the three methods studied. In general, GR took a few milliseconds to run but presented solutions with poor quality. For instance, consider the larger gap often given by GR in this experiment (depicted in Figure 4.4). Although LR has a higher running time, it provided high-quality solutions. LP, in turn, has intermediate speed and quality. Thus, recalling the running time of LR in the tested instances (average of 15s and maximum of 43s) and the quality of its solutions, we recommend the use of LR observing the often large running time of spatial queries with mid-size or large datasets.

4.12 Broader Applicability of FM

The model proposed here is suitable for the scheduling of jobs composed of data partitions previously distributed in a number of interconnected machines, which need to be aligned to compute a predicate or process an algorithm (θ). We need to know in advance, or through estimates, the processing costs of θ and the network traffic caused by the alignment of the data partitions.

This definition takes into account the processing of multiway spatial join queries, as it is the main application described in this thesis, but can be generalized for other kinds of distributed systems as well. Two of these systems that have recently gained much

attention in the research community are MapReduce frameworks [24] and its in-memory and more general counterpart, Spark engine [92].

The jobs in these systems are characterized by key-value pairs produced by a *map* function. Each key can be reported by several machines. Each "key slice" produced by a machine needs to be aligned with other slices of the same key to be processed by a *reduce* function. The reduction produces the desired result by applying the predicate or algorithm (θ, as we defined here). The default scheduler is known to perform a balanced execution when the data being processed produce keys with a uniform load. However, skewed jobs present a significant challenge [49], and a load balancing mechanism is necessary to mitigate the effect of uneven work allocation [26].

Moseley et al. [60] and Verma et al. [86] recently investigated the scheduling of MapReduce jobs. Moseley et al. [60] proposed the scheduling of MapReduce jobs as a generalized version of the classical two-stage flexible flow-shop problem with identical machines [51]. They provided a 12-approximate algorithm for the offline problem of minimizing the total flow time, which is the sum of the time between the arrival and the completion of each job. Verma et al. [86] investigated the scheduling of jobs using the same flow-shop problem, but with a different objective of minimizing the maximum completion time for a set of jobs. Both studies ignored data transfer between machines. However, in MapReduce jobs, intermediate data is transferred from the map to the reduce machines and thus, the network bandwidth causes a significant bottleneck, making data locality an important issue to be considered.

As we modeled data locality in FM, and as the size of MapReduce jobs seems to support the time required to run the algorithms we proposed, as a future work we hope to apply and evaluate our algorithms within the scheduler of such systems.

4.13 Final Considerations

In this chapter, we dealt with the assignment of jobs to machines in a locally distributed system. We considered a set of jobs defined by data partitions from two datasets that are aligned by a spatial predicate when processing a multiway spatial join query. We introduced a multi-objective linear integer model for the problem, called FM, that considers the minimization of both makespan and communication cost as objectives. We discussed the difficulty of solving it and presented a simplified model, SM, for which we introduced algorithms based on two combinatorial methods: the well-known Linear Relaxation and the more sophisticated Lagrangian Relaxation.

The method based on Lagrangian Relaxation (LR) provided better solutions when compared with the Linear Relaxation method (LP), as well as with a greedy algorithm (GR). Although LR often needs significant time to compute a solution, we presented

scenarios for which it may be applied by observing how close the approximate solutions are to the optimum and the resource consumption improvement that it achieved. LP and GR are recommended for scenarios where small *ad-hoc* queries are predominant.

The number of histogram cells, and the intersection of the dataset's extent area, determine the number of jobs in the instances. The number of jobs, in turn, is a major component of the complexity of the studied methods. Although we can reduce the number of partitions to reduce the optimization time by concatenating histogram cells, it may cause worse estimates and also more skewed partitions, which often results in an unbalanced query execution. Also, a larger number of partitions increases the opportunity for using parallelism, which is important when using larger clusters and processing large queries.

Regarding the number of machines, we saw that its increase causes worse schedules when using LP and GR as well as longer computational times with LR. As the number of machines considered here can be thought of as moderate, compared to today's clusters with hundreds of machines, improving the methods in this respect will be pertinent in future research.

Furthermore, besides the number and the characteristics of the tested instances, we recall that we used an initial data distribution given by a round-robin algorithm. Other initial data distributions can present distinct behavior and should be investigated. However, as the LR method takes the parameter f into account, we believe that it should provide schedules with low resource consumption even for other data distributions.

Notwithstanding, even the schedules computed by LR presented a significant gap for FM, indicating that there is room for improving the scheduling even further. We illustrated through an estimate how the network traffic might reduce if we find near-optimal solutions for an instance. According to our computational experience, even the Linear Relaxation cannot be directly applied in reasonable time (recalling the prohibitive time to compute the root relaxation for some queries). Further research is needed to discover an exploitable structure, such that we can apply the Lagrangian Relaxation or other technique to FM. We regard this as future research.

Finally, a remaining question is how to determine a value for the parameter f. As it is non-trivial to find a suitable value for it, in the next chapter we present a Parametric Analysis for f and show how to compute the format of the objective function.

CHAPTER 5

Controlling the Consumption of Computational Resources in Query Scheduling

As discussed in Chapter 4, when defining a schedule to process a query we want to control the scheduling behavior concerning the usage of computational resources, i.e., processing and network capacity. Our evaluation showed that attempting to minimize both processing and network utilization creates conflicting objectives, in the sense that to achieve a better balance in the query execution (and consequently a reduced makespan), an extra cost is incurred to transfer partitions to machines that are underloaded. To address this issue, we introduced a parameter in the models (SM and FM), termed f, to specify the degree of preference for a more balanced execution or for a lower usage of network resources.

From a practical perspective, if the only concern for query execution is the communication cost, selecting a value for f is trivial (f=0). Analogously, if the only concern is the makespan, we can readily select a sufficiently large value for f so as to ignore the communication cost in the objective function. Often, however, the computational environment imposes constraints in both makespan and communication costs, and selecting a value for f that achieves a balance between these two objectives results in a reasonable saving of computational resources, as we have shown in the previous chapter. Each numerical instance of the models FM and SM has a particular given value for f but determining a suitable value of f is a non-trivial task, as it depends on the values of the processing and communication costs (w_j, $\hat{c}_{ia}$, and $\bar{c}_{ib}$).

In this chapter, we address the question of determining an appropriate value for f by studying the effects of variations on this value on the optimal schedule for SM. We demonstrate how to determine intervals of f values for which an optimal schedule for SM remains unchanged. By establishing these intervals, we can identify the shape of the objective function ((SM.1) in Chapter 4) and provide answers to some practical questions that arise when scheduling queries using the SM model, such as: *i*) is it possible to reduce the makespan of a query even further?, *ii*) if yes, what is the minimum makespan and the additional communication cost incurred?, *iii*) analogously, what is the increase in

makespan if we have to reduce the communication cost on a low bandwidth network?, and iv) what is the maximum value for f for which a change in makespan occurs?

For linear programming, a fully-developed theory exists to determine valid intervals for the parameters of a model, for which an optimal schedule remains unchanged. It is known as post-optimality analysis [6]. This theory, however, is not valid for integer linear programming, and it turned out that a similar theory for the integer case is inherently more challenging and yet not fully developed [8]. Notwithstanding, by adapting the concepts on post-optimality analysis for linear integer programs presented in the classic paper by Geoffrion and Nauss [36], it is theoretically possible to conduct a parametric analysis (PA) of f that produces a sequence of optimal schedules for an SM instance for every possible $f \geq 0$.

As it is expensive to find optimal schedules for practical SM instances, we assume that this should be done as few times as possible during the PA process. Also, we show that by using approximate schedules for SM we can define an upper bound for the value of the objective function. Although this does not furnish a complete PA, it provides valuable information when finding optimal schedules is either quite difficult or very expensive.

The chapter is organized as follows. The introduction for PA is given in Section 5.1. Section 5.2 presents how to obtain the range of f values for the PA and is followed by a practical illustration for a simple numerical instance (Section 5.3). Next, we introduce useful results for the PA in Section 5.4 and how to get bounds on the objective function of SM in Section 5.5. Section 5.6 presents the PA process, i.e., the general description of the steps to conduct the PA and determine the shape of the objective function. Section 5.7 illustrates the PA process via a numerical example of a modified version of a practical multiway spatial join query, with integer w_j's and c_{ij}'s. Section 5.8 describes how to compute an upper bound for the objective function by using approximate schedules for SM. Some final considerations are presented in Section 5.9.

5.1 Introduction to Parametric Analysis

Parametric Analysis (PA) is the study of a family of optimization problems, either having the same structure but differing only by the values of one or more parameters (or coefficients) or having different, but related structure [36]. By carrying out a PA for a problem, we are interested in discovering whether or not a change in some parameter implies a change to a previously-computed optimal schedule. Often, there exist valid ranges for a parameter for which an optimal schedule remains unchanged. By exploring these ranges, we may discover all the optimal schedules for a family of problems, incurring in an effort less than directly proportional to that required to solve a single member.

The most common PA analyses are concerned with the variation of coefficients in the objective function or of coefficients in the right-hand side (r.h.s.) of the constraints. Here we confine ourselves in carrying out a PA for the f parameter in the objective function of SM due to our interest in controlling the consumption of resources, as explained in the introduction of this chapter. There could be analogous analyses of the c_{ij} and w_j values as well, but often there is a huge number of c_{ij} parameters and a change in any of the w_j parameters may result in a change of the set of feasible schedules, which is a major complication.

To start the parametric analysis of f, consider the SM model given in Chapter 4, reproduced here as (SM.1) – (SM.4):

$$Z_{SM} = \text{Min } f x_0 + \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}, \tag{SM.1}$$

$$\text{s.t. } \sum_{i=1}^{m} x_{ij} = 1, \qquad j = 1, \ldots, n \tag{SM.2}$$

$$\sum_{j=1}^{n} w_j x_{ij} + u_i \leq x_0, \qquad i = 1, \ldots, m \tag{SM.3}$$

$$x_{ij} \in \{0, 1\}. \qquad j = 1, \ldots, n;\ i = 1, \ldots, m \tag{SM.4}$$

Let us relabel Z_{SM} as Z, for short. To facilitate the PA of f, we consider that Z, x_0 and the x_{ij}'s are all functions of f. From (SM.1), let $Z(f) = M(f) + C(f)$, where:

$M(f) = f x_0$, the contribution to $Z(f)$ of the makespan x_0, and

$C(f) = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$, the contribution to $Z(f)$ of the total communication cost.

For the sake of an efficient PA, we assume throughout that the w_j's are integers in any SM instance. This assumption is not unrealistic for practical SM instances, as the parameters that result from the probability formulas of the cost model (Chapter 3) can be safely ignored or rounded if there is uncertainty in the estimates.

The following proposition is useful in the PA process and arises from the fact that f is confined in the objective function:

Proposition 5.1. *Changes to f do not affect the set of feasible schedules for SM.*

Proof. As f appears only in the objective function (SM.1), varying f changes only the total cost $Z(f)$ and not which schedules are feasible, i.e., the ones obeying (SM.2) and (SM.4). □

5.2 Finding Bounds for PA

In this section, we find a suitable range of f values for the PA. A lower bound for f arises from the fact that f is a time–cost conversion factor, and thus, $f \geq 0$. When $f=0$, the first expression in (SM.1), i.e., $M(f)$ disappears. This makes constraint (SM.3) irrelevant, and it can be dropped. SM then becomes SM_0 ($f=0$), given by (SM_0.1), (SM.2), and (SM.4), which can be solved trivially by greedily assigning each job to its lowest cost processor.

$$Z_{SM_0} = \min \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}, \tag{SM_0.1}$$

$$\text{s.t.} \sum_{i=1}^{m} x_{ij} = 1, \qquad j = 1, \ldots, n \tag{SM.2}$$

$$x_{ij} \in \{0, 1\}. \qquad j = 1, \ldots, n;\ i = 1, \ldots, m \tag{SM.4}$$

Upper and lower bounds on x_0, the makespan, are useful when determining an upper bound for f, as we shall see shortly. An upper bound on the makespan x_0 for any feasible schedule for SM, denoted here by x_0^{UB}, can be determined by computing the makespan of the schedule obtained by solving SM_0. A lower bound on the makespan, denoted as x_0^{LB}, can be obtained from the expression (5-1) below. If $x_0^{LB} = x_0^{UB}$, the PA process can be terminated as the schedule identified when solving SM_0 is optimal for all $f \geq 0$.

$$x_0^{LB} = \left\lceil \frac{\sum_{j=1}^{n} w_j}{m} \right\rceil \tag{5-1}$$

Furthermore, we can identify an upper bound on f, denoted by f', that establishes the range for conducting the PA for SM. We want to find the minimum value of f' such that it is unnecessary to consider values of $f \geq f'$. That is, we want to find the value f' with the characteristic that the optimal schedule for any $f \geq f'$ is the same as the one for $f=f'$. A schedule obtained for f' corresponds to the lowest possible makespan of any feasible schedule.

To find f', we solve the following family of problems that are versions of SM in which the makespan is fixed at $x_0 = x_0^{LB}$ and then progressively incremented by one unit at a time. The makespan x_0 is no longer a decision variable, and $M(f)$ in SM can be dropped. Furthermore, the right–hand side of (SM.3), can be replaced by $x_0^{LB} + q$ in (SM_q.3), where the parameter q is incrementally increased until two feasible schedules with the same overall total cost are identified. Then we have the family of models SM_q ($q = 0, 1, \ldots$) given by (SM_q.1) – (SM_q.4), which constitute a special case of the Generalised Assignment Problem, GAP [85], a known NP-Hard problem.

$$Z_q(q) = \text{Min} \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij}, \qquad (\text{SM}_q.1)$$

$$\text{s.t.} \sum_{i=1}^{m} x_{ij} = 1, \qquad j = 1,\ldots,n \qquad (\text{SM}_q.2)$$

$$\sum_{j=1}^{n} w_j x_{ij} + u_i \le x_0^{LB} + q, \qquad i = 1,\ldots,m \qquad (\text{SM}_q.3)$$

$$x_{ij} \in \{0,1\}. \qquad j = 1,\ldots,n;\ i = 1,\ldots,m \qquad (\text{SM}_q.4)$$

To find f', we solve SM_q for successive values of $q = 0, 1, \ldots$ until two distinct, feasible schedules have been identified. Note that some SM_q instances may not have a feasible schedule corresponding to certain values of q. Suppose that the first two feasible schedules found correspond to $q = q_1$ and $q = q_2$ with $x_0^{LB} \le x_0^{LB}+q_1 < x_0^{LB}+q_2$. Hence, the first (second) schedule corresponds to makespan $x_0 = x_0^{LB} + q_1$ ($x_0 = x_0^{LB} + q_2$) with total communication cost $Z_q(q_1)$ ($Z_q(q_2)$) where $Z_q(q_1) > Z_q(q_2)$. Exceptionally, when $Z_q(q_1) = Z_q(q_2)$, there is only one feasible schedule with $f' = 0$. The PA for this case is trivial, and so henceforth we assume that $q_1 < q_2$.

We wish to find the lowest f' so that the total cost $Z(f')$, as defined by (SM.1), of the two feasible schedules found are equal. This occurs when:

$$Z(f') = (x_0^{LB} + q_2)f' + Z_q(q_2) = (x_0^{LB} + q_1)f' + Z_q(q_1). \qquad (5\text{-}2)$$

Solving for f', we have:

$$f' = \frac{Z_q(q_1) - Z_q(q_2)}{q_2 - q_1}. \qquad (5\text{-}3)$$

For $f \ge f'$, the schedule with makespan $x_0 = x_0^{LB} + q_1$ and total communication cost $Z_q(q_1)$ is optimal. Hence $f = f'$ is a suitable upper bound for f, and it is thus sufficient to conduct the PA in the range $0 \le f \le f'$. The models in [36] introduce a fractional parameter θ and instead of a PA over the range $0 \le f \le f'$, the authors conduct an equivalent PA over $0 \le \theta \le 1$. Such a transformation is unnecessary for SM, and our PA remains over $0 \le f \le f'$.

In the next section, we provide a simple numerical example to aid the understanding of the proposed method to find f'. The method is also used to produce f' for the SM instance given in Section 5.7.

5.3 Bounds for a Simple Numerical Example

In this section, we illustrate the ideas on Section 5.2 via the following numerical example. Consider the SM_q instance with $m = 4$, $n = 5$, $w_1 = 3$, $w_2 = 3$, $w_3 = 3$, $w_4 = 5$, $w_5 = 6$, $u_i = 0$, $i = 1, \ldots, 5$; and the c_{ij}'s given in Table 5.1.

Table 5.1: *The communication costs for the SM instance.*

c_{ij}	**1**	**2**	**3**	**4**	**5**
1	0	0	300	0	0
2	0	0	200	0	0
3	0	0	100	0	0
4	0	0	300	0	0

Using (5-1), we have that:

$$x_0^{LB} = \left\lceil \frac{\sum_{j=1}^{n} w_j}{m} \right\rceil = \left\lceil \frac{20}{4} \right\rceil = 5.$$

In solving the instances SM_q for $q = 0, 1, 2, 3$, that is, for $x_0 = 5, 6, 7, 8,\ldots$, it is easily seen that, as $w_5 = 6$, there is no feasible schedule for $q = 0$ ($x_0 = 5$). Furthermore, there is no feasible schedule for $q = 2$ ($x_0 = 7$) either. Let $p(j) = i$ if the job j is assigned to processor i, $j = 1, \ldots, n$. The vector of $p(j)$'s is denoted by p.

The first feasible schedule occurs for $q_1 = 1$, with $x_0 = x_0^{LB} + 1 = 6$, $p = (1, 2, 2, 3, 4)$ and the total communication cost $Z_q(1) = 200$. The second feasible schedule occurs for $q_2 = 3$, with $x_0 = x_0^{LB} + 3 = 8$, $p = (1, 2, 3, 3, 4)$ and the total communication cost $Z_q(3) = 100$. Using (5-3), we have that:

$$f' = \frac{Z_q(q_1) - Z_q(q_2)}{q_2 - q_1} = \frac{200 - 100}{3 - 1} = 50.$$

Therefore, for $f \geq 50$, the schedule: $x_0 = 6$, $p = (1, 2, 2, 3, 4)$, with communication cost 200 and total cost $6f + 200$ is optimal. Thus, the PA can be carried out for this instance within the range $0 \leq f \leq 50$.

5.4 Useful Results for PA

PA involves finding optimal schedules (and their costs) for the problem (SM.1) – (SM.4) for all ranges of f within $0 \leq f \leq f'$. Recall that f' has the property that the optimal schedule for $f = f'$ remains optimal for all $f \geq f'$. We now ask the following question. If an optimal schedule has been found for a particular value of f, for what other values of f will the resulting schedule still be optimal? The following proposition (restated

here regarding SM) is stated by Geoffrion and Nauss [36, p. 459] as Proposition 3.3, is very useful for the PA of SM.

Proposition 5.2. *The optimal value $Z^*(f)$ for $0 \leq f \leq f'$ is piecewise–linear, continuous and concave on its finite domain.*

Proof. Noltemeier [64]. □

Suppose that $Z^*(f)$ is drawn as a function of f. As an example, the graph of $Z^*(f)$ for an SM instance (M_{pa}) is given in Figure 5.1. For the sake of clarity, the axes have not been drawn to scale. Proposition 5.2 implies that $Z^*(f)$ consists of straight–line segments (piecewise linear) joined together (continuous) and, starting from f=0, the segments never increase in slope (concave). The value of f corresponding to a break in the graph of $Z^*(f)$ is termed a *breakpoint*. It can be seen in Figure 5.1 that M_{pa} has six breakpoints, at f=4, 5, 24, 29, 31, and 49. Any SM instance has two distinct optimal schedules at each of its breakpoints f, each with total cost $Z^*(f) = fx_0^1 + C_1 = fx_0^2 + C_2$ where, x_0^1 (x_0^2) is the makespan of the first (second) schedule, and C_1 (C_2) is the total communication cost of the first (second) schedule.

The following propositions for SM based on the rationale behind the theory of the linear equation and the already presented propositions are useful in the PA process:

Proposition 5.3. *The slope of $Z^*(f)$ for each segment is the makespan x_0^*, and the point where each segment crosses the vertical axis (where f=0) is the total communication cost $C(f)$, of the corresponding optimal schedule.*

Proof. $Z^*(f) = M(f) + C(f)$ is a linear function in the slope-intercept form, $y = mx + b$, where mx corresponds to $M(f) = fx_0$ and b corresponds to $C(f)$. □

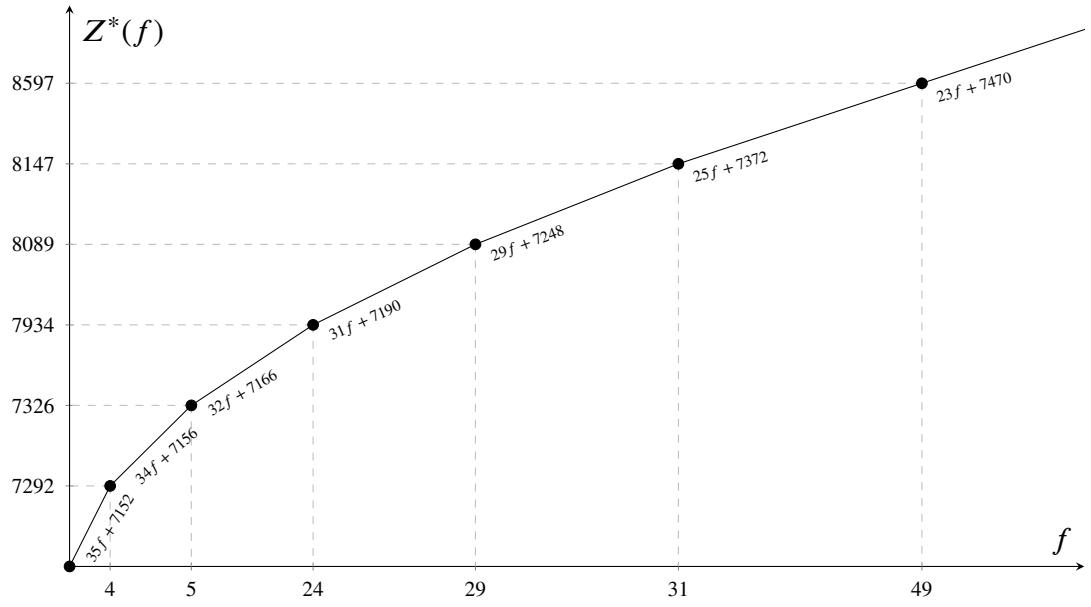


Figure 5.1: *The parametric analysis of M_{pa}.*

Proposition 5.4. *As f increases, x_0^* never increases, and $C(f)$ never decreases.*
Proof. Follows easily from Propositions 5.2 and 5.3. □

Proposition 5.5. *$Z^*(f)$ has a finite number of segments.*
Proof. Elementary, because the feasible region of SM is bounded. □

Proposition 5.6. *The optimal schedule remains the same over each segment.*
Proof. By Proposition 5.4, and observing that an optimal schedule with distinct x_0^* and $C(f)$ causes a new breakpoint on $Z^*(f)$, and consequently, a new segment. □

A natural question to ask is what the segments for $Z^*(f)$ are? The next two sections present a method to find them. We first introduce how to find bounds on $Z^*(f)$ (Section 5.5) and next, in Section 5.6, we show how to obtain the segments for PA.

5.5 Bounds on $Z^*(f)$

We now find bounds on $Z^*(f)$ to be used in the iterative PA process for SM over $0 \leq f \leq f'$. The computation of bounds is illustrated in Figure 5.2. Figure 5.2A shows the points $P_1 = (f_1, Z^*(f_1))$ and $P_2 = (f_2, Z^*(f_2))$ representing the total costs of the known optimal schedules corresponding to $f{=}f_1$ and $f{=}f_2$. Straight lines $Z_1(f) = fx_0^1 + C_1$ and $Z_2(f) = fx_0^2 + C_2$ have been drawn through P_1 and P_2 with slopes x_0^1 and x_0^2 and $Z(f)$ intercepts of C_1 and C_2. Here x_0^1 and x_0^2 are the makespans of the first and second schedules and C_1 and C_2 are the communication costs of the first and second schedules, respectively. The following proposition is helpful.

Proposition 5.7. *An upper boundary of the region of uncertainty of $Z^*(f)$, termed* UB(f)*, is determined by the lower envelope (pointwise minimum) of the (linear) functions $fx_0^r + C_r$ drawn through plotted points representing known feasible schedules, $P_1, \ldots, P_s$, for $r = 1, \ldots, s$.*
Proof. By Proposition 5.1. □

Suppose that the plotted point (f_3,UB(f_3)) is represented by P_4. Applying Proposition 5.7 in Figure 5.2A, UB(f) corresponds to the piecewise linear curve $P_1P_4P_2$.

Proposition 5.8. *A lower boundary of the region of uncertainty of $Z^*(f)$, termed* LB(f)*, is determined by the linear interpolation between the plotted points representing known optimal schedules.*
Proof. By Proposition 5.2. □

Applying Proposition 5.8 in Figure 5.2A, LB(f) corresponds to the straight line P_1P_2. Combining Proposition 5.7 and 5.8, we obtain the shaded area in Figure 5.2A, which represents the region of uncertainty of $Z^*(f)$. Further lower bounds are sometimes available through the use of approximation techniques such as Lagrangian Relaxation.

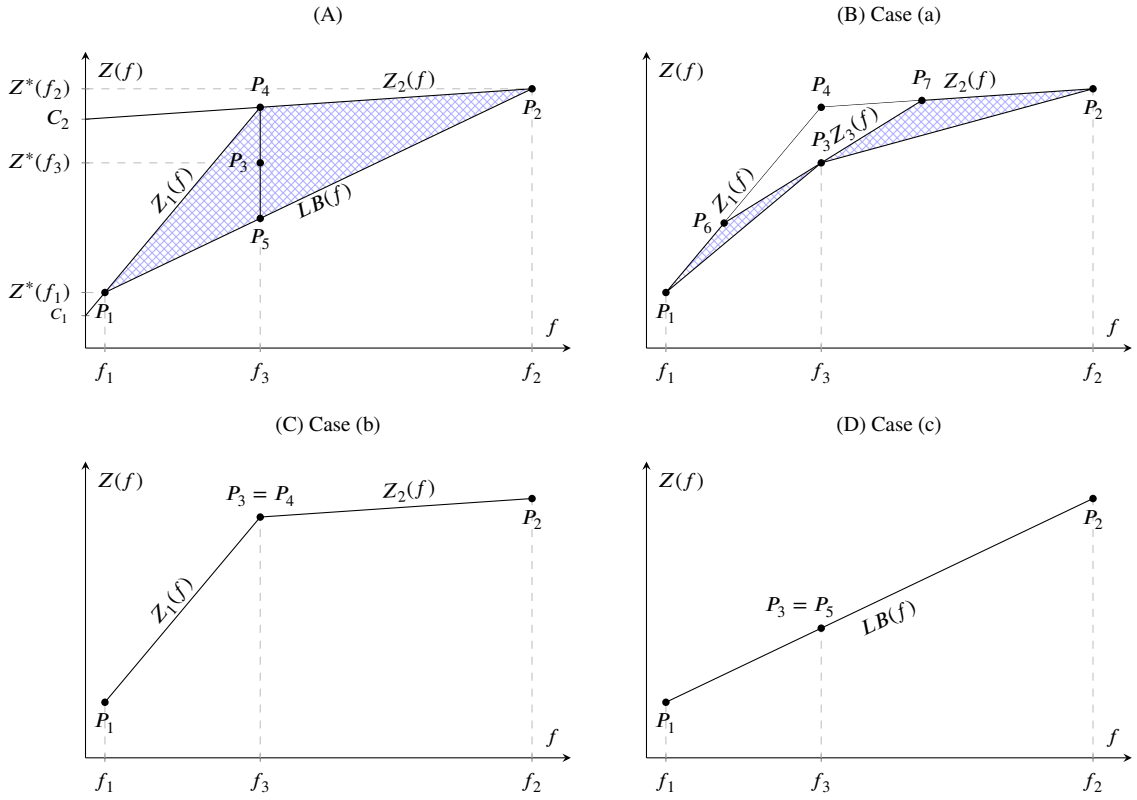


Figure 5.2: *(A) Evaluation of f_1, f_2, f_3. (B) Case (a): $P_3 \neq P_4, P_5$. The PA continues. (C) Case (b): $P_3 = P_4$. A breakpoint at f_3. (D) Case (c): $P_3 = P_5$. No breakpoint at f_3.*

5.6 The PA Process

Following [36], the PA process is based on combining Propositions 5.1 and 5.2. Each iteration of the PA process analyses the feasible schedules corresponding to two different values of f, being f_1 and f_2, where $0 \leq f_1 < f_2 \leq f'$ and a further optimal schedule corresponding to an intermediate value of $f{=}f_3$. Next, we show how to compute f_3 from $Z(f_1)$ and $Z(f_2)$, with $f_1 \leq f_3 \leq f_2$.

The values to start the process are $f_1 = 0$ and $f_2 = f'$. The PA process is applied to attempt to reduce the area of uncertainty (shaded area in Figure 5.2A) by solving the SM instance exactly at an intermediate value $f{=}f_3$. To reduce the area by the largest possible amount, f_3 should be chosen to coincide with $\mathrm{Max}_f(\mathrm{UB}(f) - \mathrm{LB}(f))$ [36]. The value $\mathrm{UB}(f)$ can be computed as the intersection of $Z_1(f)$ and $Z_2(f)$, which is represented by P_4 in Figure 5.2A. Solving for f_3 we have:

$$\begin{aligned} f_3 x_0^1 + C_1 &= f_3 x_0^2 + C_2, \\ f_3 &= \frac{C_2 - C_1}{x_0^1 - x_0^2}, \end{aligned} \tag{5-4}$$

where $x_0^1 > x_0^2$ and $C_2 > C_1$.

The resulting point $P_3 = (f_3, Z^*(f_3))$ is plotted in Figure 5.2A, with $\text{LB}(f_3) \leq Z^*(f_3) \leq \text{UB}(f_3)$, and P_3 lying on the line segment P_4P_5. There are only three possible cases for the position of P_3 in P_4P_5: (a) P_3 is strictly between P_4 and P_5, (b) $P_3 = P_4$, or (c) $P_3 = P_5$. Cases (a), (b) and (c) are depicted in Figures 5.2B, 5.2C, and 5.2D, respectively, and are discussed next.

For case (a), when P_3 is strictly between P_4 and P_5, as shown in Figure 5.2B, the straight line $Z_3(f) = fx_0^3 + C_3$ has been drawn through P_3 with slope x_0^3 and with $Z(f)$ intercept C_3, where x_0^3 is the makespan of the optimal schedule for $f{=}f_3$ and C_3 is its communication cost. Suppose that this line $fx_0^3 + C_3$ intersects P_1P_4 (P_4P_2) at point P_6 (P_7), as shown. Then, by Proposition 5.7, $Z^*(f) \leq Z_3(f)$ and the triangle $P_6P_4P_7$ can be eliminated from the area of uncertainty for $Z^*(f)$. Consequently, this area is reduced to the quadrilateral $P_1P_6P_7P_2$. Furthermore, through Proposition 5.8, the straight lines P_1P_3 and P_3P_2 constitute lower bounds on $Z^*(f)$. This implies that the triangle $P_1P_3P_2$ can be eliminated from the area of uncertainty for $Z^*(f)$. This means that this area is reduced to the two triangles $P_1P_6P_3$ and $P_3P_7P_2$. Thus, the plotted function $Z^*(f)$ passes through P_1 with slope x_0^1, through P_3 with slope x_0^3 and through P_2 with slope x_0^2. It remains to determine the behavior of $Z^*(f)$ in the triangles $P_1P_6P_3$ and $P_3P_7P_3$. This can be performed by recursively applying the PA process to these two triangles.

For case (b), when $P_3 = P_4$, as shown in Figure 5.2C, the area of uncertainty delimited by triangle $P_1P_4P_2$ can be eliminated. There is a breakpoint at $f{=}f_3$, and for $f_1 \leq f \leq f_2$, $Z^*(f)$ can be plotted as the piecewise linear segments P_1P_4 and P_4P_2.

For case (c), when $P_3 = P_5$, as shown in Figure 5.2D, the area of uncertainty delimited by triangle $P_1P_4P_2$ can be eliminated. There is no breakpoint at $f{=}f_3$, and for $f_1 \leq f \leq f_2$, $Z^*(f)$ can be plotted as the piecewise linear segment P_1P_2.

Due to Proposition 5.5, there is a finite set of breakpoints. Furthermore, the proposition below can be used to obviate the need to find an optimal schedule for certain f values.

Proposition 5.9. *Suppose that feasible schedules have been found for f values f_t and f_{t+1}, with makespans x_0^t and x_0^{t+1}, respectively, differing by unity, for some nonnegative integer t. Then there is exactly one breakpoint in $\left[f_t, f_{t+1}\right]$.*

Proof. Due to the fact that the w_j's are assumed to be integers and because of Propositions 5.4 and 5.6. □

In practice, the PA converges quite quickly to the entire set of breakpoints. To further aid the understanding of the process, we illustrated it via the numerical example presented in the next section.

5.7 A Numerical Example of PA

The following example illustrates the PA process applied to a non-trivial SM instance (M_{pa}), which is a modified version of a practical multiway spatial join query instance, with integer w_j's and c_{ij}'s, $n = 69$, $m = 16$, and $u_i = 0$, $i = 1, \ldots, 16$. We modified the values of the processing and communication costs to improve the illustration of the process. The instance is presented in Appendix B for reference and the $Z^*(f)$ function produced by applying the PA process is illustrated in Figure 5.1.

The UB of the makespan x_0^{UB} is obtained by setting f=0 and solving SM_0 to identify the feasible schedule. The schedule can be generated by greedily assigning each job to its lowest cost processor. The result is a feasible schedule with makespan $x_0^{UB} = 35$ and communication cost 7152. Using (5-1), we identify the lowest possible makespan in any feasible schedule, x_0^{LB}:

$$x_0^{LB} = \left\lceil \frac{\sum_{j=1}^{n} w_j}{m} \right\rceil = \left\lceil \frac{362}{16} \right\rceil = 23.$$

We first solve the instances SM_q for $q = 0, 1, 2$, that is, for $x_0 = 23, 24, 25$. The first feasible schedule occurs for $q_1 = 0$, with $(Z^*, x_0^*, C^*) = (8597, 23, 7470)$, i.e., a total cost of 8597, a makespan of $x_0 = x_0^{LB} = 23$, and a communication cost of 7470.

The second feasible schedule occurs for $q_2 = 2$, with $(Z^*, x_0^*, C^*) = (8597, 25, 7372)$, a total cost of 8597, a makespan of $x_0 = x_0^{LB} + 2 = 25$ and a communication cost of $Z_q(2) = 7372$. There is no feasible schedule for $q = 1$, with makespan $x_0 = 24$.

Using (5-3), we have that:

$$f' = \frac{Z_q(q_1) - Z_q(q_2)}{q_2 - q_1} = \frac{7470 - 7372}{2 - 0} = 49.$$

Therefore, the feasible schedule with $x_0 = 23$, communication cost 7470 and total cost $23f + 7470$ is optimal for all $f \geq 49$. So, f' is fixed at 49. As $x_0^{LB} = 23 < 35 = x_0^{UB}$, the proposed PA process is now applied within the range $0 \leq f \leq 49$. The challenge is to find the optimal schedules, its total cost $Z^*(f)$, its makespan and communication cost, for each piecewise segment in this range.

For short, let us denote as $(Z^*(f_i), x_0^i, C_i)$ the values for a feasible schedule obtained for f_i, where i indicates the sequence of f values investigated in the course of PA process. For example, for the first f below, we have $f_1 = (7152, 35, 7152)$.

Recall that for each iteration of the PA process we start with feasible schedules for two values of f and compute a further optimal schedule corresponding to an intermediate value of f. The results for the first iteration is presented in the following table. We already have two schedules corresponding to $f_1 = 0$ and $f_2 = 49$, with the values in the second

column, for which we define the two points in the (third column). The initial interval of uncertainty for $Z^*(f)$ is the triangle $P_1P_4P_2$, plotted in Figure 5.3A. By applying Proposition (5.7) we obtain UB(f). LB(f) is obtained by applying Proposition (5.8). The intermediate value of f (column interm. f) is obtained using (5-4), and finally, by computing the optimal schedule for it, we can determine the case of the PA to be applied.

f	$(Z^*(f_i), x_0^i, C_i)$	$P = (f_i, Z^*(f_i))$	**UB**(f)	**LB**(f)	**interm.** f
$f_1 = 0$	(7152, 35, 7152)	$P_1(0, 7152)$	$35f + 7152$	$29.49f + 7152$	$f_3 = 22$
$f_2 = 49$	(8597, 25, 7372)	$P_2(49, 8597)$	$25f + 7372$		

The optimal schedule for f_3, presented in the following table, provides a point $P_3 = (22, 7870)$ strictly between P_4 and P_5 (Figure 5.3A), which identifies a case (a). Thus, by applying Propositions (5.7) and (5.8), we define new upper and lower bounds to reduce the area of uncertainty for $Z^*(f)$, as illustrated in Figure 5.3B. The intervals of uncertainty of $Z^*(f)$ are now the triangles $P_1P_6P_3$ (Figure 5.3C) and $P_3P_7P_2$ (Figure 5.3D).

f	$(Z^*(f_i), x_0^i, C_i)$	$P = (f_i, Z^*(f_i))$	**UB**(f)	**LB**(f)
$f_3 = 22$	(7870, 32, 7166)	$P_3(22, 7870)$	$32f + 7166$	P_1P_3: $32.64f + 7152$ P_3P_2: $26.92f + 7744$

The PA process continues recursively in the triangles $P_1P_6P_3$ and $P_3P_7P_2$ until a case (b) occurs and identifies a breakpoint, or a case (c) occurs which implies there is no

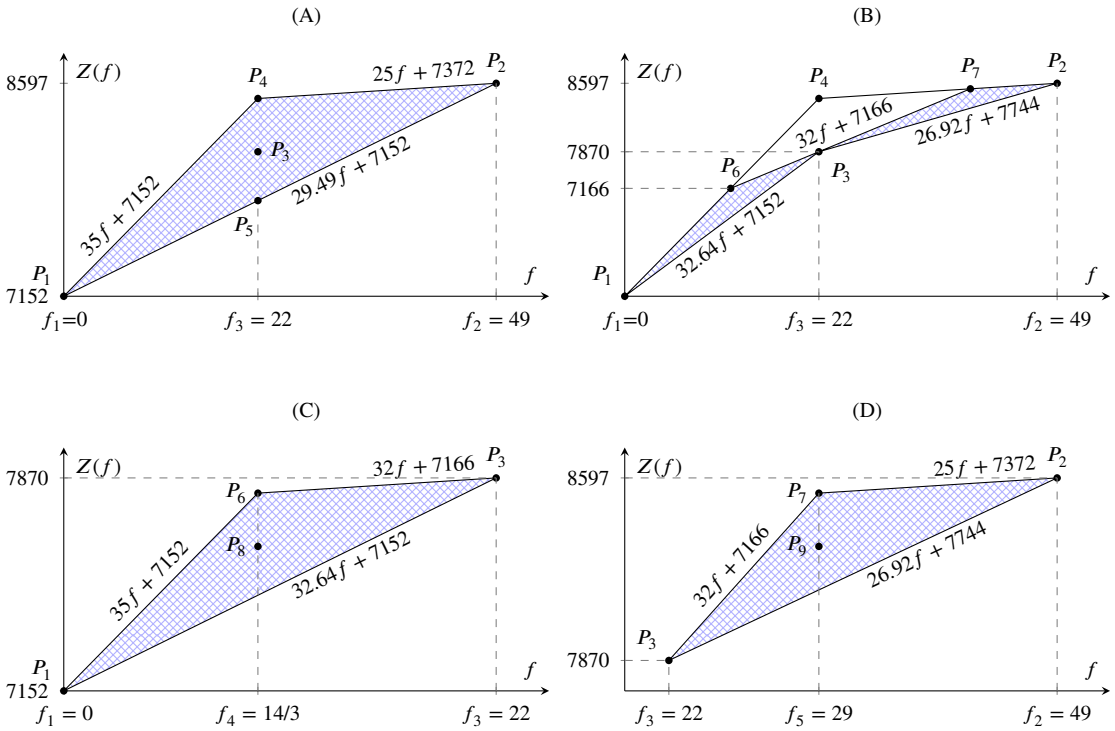


Figure 5.3: *First iteration of the PA for* M_{pa}.

breakpoint in the particular range of f at hand. For the sake of brevity, the other iterations for this PA are presented in Appendix B.

We found optimal schedules for nine values of f, as indicated in Table B.1. This produced a complete PA of f for M_{pa}. There are six breakpoints: f=4, 5, 24, 29, 31, and 49. At each of these breakpoints, there are two distinct optimal schedules. The results are summarized in Table 5.3, and $Z^*(f)$ is presented in Figure 5.1.

From Table 5.3, it can be seen that it is possible to reduce the makespan of M_{pa} by 34.3% (from 35 to 23), with a corresponding increase of 4.4% in the communication costs (from 7152 to 7470). The last improvement of the makespan occurs for f=49. No value of f>49 can further improve it. Furthermore, the communication cost increases smoothly in the first segments and then jumps after the fifth segment ($29 \leq f \leq 31$).

Table 5.2: *The f's examined in the parametric analysis of M_{pa}.*

Breakpoint?	f	$Z^*(f)$	x_0^*	C^*	x_0^*	C^*
No	0	7152	35	7152	-	-
Yes	4	7292	35	7152	34	7156
No	14/3	7314.67	34	7156	-	-
Yes	5	7326	34	7156	32	7166
No	22	7870	32	7166	-	-
Yes	24	7934	32	7166	31	7190
Yes	29	8089	31	7190	29	7248
Yes	31	8147	29	7248	25	7372
Yes	49	8597	25	7372	23	7470

Table 5.3: *A summary of the parametric analysis of f for M_{pa}.*

Range of f	x_0^*	$C(f)$	$Z^*(f)$	**Range of** $Z^*(f)$
$0 \leq f \leq 4$	35	7152	$35f + 7152$	$7152 \leq Z^*(f) \leq 7292$
$4 \leq f \leq 5$	34	7156	$34f + 7156$	$7292 \leq Z^*(f) \leq 7326$
$5 \leq f \leq 24$	32	7166	$32f + 7166$	$7326 \leq Z^*(f) \leq 7934$
$24 \leq f \leq 29$	31	7190	$31f + 7190$	$7934 \leq Z^*(f) \leq 8089$
$29 \leq f \leq 31$	29	7248	$29f + 7248$	$8089 \leq Z^*(f) \leq 8147$
$31 \leq f \leq 49$	25	7372	$25f + 7372$	$8147 \leq Z^*(f) \leq 8597$
$49 \leq f$	23	7470	$23f + 7470$	$8597 \leq Z^*(f)$

5.8 Upper Bound of $Z^*(f)$ Using Approximate Schedules

When optimal schedules are unavailable or are difficult to find, approximate schedules to SM, such as the ones provided by the method based on Lagrangian Relaxation (LR) given in Chapter 4, can be used instead to obtain upper bounds for $Z^*(f)$. In this section, we discuss how to calculate those upper bounds and discuss certain limitations of the process.

A sequence of upper bounds on the region of uncertainty of $Z*(f)$ can be deduced by following a modified PA process that is limited to UB identification, rather than finding exact schedules. We apply Proposition 5.7 on the initial range for f, and next compute the intermediate values of f by (5-4). For each intermediate value of f, instead of getting an optimal schedule, we compute a schedule by an approximate method. The process is repeated in the same way as the original PA process but with a limitation described next. This procedure does not necessarily lead to a complete PA. For instance, it cannot provide the segments of $Z^*(f)$. However, it may still produce valuable, partial results.

The mentioned limitation in the process occurs when considering the resulting point for the schedule of an intermediate value f_3 to decide which case is pertinent in order to continue the PA process. By using optimal schedules, only the three mentioned cases (a), (b), or (c) in Figure 5.2 can occur. By using approximate schedules, however, a fourth case may occur, for which the point defined by the approximate schedule appears above UB(f), that is, the quality of the schedule is relatively bad considering the partial UB already defined. In this situation, the procedure cannot further refine the upper bound in the respective range of f.

We present in Figure 5.4 a computational experience gained by using this procedure in M_{pa}. We plotted in Figure 5.4A both the original shape of $Z^*(f)$ given by the complete PA process and the UB defined by the approximate schedules. The difference is almost imperceptible for some ranges of f, indicating that the UB is close to $Z^*(f)$. We draw the highlighted region in (B). The point P is the value of the approximate schedule for f=22 (7928, 25, 7378), for which the fourth case mentioned occurs. The dotted line is the given upper bound defined by this schedule, and it is outside the region already defined. The method cannot improve further the UB in this region.

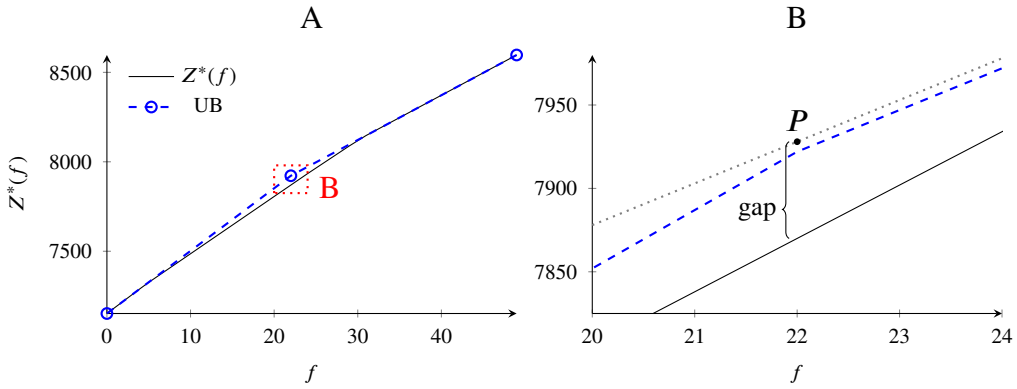


Figure 5.4: *Shape of $Z^*(f)$ for M_{pa} and the upper bound defined using approximate schedules. (B) is the zoomed region highlighted in (A), with the point P of the approximate schedule and its gap.*

We also computed the segments of $Z^*(f)$ and the UB for the practical SM instance J_3. We choose J_3 because it presented the largest GAP for $m = 4$ in the experiments in the previous chapter. Further, for $m = 4$ we can quickly compute the complete segments of J_3. The range for PA in this instance was $0 \leq f \leq 3.62$, and it presented 72 segments. The result is depicted in Figure 5.5. As we can see, the UB is very close to $Z^*(f)$ in (A), and the difference is visually imperceptible. There is a dense region of segments between f=0 and f=1.7, indicated by the markers. We zoomed in on the highlighted regions in (B) and (C). These are the regions with the largest difference between UB and $Z^*(f)$.

We can conclude from this experiment that the defined UB is very close to $Z^*(f)$. This result is due to the small gaps provided by the LR method. Thus, in scenarios where optimal schedules are unavailable or are difficult to find, the UB can be used to answer the practical questions previously mentioned with reasonable precision, provided that good approximate schedules exist.

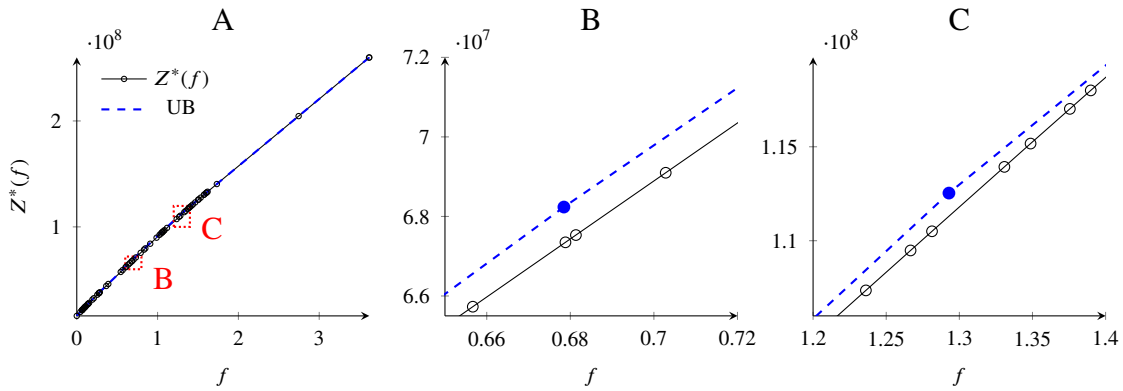


Figure 5.5: *Shape of $Z^*(f)$ for J_3 with $m = 4$ and the upper bound defined using approximate schedules. (B) and (C) are the zoomed regions highlighted in (A).*

5.9 Final Considerations

In this chapter, we investigated a method to control the scheduling behavior concerning the usage of computational resources, i.e., processing power and network capacity, by using the parameter f introduced for SM in Chapter 4. We proposed a method based on the post-optimality theory of integer linear programming, which can identify a range of f and all the segments for which distinct schedules for an instance of SM is produced. This method can provide answers to practical questions that arise when scheduling queries using the SM model and identify a value of f depending on the constraints of processing power and network costs imposed in a system.

Furthermore, in scenarios where optimal schedules are unavailable or difficult to find, we introduced a method that can use approximate schedules to define an upper bound on the value of $Z^*(f)$, termed UB(f). The method can also be used to answer the practical questions previously mentioned with reasonable precision, provided that good approximate schedules exist, such as the ones provided by the LR method (Chapter 4).

In the next chapter, we introduce the last two missing pieces of the distributed multiway spatial query optimizer, namely, the method for selection of plans and the specifications of an execution engine that can perform the selected execution plan for a query, respecting the defined schedule for it.

CHAPTER 6

Selection and Execution of Multiway Spatial Join Query Plans

In this chapter, we introduce the methods for the query optimizer proposed in this thesis and present a query execution engine that can execute multiway spatial join queries following the schedules provided by the optimizer.

We recall from previous chapters that a query optimizer performs the optimization of a query in a sequence of steps. First, the optimizer enumerates the set of equivalent plans for the query (plan enumeration) and next it analyses each of the enumerated plans to estimate the cost of processing the plan, with regard to the specified order determined by the plan and using the cost model (Chapter 3). During the cost estimation, a schedule is generated by determining some of the costs in a distributed environment (Chapter 4 and 5). Finally, the optimizer selects a plan to execute the query, based on the final cost of each estimated plan. This last step is covered in the present chapter.

The chapter is organized as follows. We introduce the query selection method in Section 6.1, and next, we present the query execution engine used in the experiments (Section 6.2). The evaluation of these two components is presented in Section 6.3. In the evaluation, we return to some properties of the cost model, that are dependent on the query scheduling method. One of these properties is the precision of the estimates of the communication cost (Section 6.3.1). The evaluation by the query optimizer, with regard to the selection of good execution plans, is presented in Section 6.3.2, followed by the analysis of the resource consumption in real query execution (Sections 6.3.3 and 6.3.4). The scalability of the query execution engine is evaluated in Section 6.3.5), and finally, we present our conclusions and final considerations in Section 6.4.

6.1 Selection of Distributed Execution Plans

In this section, we propose a method for plan selection. We recall from Chapter 2 that the evaluation of the predicate of a spatial query uses CPU intensive algorithms from Computational Geometry as part of the identification of the result set. Often, the objective

of processing a query in a distributed system is to achieve a short execution time, by sharing the load evenly among the computing resources. In the plan scheduling method, we pursued this objective by scheduling the steps of each plan in a way that minimizes the weighted sum of the makespan and the communication costs. The weight, defined by the f parameter, was used to choose between makespan minimization or a compromise between the makespan and the communication cost. However, each plan step has a particular range of f, and thus, a single value for f is not adequate for all steps of the plan. Due to this, we define a new weight for the entire query, called f^q, $0 \leq f^q \leq 1$. Based on the value of f^q, we can define the particular value of f for each step, observing the range $0 \leq f \leq f'$, as defined in Chapter 5. For example, $f^q = 1$ implies that the maximum f value for each step should be used when scheduling the plan.

To select a plan, we start with the array $plan\_cost$ returned by the procedure Estimate-Plan-Cost (Algorithm 3.5). It contains both the aggregated load for each machine, the steps of a query and the aggregated communication cost. We apply the same reasoning of the objective function Z_{FM} (FM.1), in the sense that we weight the final makespan for the plan by f^q and sum it with the total communication cost. We illustrate it formally in Algorithm 6.1.

It is worth noting that many equivalent query steps arise during the enumeration of plans, i.e., steps that estimate the same sub-query, with the same datasets or intermediate results, and in the same order. Obviously, these steps do not require to be repeatedly estimated, provided that dynamic programming or *memoization* can identify them in the plan enumeration procedure. As the plan enumeration is outside the scope of this thesis, we do not discuss this identification step here and consider this process as a single loop within the planning procedure. In a practical implementation, however, this integration between the plan enumeration and the plan selection methods should be implemented.

The procedure Select-Plan, in Algorithm 6.1, illustrates how to select a plan for a query, by calling the previously-proposed methods. The procedure starts with the set of plans P for the query Q (line 1) and iterating over each plan $P_k \in P$ (line 2) to compute its makespan $P_k^{x_o}$ (line 4), communication cost P_k^C (line 5), and its overall weighted cost P_k^Z. Finally, the best plan is selected by taking the plan with minimum P_k^Z (line 7). The procedure Aggregate-Costs, called by Estimate-Plan-Cost, returns the load field of the $plan\_cost$ array already weighted by the value of f for each step. Also, the function $stepf$ used inside Estimate-Plan-Cost is defined by (6-1).

$$stepf(f^q) = f' * f^q \tag{6-1}$$

Algorithm 6.1: Procedure SELECT-PLAN to select a plan to execute a query Q.

SELECT-PLAN(Q, f^q)

1 P = ENUMERATE-PLANS(Q)
2 **for each** $P_k \in P$
3 ESTIMATE-PLAN-COST(P_k, f^q, $plan\_cost$)
4 $P_k^{x_0} = \max_{i=1}^{m}(plan\_cost.load[i])$
5 $P_k^C = \sum_{i=1}^{m} plan\_cost.comm[i]$
6 $P_k^Z = P_k^{x_0} + P_k^C$
7 **return** $argmin_{P_k \in P}\left(P_k^Z\right)$

6.2 Query Execution Engine

In this section, we present the execution engine used to perform the multiway spatial join queries and some details of our implementation. The components of the execution engine and its capabilities are described as follows. There are three components: the client, the controller, and the worker modules, as illustrated in Figure 6.1.

The Client Module (see Figure 6.1) provides the API (Application Programming Interface) to load datasets into the system and to execute multiway spatial join queries. It is composed of the Data and Query parsers. The *Data parser* sub-module loads the data files in their respective formats, and computes the necessary metadata information, required by the Split method (Chapter 3, Section 3.2), such as the dataset spatial extent and the average length of all objects. The *Query parser* is the sub-module responsible for parsing the query received in a string format and producing a query graph from the input that will be used inside the Controller module, to optimize the query.

The Controller Module (see Figure 6.1) controls the data loading process and performs the query optimization and execution. It maintains the context of the system, which includes information about the queries and jobs to detect the progress of the execution. It also maintains a cache of the metadata required for the data partitions loaded in the system. The Query Optimizer sub-module implements the plan enumeration algorithm, the plan scheduling method (Chapters 4 and 5), the plan selection (Section 6.1), and the algorithms related to the cost model (Chapter 3). The Data partitioner sub-module receives the spatial objects from the Client module, forms the data partitions, and performs the data distribution. The Query Controller sub-module interfaces with all parts of the system and performs the query execution.

The Worker Module (see Figure 6.1) stores the data partitions loaded into the system and their metadata and also performs the execution of jobs over these partitions. An instance of the worker module exists on each machine of the cluster and knows about the existence of the others worker instances in the network. When executing query jobs,

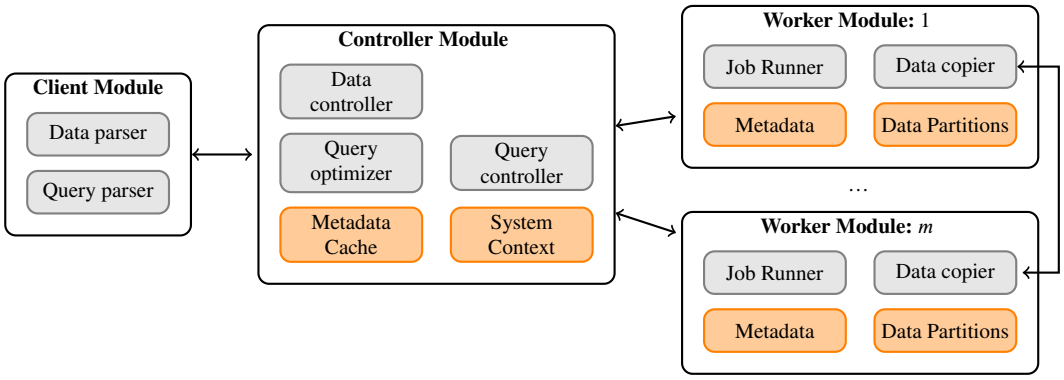


Figure 6.1: *Modules of the execution engine and their interactions.*

the Data copier of an instance may possibly ask other instances for missing data partitions in the local repository that are necessary to execute a query job, including intermediate data partitions, generated while executing a query.

The Worker Module also exploits intra-machine parallelism, i.e., it runs on machines with multiple processors or cores, using threads and shared memory. Besides the distributed partitioning of a query, each query job can be processed in parallel (internally within each machine). This division is simpler than the distributed division, due to use of shared memory. Another important task that is performed (in parallel with the predicate checking) is the copying of data partitions from other workers by the Data copier sub-module.

The processing of query jobs by the Worker Module is illustrated in Figure 6.2. The jobs are received from the Controller module and stored in the Receive queue. A Classifier uses the local metadata to classify the job into three categories: *i*) *ready* to execute, meaning that all data required to execute the job is locally available; *ii*) *waiting for remote data*, i.e., to perform the job, one or more data partitions require to be copied from other workers; and *iii*) *waiting for local data* to be available, i.e., the job requires an intermediate partition that will be generated at the current node, but still is not available.

The *ready* jobs are directly transferred to the Execute queue, where the predicate is performed, and the intermediate result set is generated. Jobs that require additional data are moved to the Wait queue. When the data required becomes available, the job is transferred to the Execute queue. To prevent high memory consumption, the sizes of these queues are defined as parameters of the system, and should be defined with regard to the network capacity, i.e., faster networks should require smaller queues and vice-versa.

The wait queue is a fundamental component of the system as it prevents network contention. In other words, as the scheduling of a query considers the local availability of data to define where each job will be executed, there will be a set of jobs for a query that can be executed without passing through the Wait queue. The other jobs that require local or remote data will pass through the Wait queue. The Data copier sub-module can handle

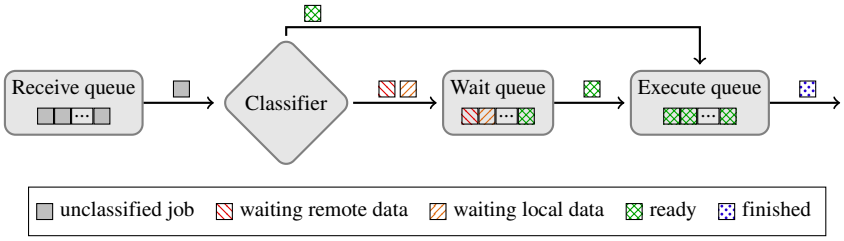


Figure 6.2: *The flow of jobs inside the Worker module.*

the jobs waiting for remote data by copying the remote partitions in parallel, while the CPU continues to process the jobs in the Execute queue. As the predicate is a complex algorithm, the execution queue is expected to have jobs to deal with almost all the time, keeping the machine processors busy. We shall study this behavior in the Evaluation section, given next. In our implementation, we created one wait queue for each machine that is contacted, and the system can copy data partitions from all other machines at the same time.

To implement those described modules, we used two programming languages: C and Go[1]. The C language was used to code low-level algorithms that interface with the GDAL[2] library to load datasets files in ESRI Shapefile (.shp), ESRI FileGDB (.gdb) and GeoJSON formats (Data parser module) and with GEOS[3] library to process spatial predicates in the Job Runner module. The interoperability of C and Go code was implemented using the native CGO extension.

The Go language was used in the distributed part of the system to implement the communication protocols and the parallel join processing on the Worker node. All queues are implemented using the mechanism of *channels* provided by the language. The communication protocols used to provide the interaction between the modules runs over TCP sockets, and the serialization of data structures was implemented using the GOB package of the Go language.

6.3 Evaluation

This section presents the evaluation of the proposed query optimizer. The experiments were performed on the Amazon EC2 Platform, using virtual machines of type m4.xlarge[4]. Each machine has four vCPUs of type Intel(R) Xeon(R) CPU, E5-2686 v4 model, running at 2.30GHz, and with 16GB of RAM. The machines were allocated to the same data center, interconnected by a virtual network. A machine can utilize up to 10 Gbps for single-flow and 20 Gbps for multi-flow traffic in each direction (full duplex). The

[1]http://golang.org
[2]http://www.gdal.org
[3]https://trac.osgeo.org/geos/
[4]These machines are distinct from those used in previous chapters due to constraints in our quota in the cloud environment. However, the total amount of memory is sufficiently large to run all queries in memory.

operating system used was the default Ubuntu Server 16.04 LTS offered by the provider through an AMI image.

We focused on the evaluation of multiway spatial joins, by using the queries $M_{1..6}$ defined in Chapter 4, and extending this set with two larger queries: M_7, that is a multiway join between larger datasets, and M_8, that uses more datasets and has more steps. The additional datasets are presented in Table 6.1. The three datasets prefixed with "Tiger" are from the TIGER 2016 spatial database[5]. Their size ranges from 101 MB to 4.8 GB, and they represent spatial layers in the USA. The other datasets are from the Digital Chart of the World (DCW) and represent polygons of worldwide land coverage. The size column in the table displays the original binary representation of the data: ESRI FileGDB Format for Tiger datasets and Shapefile format for the others. When loaded into the system, the datasets occupy more than three times the space indicated, increasing the required main memory to store the spatial objects, as well as the required network bandwidth to copy data partitions.

Table 6.2 presents all queries used, showing their number of steps, number of plans, and join cardinality. The plan for a query is denoted in the experiment results using the format $M_1^{p_1}$, where p_1 is the first plan for query M_1.

Table 6.1: *Additional datasets used in the experiments.*

Name	Abrev.	Type	Cardinality	File Size
Tiger Block	TBL	Polygons	11,278,412	4.8 GB
Tiger Area Water	TAW	Polygons	2,293,342	823.0 MB
Tiger School	TSC	Polygons	10,896	101.0 MB
Tundra	TU	Polygons	37,792	26.5 MB
Lakes	LA	Polygons	338,819	130.3 MB
Grass	GR	Polygons	145,605	87.7 MB
Trees	TR	Polygons	167,392	95.7 MB
Sand	SA	Polygons	18,866	22.9 MB
City Areas	CA	Polygons	36,432	12.8 MB

Table 6.2: *Multiway spatial join queries used in the experiments.*

Name	Query	Steps	Plans	Join Card.
M_1	A ⋈ RI ⋈ RA ⋈ CR	3	6	104
M_2	RI ⋈ RA ⋈ EC	2	2	46,235
M_3	RI ⋈ HI ⋈ RA	2	2	16,285
M_4	R ⋈ RI ⋈ RA ⋈ EC	3	6	2,254
M_5	A ⋈ HI ⋈ CR ⋈ C	3	6	5,308
M_6	RI ⋈ EC ⋈ HI ⋈ RA	3	6	9,557
M_7	TAW ⋈ TSC ⋈ TBL	2	2	4,398,772
M_8	RU ⋈ LA ⋈ GR ⋈ TR ⋈ RA ⋈ SA ⋈ CA	6	720	8,420

[5]http://www.census.gov/geo/maps-data/data/tiger-geodatabases.html

We used the LR method (Chapter 4) to build the schedules for the queries and f^q=0.5. Some experiments used different values of f^q and also other scheduling methods. We indicate in each subsection when this occurs.

6.3.1 Evaluation of the Communication Cost Estimate

After the scheduling of a query plan, we know which partitions will be copied during the plan execution, and we can estimate the incurred cost by using the cost model proposed in Chapter 3. The experiment presented here compares the estimated and real communication costs, for each query and plan. The real communication cost was measured in the network interface. The purpose of the experiment is to check if the estimated communication cost computed via the cost model for the intermediate steps supports the identification of plans with a lower communication cost. For the first step of a plan, as discussed in Chapter 3, we have the exact size of the partitions in the histograms.

We planned and executed each query plan using the Lagrangian Relaxation method (Chapter 4, Section 4.7), for eight machines ($m = 8$). To convert the communication cost to bytes, we multiplied the given number of points by 32 (two double float points). As the serialization protocol creates a kind of compression[6], we expect the communication cost to be overestimated. However, as it will be overestimated for all plans, we still can identify the ordering of plans.

The results are shown in Figure 6.3. We plotted the values as a line chart to facilitate the comparison of the contour lines for estimated and real values. The markers determine the communication cost for each plan. The total communication cost for each plan of M_1 to M_6 can be seen in (A), which ranges from 64.4 MB ($M_1^{p_5}$) to 2.4 GB ($M_6^{p_3}$). The range of estimated values is 90.5 MB to 3.8 GB, for the same two query plans, respectively. In (B) we depict the values for M_7, and in (C) we present the values for all 720 plans of M_8. By checking (A), (B), and (C), and the similarity of the contour line for estimated and real values, it is evident that the method successfully identified the pattern of the communication cost. As expected, the communication cost was overestimated for all plans but, in general, the ordering of plans with respect to communication cost was mostly preserved (some exceptions can be observed where the peak shown for the estimated value does not follow the same amplitude of the real cost, e.g., around plan p_{200} and p_{500} in (C)).

Figure 6.4 presents the estimated communication cost per machine for queries M_1 to M_6. The error bars indicate the minimum and maximum communication cost for a machine. The chart shows that the communication cost per machine was also well estimated by the cost model, as the error amplitude in the estimated line follows the

[6]The encoding algorithm byte-reverses a float point value and ignores low bits. The details can be checked in golang.org/pkg/encoding/gob/.

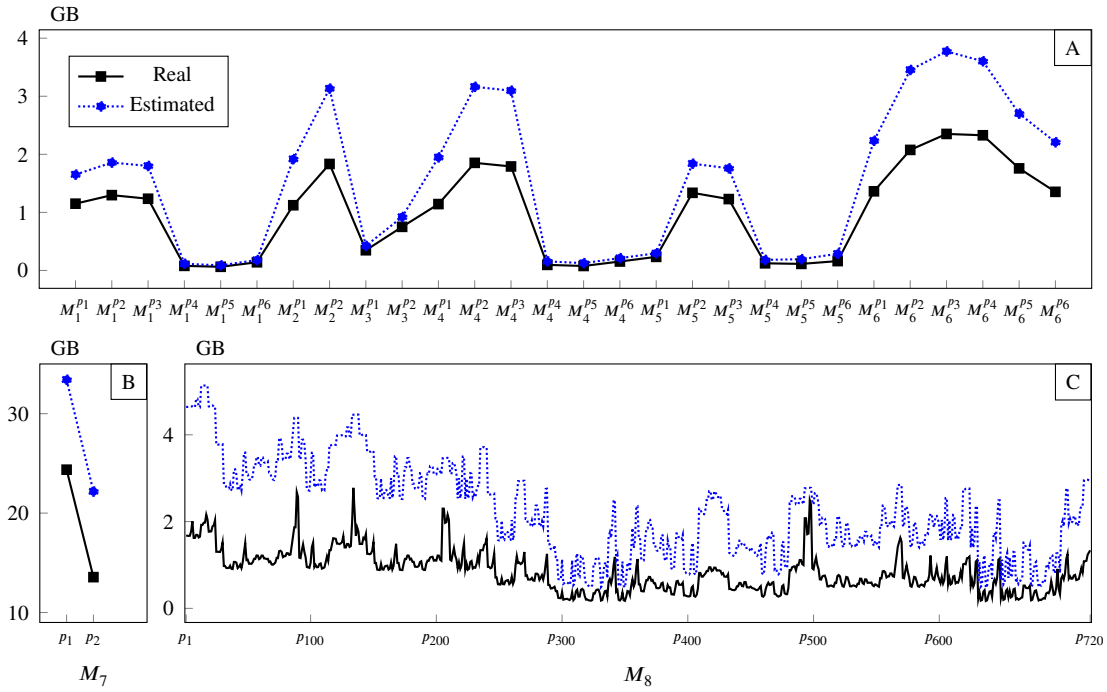


Figure 6.3: *Estimated and real communication costs of multiway queries. (A) shows cost for $M_{1..6}$, (B) for M_7, and (C) for M_8.*

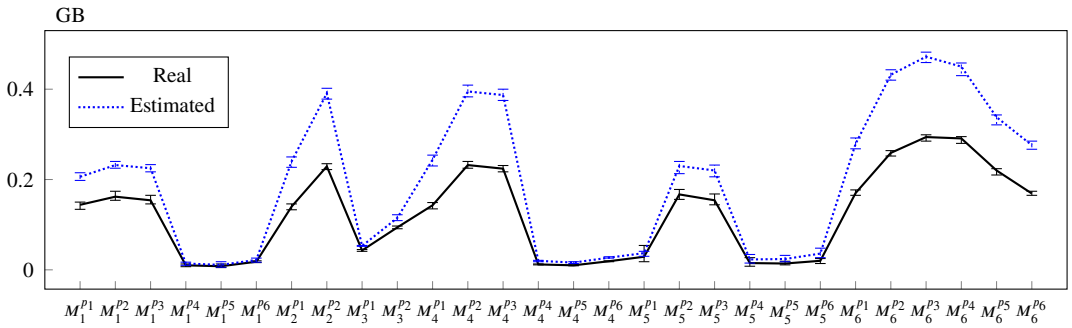


Figure 6.4: *Average communication per machine and error bars showing the minimum and maximum communication costs of all machines for queries M_1 to M_6.*

pattern of the amplitude for real values. Thus, we can conclude that the proposed cost model provided an estimate of the total and per machine network communication cost that followed the pattern of the real data transfer measured when running the query plans on the execution engine, enabling the distinction between plans with high and low network usage.

6.3.2 Evaluation of the Selection of Distributed Execution Plans

In this section, we present the evaluation of the plan selection method introduced in Section 6.1. The experiment consisted of selecting an execution plan using the plan

selection method and then measuring the actual execution time of all plans in the cluster. The purpose of this was to check whether the selected plan (based on the estimates) was indeed the plan with the shortest execution time and smallest communication cost. We also verified whether or not the values of P_k^Z for each query established an ordering of the execution plans equivalent to the ordering of plans based on real execution time.

Figure 6.5 (A) shows the estimated cost of all plans for queries M_1 to M_7. Each bar indicates the value P_k^Z for the plan P_k. The lowest bar in each group indicates the plan with the smallest P_k^Z, i.e., the plan selected by the optimizer. The measured execution time of each plan is presented in (B). The chart uses a logarithmic scale due to the difference between the execution times of the best plan (in the order of seconds) and the worst plan (in hundreds of seconds). The chart shows that the selected plan was indeed the best one for six of the seven queries. M_5 is the only query for which the selected plan (p_4) was not the best one (p_5), but both are very similar plans with regard to execution time (0.3s $\times$ 0.4s). Furthermore, the plan selection method was able to establish a consistent relative ordering of all plans for almost all queries. The only two differences in the ordering are $M_5^{p_4}$ and $M_5^{p_5}$, which appear in inverse order between estimated and real execution, and $M_6^{p_2}$ - $M_6^{p_4}$ which also are similar plans.

The evaluation of query M_8 is depicted in a separate chart due to its number of plans (Figure 6.6). We plotted the contour line of the value P_k^Z for each of the 720 plans, as well as the contour line for the execution time. The two values were normalized. This large query led us to a more realistic evaluation of the method since the query has many plans that are very similar with regard to resource consumption. We can split the horizontal axis into six major groups of plans with regard to the execution time (check the vertical lines): (1) a set of bad plans from p_1 to p_{118}, (2) a set of slightly worse plans from p_{119} to p_{239}, (3) a set of good plans from p_{240} to p_{357}, (4) another set of good plans from

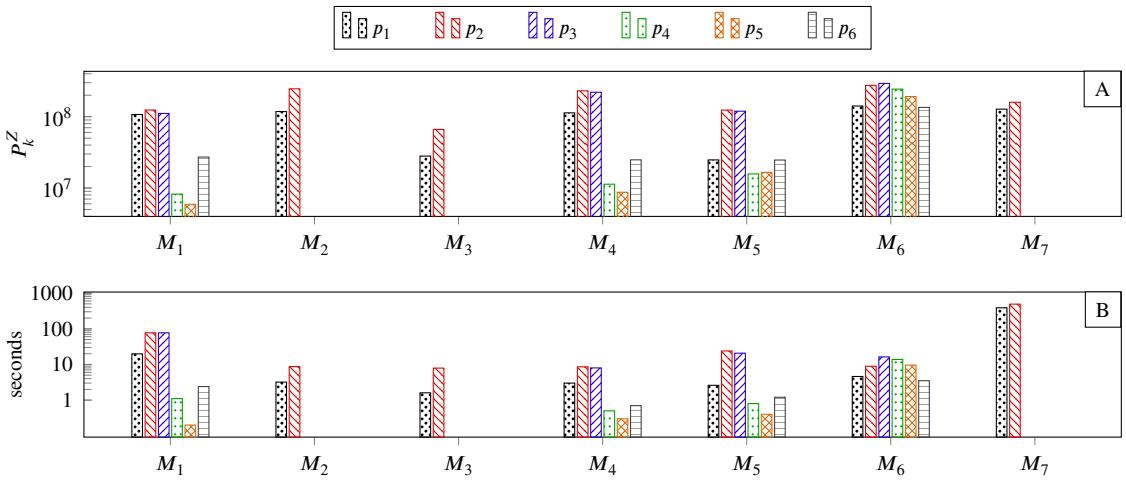


Figure 6.5: *(A) P_k^Z for each query plan P_k. (B) Execution time in ms for each query plan using eight machines ($m = 8$).*

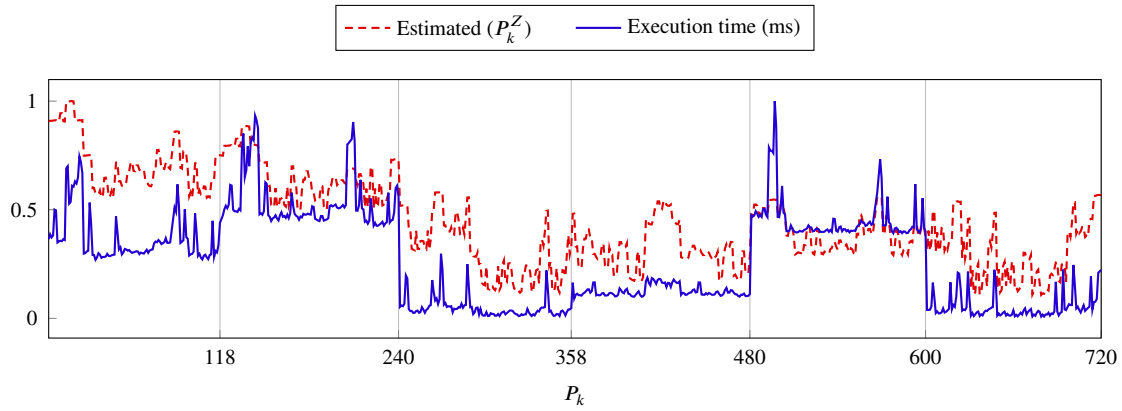


Figure 6.6: *Execution time and cost of each plan for M_8, using eight machines ($m = 8$).*

p_{358} to p_{480} but slightly worse than (3), (5) another set of bad plans from p_{481} to p_{600}, and finally, (6) another set of good plans from p_{601} to p_{720}. Let us concentrate on the two sets of good plans (3) and (6). These two sets are not completely homogeneous, as there are plans that represent peaks in the execution time. Comparing the two lines, we can see that the estimated values are strictly associated with those peaks.

Furthermore, the region with lower values in (3) and (6) contains the plans ranked in the top 10 plans. Table 6.3 shows the details of these plans. The plan ranked first with regard to the execution time was in the set (6), and indeed, coincided with the plan ranked first by using the estimated values. In other words, the method identified the best plan for the query. The second, third, sixth, seventh, and ninth plans are also ranked similarly by the estimate. The others are ranked slightly further away by the estimates than in the top 10 ranking of execution time.

The analysis of M_8 shows that, due to the estimated values produced through the methods and the dataset approximations used, a likely scenario in a larger set of queries will be the selection of a very good plan, with regard to the subsets of good and bad plans that were identified with a reasonable distinction between them.

Table 6.3: *Top 10 plans for M_8 and their respective rank based on the estimated costs.*

Plan	Rank	Est. Rank	Execution time (ms)	P_k^Z
p_{679}	1	1	1,281	63,733,438
p_{681}	2	5	1,285	67,381,386
p_{673}	3	3	1,335	64,304,850
p_{663}	4	17	1,418	73,936,524
p_{661}	5	34	1,492	77,586,539
p_{633}	6	4	1,611	68,292,784
p_{350}	7	9	1,636	71,472,129
p_{331}	8	12	1,637	71,277,014
p_{637}	9	2	1,647	64,614,058
p_{309}	10	23	1,661	78,261,003

6.3.3 Resource Consumption of Schedules

In this section, we compare the estimated and real resource consumption (cpu and network usage) for the two large queries, M_7, and M_8. In this experiment, we considered the three methods for query scheduling proposed in Chapter 4 and the best and worst plans for each of these two queries. The objective was to measure the effectiveness of the query scheduling with regard to resource consumption and to compare the performance of the schedules generated by the three methods in the execution engine.

Table 6.4 presents the result of the experiment. The available CPU time reported reflects the total processing power available in the cluster, i.e., it considers eight machines with four CPUs each (32 cores). For example, the plan $M_7^{p_1}$ scheduled by LR (first line) executed in 5'32" (wall clock time). To execute the plan in this amount of time, the execution engine in fact used 145'35" from the 177'16" of available CPU time (considering 5'32"×8 × 4). The difference, i.e., 31'40", was wasted by some machines, due to the early termination of job processing (unbalanced execution).

We recall from Chapter 4 that the LR method generated schedules with lower resource consumption compared to that generated by the greedy and LP relaxation methods. By analyzing Table 6.4, we can check if the LR really does result in lower resource consumption when performing the query plans in the execution engine. The difference is significant for larger queries, e.g., 50" for $M_7^{p_1}$. The wasted CPU time is also reasonably smaller for LR in all large query plans, e.g., 31'×51'×70' in $M_7^{p_1}$ for LR, GR, and LP, respectively. This result reinforces the superiority of LR. The lower execution time usually compensates for the large optimization time, but more importantly, in an environment with repetitive query execution, this can produce a consistent increase in system throughput. In the next section we investigate the wasted CPU time by examining the resource consumption as well as the plan execution.

Table 6.4: *Resource consumption for M_7 and M_8 in a cluster with 32 cores, eight machines.*

Plan	Method	Network (GB)	Exec. Time	CPU Time Used	CPU Time Available	CPU Time Wasted
$M_7^{p_1}$	LR	34.44	5'32"	145'35"	177'16"	31'40"
	LP	23.89	6'45"	145'41"	216'28"	70'46"
	GR	36.06	6'22"	152'31"	203'50"	51'19"
$M_7^{p_2}$	LR	55.12	7'39"	211'08"	245'16"	34'07"
	LP	45.45	8'05"	212'38"	258'57"	46'19"
	GR	58.16	9'09"	207'55"	292'54"	84'59"
$M_8^{p_{679}}$	LR	0.73	1.3"	29"	41"	11"
	LP	0.71	1.3"	28"	41"	12"
	GR	1.10	1.7"	41"	55"	14"
$M_8^{p_{497}}$	LR	6.28	2'17"	45'45"	73'25"	27'40"
	LP	11.17	2'38"	45'59"	84'44"	38'45"
	GR	12.78	3'00"	46'46"	96'24"	49'37"

6.3.4 Resource Consumption Along Query Execution

This section presents measurements related to the query execution engine, showing the behavior of the CPU and network usage while executing query plans. As each plan presents a particular CPU and network usage pattern, it is not possible to generalise the result for all plans. Thus, we selected two contrasting plans to present here: $M_7^{p_1}$, and $M_6^{p_6}$. We used $f^q = 1$ in this experiment, to force the maximum balance for the plans, at the cost of an increased network usage. We measured the use of CPU, the use of network and the size of the wait and execute queues on each machine in the cluster, at each second for $M_7^{p_1}$, and at each quarter of second for $M_6^{p_6}$ (due to its smaller execution time). The wait queue capacity was fixed at 50 jobs per machine (at most 400 jobs in the Worker Module when $m = 8$). The execute queue capacity was fixed at 100 jobs per Worker module.

Figure 6.7 shows the result for $M_6^{p_6}$. There are four charts: the percentage of CPU usage in the cluster (A), the Network usage in Mb/s (B), the size of the execute queue for each machine (C), and the size of the wait queue for each machine (D). The horizontal axis, common to all charts, indicates the time, from zero to 3.5 seconds. The charts depict the usage of the respective resource at each instant of the plan execution. This query plan has three steps with 5572, 5544, and 5450 jobs each, respectively. The total network usage for the plan was 1.36 GB. During the plan execution, none of the resources reached its limit. Although the network usage does not reach the maximum bandwidth available, the number of fragmented transfers limited the use of CPU to approximately 85%. The plan execution finished at the same time (within a second) in all machines, indicating a balanced execution. This can be seen by inspecting the decrease in the CPU usage curve.

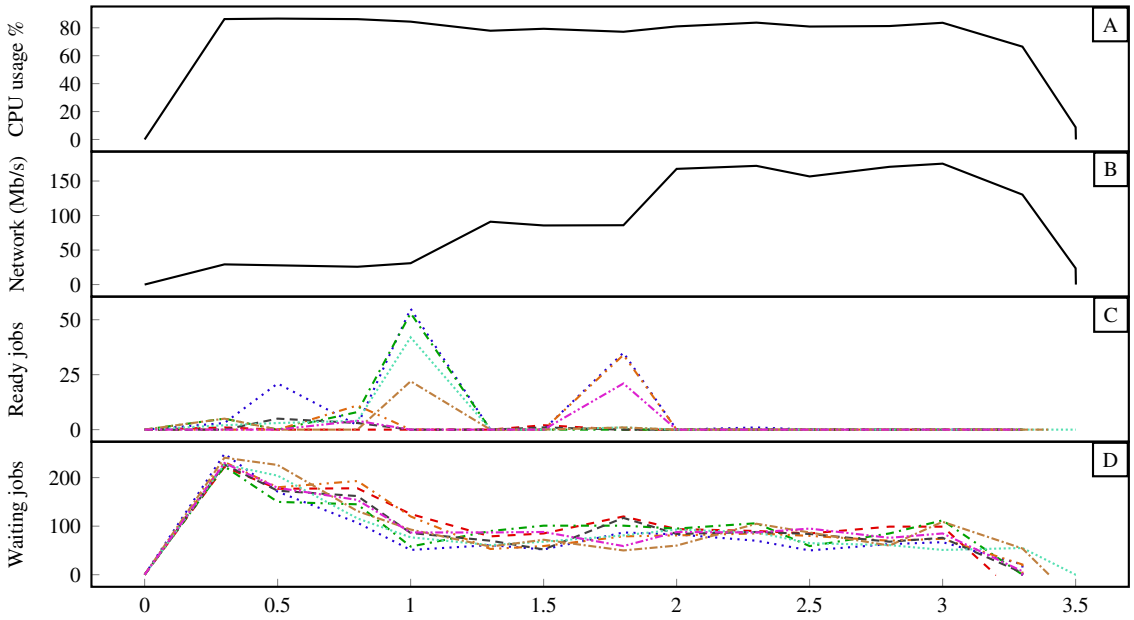


Figure 6.7: *Runtime measurements for $M_6^{p_6}$ scheduled by LR method.*

Similarly, Figure 6.8 presents the same measurements for $M_7^{p_1}$. This plan ran for 332 seconds (5'32") and used 36.06 GB of network bandwidth during its execution. It has two steps with 1420 and 1416 jobs each, respectively. CPU usage stayed at approximately 100% almost all the time, except for two intervals at 30–60s and 85–95s. These two intervals coincide with two empty execution queues (in C, the dotted and the dashed lines at the bottom) for which the two respective machines were waiting for data partitions, as can be seen in the same interval in the wait queue (D). This behavior was caused by the f^q used, which significantly increased the use of the network. The interval for which the execution queue reduces the most in all machines (from 50s to 120s) coincides with the end of the first plan step, when the worker modules start to copy intermediate data partitions from other machines.

Another relevant aspect of the execution of $M_7^{p_1}$ is the fact that the machines finish the execution at distinct times (note the steps in the CPU usage curve, in the last minute of the execution). The schedule provided by the LR method for this plan indicates an almost balanced execution. However, our investigation indicated that the estimated costs failed to identify the complexity of some data partitions. This fact can be evidenced by observing that the machine that ran the longest in the plan is indeed the one that always required the most execution queue time (see the dash dotted line in (C)).

This experiment shows how the precision of the estimates interferes in the execution, with regard to query execution balance, despite the efficiency of the scheduling method. Although it can have a small effect in smaller queries, larger ones such as M_7 can present a reasonable impact. To address this issue, we point out a few directions for future work. As we have shown in Figure 6.5, the selection of the execution plan can identify a

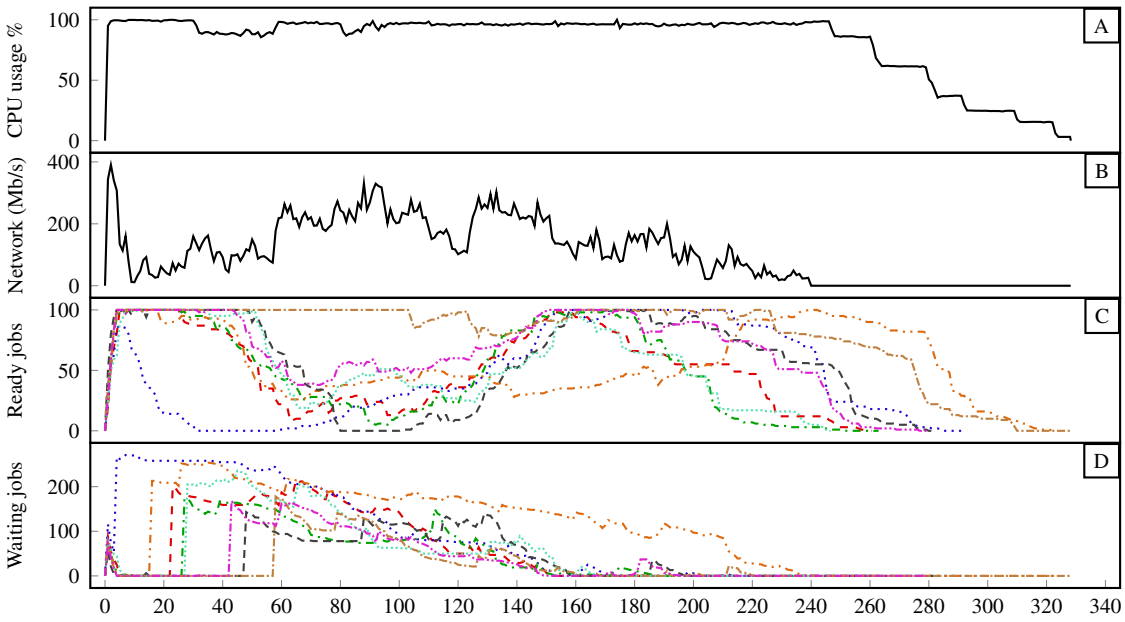


Figure 6.8: *Runtime metrics for $M_7^{p_1}$ scheduled by LR method.*

plan that is orders of magnitude less costly than the worst plan for a query, i.e., the plan selection cannot be neglected in an efficient system for spatial query processing. With regard to this, the first direction we propose is to follow the suggested improvements for the cost model, proposed as future work in Chapter 3. Second, a more elaborate execution engine can be implemented, in which the machine that finishes its processing earlier offers help to others machines that still have jobs to process. This will usually result in a more balanced execution, at the expense of an increase in network usage at the end of plan execution. Finally, in a scenario where multiple queries are processed in batch mode, the remaining load of the first query can be used in the scheduling of the next query, by means of the u_i parameter that features in our proposed models.

6.3.5 Scalability of Query Execution

We also evaluated the scalability of the execution engine when running the multiway queries M_1 to M_8. For this experiment, the best plan for each query, as identified in Section 6.3.2, was executed with $m = 4$, 8, 16, and 32 machines. The result is presented in Figure 6.9. To compute the scalability of each query, we assume that the execution time is equals to one in $m = 4$, and calculate the reduction in the execution time in the other cluster sizes. The solid line labeled "ref." provides a reference for linear scalability.

As the chart in Figure 6.9 shows, except for M_5, all other queries exhibited near linear scalability, reducing nearly half the time to run the query when we double the number of machines. Query M_5 did not exhibit the same scalability due to existing dependencies between its data partitions, that reduces the CPU usage along its execution, when $m \geq 8$. This behavior can occur to some queries as a more complex pattern of dependencies between data partitions can reduce the CPU usage while partitions are

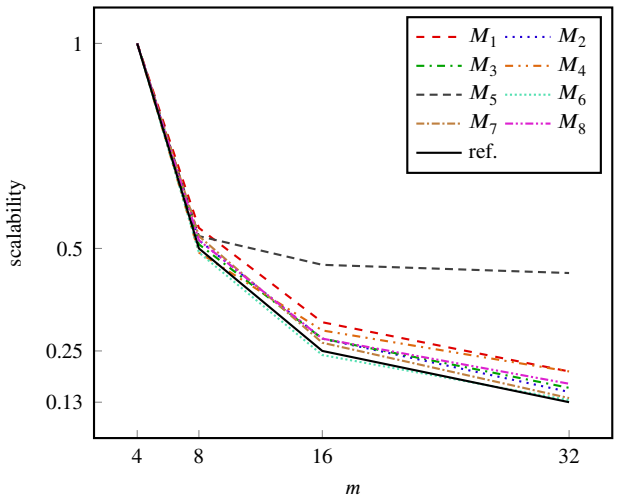


Figure 6.9: *Scalability of the execution engine, running multiway spatial join queries.*

processed or transferred from other machines. Also, queries with a small number of jobs, similar to M_5, are more susceptible to this problem as the number of jobs to fill the execution queue is limited. Although this can be mitigated by a more elaborated data partitioning, it is also dependant on the query itself and in the characteristics of the datasets that are joined. Thus, we regard this issue as future research.

6.4 Final Considerations

In this chapter, we introduced some methods for the query optimizer that is proposed in this thesis. We also presented a simple query execution engine that can execute multiway spatial join queries following the schedules provided by the optimizer. The evaluation showed that the estimated communication cost identified the pattern of network usage when the query plans were run on the execution engine. Furthermore, the plan selection method chose the best plan for almost all queries in our experiments and usually established a good ordering of the plans.

We also investigated resource consumption in the execution of a query plan for the three proposed methods for queries scheduling and how the estimated query effort can interfere with the balanced execution of the plans. We also indicated future directions of research to further improve the precision of the cost model and the query execution engine.

We have assumed throughout this thesis that the machines in the cluster are used exclusively to process a single query at a time (thus, without interference from other queries). Although this is a reasonable assumption due to the size and the resource consumption of spatial queries, it does not allow for the processing of multiple queries concurrently (i.e., a multi-query workload, with the queries sharing the cluster resources [66]). In the multi-query scenario, the execution engine must be able to decide where and when to run queries concurrently before plan scheduling. Also, other possible sources of processing costs are not represented in the proposed models, such as operating system costs, which can interfere with the efficient usage of the cluster resources. These issues are left as future research.

Another question that requires future work is the processing of other types of multiway spatial queries. Although the chain queries are a representative kind of multiway spatial join query and have many practical uses, the other types of queries mentioned in Chapter 2 have different CPU and network usage patterns. For each intermediate result, a chain query checks the predicate between the spatial object in the main dataset (at the root of the plan tree) and the spatial object of the dataset in the next step. In contrast, due to the non-transitivity of spatial operators, clique queries for each intermediate result verify the predicate of all previous spatial objects [55]. The CPU usage pattern of bushy queries,

including cycle queries, lies in between the usage pattern of chain and clique query, as bushy queries can be decomposed into a set of chain and clique sub-queries [2].

In the next chapter, we present a literature survey of work on issues that are different from, but related to the ones discussed in the present thesis. Further, we compare the reported solutions with the respective ones addressed in this thesis.

CHAPTER 7

Related Work

Recently, growing attention has been focused on the processing of spatial data in distributed systems and a number of studies on this topic have been published (see [29] for a survey). In this chapter, we present a literature survey of multiway spatial join processing in distributed systems and compare the reported methods with the respective ones identified in this thesis.

We begin with a discussion of work proposing foundations on multiway spatial join query optimization and execution in Section 7.1, briefly recalling some studies discussed in detail in Chapter 2 to place it in context. Recently reported work based on underlying frameworks for distributed data processing is discussed in Sections 7.2 and 7.3. We also discuss systems designed from scratch in Section 7.4. Section 7.5 presents a comparison of the execution time of multiway spatial join queries performed in our execution engine and in the other works surveyed. A summary of the comparison is presented in Section 7.6. We conclude by presenting our conclusions and final considerations in Section 7.7.

7.1 Foundation Work on Spatial Query Optimization

Mamoulis and Papadias [57] presented an in-depth study of non-distributed spatial join queries processing, describing algorithms, data partitioning, plan cost estimation, and plan selection. To the best of our knowledge, it is the most relevant study in this regard. The same authors also described a cost-model for window queries, simple spatial join, and multiway spatial join in non-distributed systems [56]. Several of the authors' algorithms and methods were mentioned in Chapter 2 and adapted in our work in order to build a solution for distributed systems. Other work also propose non-distributed query optimizers, such as the work of Fournari et al. in [33, 34]. However, they are limited to simple spatial join queries.

A number of researchers (e.g., [62, 69, 70]) propose distributed algorithms for spatial join using two datasets at a time. Despite the fact that these algorithms are not designed for multiway spatial join, they comprise a set of general algorithms for spatial join

processing. Due to this, their design principles constitute a foundation, and are employed to process multiway spatial join queries in successive steps both in the present study and in other systems mentioned in the following.

Apart from those foundations, there is also significant research on distributed database technology [65], with some DDBMS supporting features for spatial data handling. In our work, we considered the distribution of query execution at the intra-operator level [65], i.e., even a single step of the query plan can be distributed in the cluster. This design attempts to address the efficient and scalable processing of multiway spatial join over large datasets. Regarding spatial data, the traditional focus of DDBMS systems is to support multi-query workloads and remains at the inter-operator and inter-query level of parallelism. To the contrary, the body of research dedicated to processing spatial data in distributed data processing frameworks, such as MapReduce [24], Spark [92], Impala [7], focuses on the intra-operator level. Since our scope does not include multi-query workloads, we limited our comparison in this regard. Notwithstanding, as discussed in Chapter 2, we attempted to consider both intra-operator parallelism and query planning and optimization. Query planning and optimization has been considered mostly only in DDBMS systems.

There are exceptions, however, with regard to intra-operator parallelism on spatial DDBMSs. Recently, a distributed version of the DBMS Secondo [40] was presented by Nidzwetzki and Güting [63] that uses Apache Cassandra [50] as the underlying data storage subsystem and a query execution engine that supports query optimization. The discussion of this work is given next.

In the following, we review systems to process multiway spatial join queries. We have divided the discussion into three sections: systems that are built on existing frameworks such as *i*) MapReduce, *ii*) Spark, and *iii*) systems that are designed from-scratch with regard to query planning and execution.

7.2 MapReduce-based Work

In this section, we cover the work proposed in the literature to process multiway spatial join queries using the MapReduce framework [24].

The early studies based on MapReduce focused on how to cope with the design decisions of the framework, such as the need to process the data in two phases (the map and reduce functions) and the need to create homogeneous tasks with regard to the load they cause in the reduce phase. Some examples of works that proposed methods to deal with these limitations are SJMR [93], VegaGiStore [94], and SpatialHadoop [28]. These studies support the processing of only simple spatial joins, and a strategy to process multiway queries is not presented.

With regard to the processing of multiway spatial join queries, another limitation imposed by the design of MapReduce is the need to persist the intermediate results when processing the query through pairwise joins. This causes a high I/O cost due to having to read and write the intermediate data from disks, which in general causes a degradation of the performance of multiway query processing.

Furthermore, the default load balancer[1] of MapReduce also imposes challenges to spatial data processing. The default algorithm lazily assigns tasks to available slots in the cluster. This behavior requires tasks to have evenly spread loads to perform a balanced execution [49]. As discussed earlier, spatial data is by nature non-uniform and thus, the occurrence of straggler tasks is expected, resulting in an unbalanced execution. Although other algorithms that improve the execution of non-uniform tasks in MapReduce frameworks are available (e.g., [60, 86]), all reported studies maintain a focus on providing equally-sized-tasks.

In the following, we describe some relevant works and discuss how each of them address these issues.

Hadoop GIS: This work was first introduced by Aji et al. [2] and later improved by the same authors in [3]. To provide evenly-sized tasks, the authors used a recursive data partitioning method, which splits a set of initial partitions (or tiles) until an arbitrary trade-off is reached. To overcome the high I/O cost for intermediate results, an algorithm based on Synchronous Traversal [67] is used, in which all query steps are processed in a single pass, combining all the datasets at once. Although the authors mention a query optimizer based on heuristic rules, no detail is provided about the rules used. Also, as it is based on predefined rules (not cost-based), a plan selection method is not mentioned.

Hadoop ϵCP: This work was first introduced in [39] and later improved in [38]. The authors proposed a system to process multiway spatial join queries by processing all datasets in two passes: the first identifies which data partitions need to be joined with each other (the Controlled-Replicate method) and the second performs the predicate check. While identifying the data partitions to be joined, the Controlled-Replicate method attempts to minimize the communication cost incurred. This step is necessary as dataset metadata is not considered when identifying the data partitions pairs that need to be processed (like the metadata provided by the histograms we propose in Chapter 3). With regard to the data partitioning, the authors used a grid of disjoint cells to split the spatial extent, using replication of objects that intersect more than one grid cell. Duplication avoidance is achieved by using the Reference Point method [25]. The work addresses only the filter step of join queries, i.e., it processes only the MBR of spatial objects, and does

[1]The term scheduling on MapReduce literature is reserved for the distribution of cluster resources to multi-user loads, similar to multi-query database loads.

not identify the real result set for a query. Furthermore, the authors did not consider the use of either a rule-based nor a cost-based optimizer.

Sphinx: The work of Eldawy et al. [30] extends the distributed SQL processing engine Impala [7] to process multiway spatial join queries. Impala is an SQL query engine for scalar data that runs on top of a MapReduce framework. It provides a query optimizer that can enumerate and determine the cost of plans based on the cardinality and the number of distinct values of data partitions. Furthermore, it determines a distributed execution plan before the query execution, but with the single goal of reducing the communication cost in the cluster. Although it uses MR, only the storage subsystem is used. The map and reduce mechanism was replaced by a customized execution engine. Sphinx introduces spatial data types, spatial indices, and spatial predicate checking algorithms into Impala, extending it in order to process spatial queries. The authors compared the execution time of simple spatial joins and concluded that Sphinx can achieve a speed-up of up to three times, compared with SpatialHadoop [28]. Although multiway spatial join is not mentioned, we believe it can be performed, since Impala stores the intermediate results in memory (or on disk) and executes a cascaded pairwise execution of steps. However, the authors focus on simple spatial joins and the details of how to identify the size and communication costs of intermediate results were not discussed. Similarly, the costs of processing tasks were not mentioned. As we investigated in Chapter 4, there is a trade-off between communication and processing costs and a schedule produced with the single goal of reducing the communication costs can lead to a significant increase in the makespan.

7.3 Spark-based Work

In this section, we present systems to process multiway spatial join queries using the Spark engine [92]. Spark has a programming model similar to MapReduce but extends it via a data-sharing abstraction called “Resilient Distributed Datasets” (RDD). Through RDDs, Spark addresses the limitation of the MapReduce model with regard to the high I/O cost incurred. This is achieved by the use of replicated external storage for the purpose of sharing intermediate data across processing steps. Also, Spark focuses on interactive tasks processing, presenting a smaller latency than MapReduce (100ms), which in turn focuses on batch oriented processing [92]. Further, Spark allows the storage and processing of data totally in-memory and its API enables an application to set the preferred locations for tasks, a feature that might be used to schedule data to be processed on specific machines.

The processing of multiway spatial queries through pairwise joins does not have the same limitations of MapReduce due to the use of in-memory RDDs. However, similar to MapReduce, in order to perform a balanced execution, Spark also requires evenly-sized tasks. Thus, beyond the proper scheduling of tasks and the selection of plans, Spark does

not have ways of handling spatial data, such as a mechanism to partition data, a language to specify spatial queries, nor predicate checking algorithms. Some early work on spatial data processing using Spark only supports simple spatial joins, not presenting a strategy to process multiway queries, and is not discussed here (e.g., [88, 89, 91]).

MSJS: To the best of our knowledge, the only study reported to process multiway spatial join queries using Spark is MSJS [27]. To process a multiway join, MSJS first re-partitions each dataset using a uniform grid, with an arbitrary number of cells. It replicates objects that intersect with more than one grid cell and also attempts duplication avoidance by using the Reference Point method [25]. After the partitioning phase, MSJS joins the partitions with the same IDs in a number of pairwise joins, producing intermediate results in memory. No strategy to determine the costs of a query is discussed. The authors evaluated the effect of the number of partitions in their algorithm and their conclusion is similar to ours in Chapter 3. That is, the number of partitions corresponding to the best performance should be carefully selected so as not to be overly large nor small. However, the number of partitions was identified in their study by experimentation rather than by an algorithm. Also, the scheduling of data partitions is performed by the default load balancer provided by Spark and the API for setting the preferred locations for tasks was not mentioned. Further, with regard to the effect of the different join plans (referred to as the input sequence), the authors concluded by experimentation that datasets should be processed in order of size, smallest first. Although this is a general principle that can result in a smaller intermediate result, we have shown in Chapter 6 that selecting the best plan for a query is not trivial with large queries and the scheduling of tasks should be considered as it impacts query execution time.

We compare our proposed query optimizer and execution engine with the results reported by Du et al. [27], using a few queries also evaluated in their work and a similar cluster size. We present the results in Section 7.5.

7.4 Systems Designed from Scratch

In this section, we review two systems built from scratch, in the sense that they did not use a general underlying framework for distributed data processing.

Distributed Secondo [63] is a distributed, general-purpose DBMS which is highly scalable and fault tolerant, using the framework Cassandra [50] to store data. A uniform grid is used for spatial data partitioning and objects that intersect with more than one cell are duplicated. The Reference Point method is used to prevent duplicate results. An algorithm adapted from SMJR [93] is used to execute the spatial join. With regard to query scheduling, the authors propose a decentralized algorithm to assign tasks to query processing nodes in which each machine creates its own jobs based on local data.

A load balancing algorithm is used at the end of the query execution in which under-utilized machines are used to reduce the load of busy ones by randomly reassigning some tasks. Similar to MapReduce based work, intermediate results also need to be persisted to process multiway queries. Further, the authors left the integration of distributed queries in the query optimizer provided by DBMS Secondo [40] as future work.

DST: Cunha et al. [18] proposed a distributed algorithm to process multiway spatial join queries in distributed systems by joining all datasets at once. It is similar to the *Synchronous Traversal* (ST) [67] algorithm and uses the independent framework for spatial data processing introduced in [20, 23]. Data partitions are created using a recursive, non-disjoint partitioning scheme, based on R-Trees, in which spatial objects do not need to be replicated. Two algorithms for data distribution are considered: the first one is based on a round-robin allocation and the second one, called Grid Proximity Area [20], allocates data partitions to machines based on the spatial boundaries of each partition. Since all datasets are processed at once, no query optimizer was presented.

7.5 Comparison of Execution Times

In this section, we compare the execution times of three multiway spatial join queries, when executed using our proposed methods and in other two related works: one based on MapReduce, Hadoop ϵCR [38], and the other based on Spark, MSJS [27]. Furthermore, we put into perspective results presented by Cunha et al. [18] (DST) and by Nidzwetzki and Güting [63] (Distributed Secondo). Sphinx [30] presented results only for simple spatial joins using two synthetic datasets with rectangles (not real spatial objects) and thus, it was not considered in this comparison.

We compared the execution times reported by Du et al. [27], for the queries PR $\bowtie$ LM $\bowtie$ AW (small, M_9), PR $\bowtie$ AW $\bowtie$ ED (medium, M_{10}), and AW $\bowtie$ LW $\bowtie$ ED (large, M_{11}). Both methods used datasets from the TIGER 2015 spatial database. The datasets and their sizes in the FileGDB format are: Primary Roads (PR) 41MB, Area Land Mark (LM) 105MB, Area Water (AW) 823MB, Linear Water (LW) 2.1GB, and Edges (ED) 14GB.

The execution environment used by Du et al. [27] was composed of four Power Edge R720 Servers. Each server has one Intel Xeon E5-2630 v2 2.60 GHz processor, 32 GB of RAM and runs a SUSE Linux enterprise server 11 SP2 operating system. According to the Intel website, the Xeon E5-2630 has six cores and 12 hyper-threads. The network capacity was not mentioned. The version 2.6.0 of Hadoop and the version 2.0.1 of Spark were used, both running on JDK 1.7.

To executed those three queries (M_9, M_{10}, and M_{11}) using our proposed methods, we configured a similar environment composed of four m4.2xlarge Amazon EC2 in-

stances, each with eight vCPUs of an Intel(R) Xeon(R) CPU, E5-2686 v4 model, running at 2.30GHz, and with 32GB of RAM. According to the Amazon specification, each vCPU is a Hyper-thread of an Intel Xeon core.[2] The machines were allocated to the same data center, interconnected by a virtual network with a capacity of up to 10 Gbps for single-flow and 20 Gbps for multi-flow traffic in each direction (full duplex). The operating system used was the default Ubuntu Server 16.04 LTS offered by the provider through an AMI image.

Thus, in comparing the specifications, we believe that the hardware used in [27] is similar to ours. Some differences that work against our own environment are: the lower CPU clock (2.6 × 2.3 GHz), the smaller number of hyperthreads per server (12 × 8), and the use of a virtualized environment that incurs the Hypervisor overhead.

The chart in Figure 7.1 presents the execution times of each query for each system. The smallest query, M_9, was performed in 6" by our method, while MSJS executed the same query in 82" (13.6x). Similarly, the medium query, M_{10}, was executed in 2'21" by our method and in 21'08" by MSJS (9x). For the largest query, M_{11}, our method took 18', compared with 28' by MSJS (1.6x). The gap between the two systems was lessened when the query required more communication in the cluster. However, the difference in the amount of time remains significant. The largest query terminates 10' earlier for our method compared to MSJS. The performance of the systems based on MapReduce is worse than the systems that perform the execution in memory, without persisting intermediate results.

Although the same queries were not executed in DST [18] and Distributed Secondo [63], we nevertheless present some results given by the authors in order to put them in perspective. The authors of Distributed Secondo [63] compared the performance of their system with those of SpatialSpark [89] and SpatialHadoop [28]. The times reported

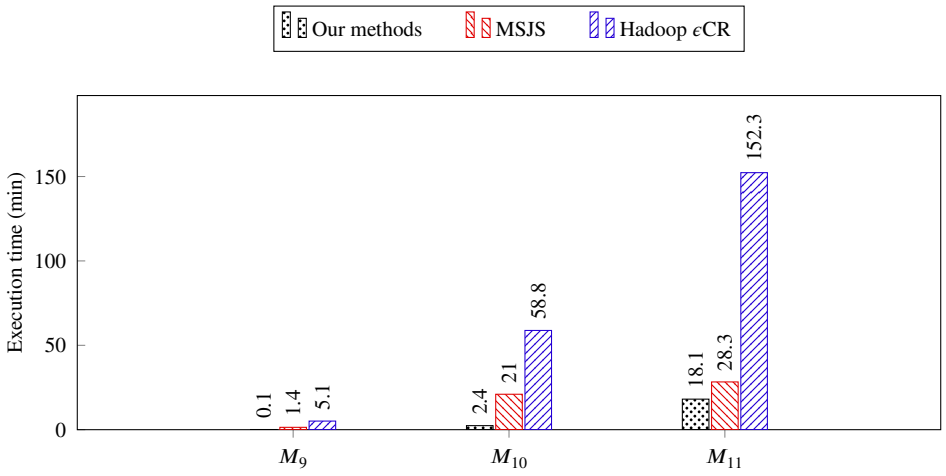


Figure 7.1: *Comparison of the execution times for M_9, M_{10}, and M_{11}.*

[2]https://aws.amazon.com/ec2/instance-types.

for a simple spatial join involving two datasets from Germany, called Roads and Building (size not mentioned), generated from Open Street Maps (http://www.openstreetmap.org), are 9'41" for Spatial Spark, 21'54" for Distributed Secondo, and 23'57" for Spatial Hadoop. Considering this comparison and, due to the fact that Distributed Secondo also stores intermediate results on disk, it seems that its performance is similar to that of MapReduce-based systems.

With regard to DST [18], even plans for small datasets seem to require relatively large runtimes. For instance, a multiway spatial join query (IHRF) involving four datasets of sizes 64MB, 15MB, 1.5MB, and 1MB, ran for 9'19" in a cluster of eight machines. The same query ran in less than one second using our method in the hardware described at the beginning of this section.

7.6 Overall Comparison of Features

Table 7.1 presents a summary of the work discussed in this chapter. We included in the table only systems that were designed to run multiway spatial join queries. The symbol ● indicates that a feature is supported as a primary objective and an evaluation is conducted. The symbol ⊙ indicates that the feature is supported by the underlying framework, sometimes not considering the specifics of spatial data, such as the case for general cost estimation based only on cardinalities and default plan scheduling (load balancing) without beforehand considering the task load in a planning step. The symbol ○

Table 7.1: *Summary of related work and their capabilities. The legend for the symbols are: supported feature ●, feature supported by the underlying framework ⊙, feature partially supported ○, and feature not supported ×.*

System	Data Partitioning	Data Distribution	Cost Estimation	Plan Scheduling	Plan Selection	Query Execution
Hadoop GIS [3]	●	⊙	×	×	×	●
Hadoop ϵCP [38]	●	⊙	×	×	×	○[a]
Sphinx [30]	●	⊙	×	×[b]	⊙[c]	●
MSJS [27]	●	⊙	×	×	×	●
DST [18]	●	●	×	×	×	●
D. Secondo [63]	●	●	×	○[d]	×	●
This thesis	●	○[e]	●	●	●	●

[a] Only considers MBRs. Thus, if the data is polygon based, only the filtering step is performed.

[b] Data partitions are distributed to machines, but a strategy to measure its loads and assign to machines with some criteria is not discussed.

[c] Rule-based selection.

[d] The scheduler does not define a global strategy to execute tasks. It is a decentralized algorithm, in which each machine creates its tasks based on local data.

[e] Round-robin based, with support for other distribution strategies.

indicates that the feature is partially supported, i.e., it is mentioned in the study and a naïve method is considered. However, better alternative methods are known and not implemented. Finally, the symbol × indicates that the feature is not supported.

As can be seen in Table 7.1, our work is the only one that considers all phases of a distributed cost-based query optimizer. Some observations are given as notes on some features. The focus of the other works was on data partitioning and on query execution and these features are present in all systems. The only work that considers plan selection is Sphinx, provided by the underlying framework (Impala). However, it is an optimizer based on rules, some of them considering costs, but only costs given by cardinalities and sizes of data partitions (no specific cost for spatial data is considered).

7.7 Final Considerations

In this chapter, we compared our proposed methods against related work in the literature, showing the originality of our cost-based query optimizer for multiway spatial join queries in distributed systems. The reasons for its better performance, when compared to other similar systems, stem from the focus on query planning, in particular, the query scheduling method and the design choices, which are distinct from the related work.

About the importance of query scheduling, we demonstrated in Chapter 4 the difference in the resource consumption between a naïve greedy algorithm and better methods based on the theory of combinatorial optimization. The previously reported systems compared here considered only a load balancing mechanism, rather than a query scheduling based on metadata. This load balancing is performed by the underlying framework or implemented from scratch, but in both cases, based on the assignment of tasks to idle machines during query execution, i.e., a kind of greedy strategy that often results in inefficient cluster resource usage.

With regard to our design choices, we considered an approach for data partitioning in which a particular grid is used for each dataset and its granularity, (size of data partitions), is based on the metadata gathered from datasets. In this way, we can maintain metadata about each partition, determine the tasks for each query in advance, and optimize a query before its execution. This strategy also does not require the repartitioning of data in each join query performed, in contrast to the related work investigated.

CHAPTER 8

Conclusion

The efficient processing of multiway spatial join queries in distributed systems imposes significant challenges. In this thesis, we addressed some of these challenges by proposing and evaluating models and methods that constitute a cost-based optimizer for multiway spatial join queries. We conclude this thesis by reflecting on our contributions (Section 8.1), the limitations of our approach (Section 8.2), and opportunities for future work (Section 8.3).

8.1 Summary of Contributions

The contribution of our work primarily lies in providing a set of comprehensive models and methods that form a cost-based optimizer for multiway spatial join queries. The optimizer is able to select a good execution plan for distributed processing, taking into account *i*) the partitioning of data based on spatial attributes of datasets, *ii*) the intra-operator level of parallelism, which enables high scalability, and *iii*) the economical use of cluster resources by appropriately scheduling the query before execution. The extent and relevance of our work can be accessed by a detailed description of our investigation and results, given next.

With the aim of designing a cost model, we started by identifying relevant metadata for spatial datasets and the distribution of data partitions in the distributed system. We proposed an improved multidimensional grid histogram to be used both as a data structure to organize such metadata, as well as a distributed data access method. We then proposed new methods for histogram construction based on the proportional overlap between spatial objects and histogram cell boundaries, which increased the precision of estimates of query costs. Furthermore, we presented improvements to overcome the imprecisions caused by the MBR simplification of spatial objects in grid histograms and introduced formulae to estimate join selectivity of spatial joins with complex spatial objects, i.e., when the two datasets have line objects or when one dataset is of type line and the other is of type polygon. Although the cost model is designed with the estimation

of multiway spatial join queries in mind, it can also estimate the selectivity of window and simple spatial join queries.

By using the costs provided by the cost model, we formulated the distributed multiway spatial join plan scheduling problem as a bi-objective linear model (FM). The model is used to schedule query tasks in distributed systems, considering the minimization of both makespan and communication cost as objectives. We discussed the challenges involved in solving this model and introduced three methods for computing schedules using a simplified version of the model (SM), namely, a greedy algorithm (GR) and two algorithms based on combinatorial methods: the well-known Linear Relaxation (LP) and the more sophisticated Lagrangian Relaxation (LR). The reduction in computing resources when scheduling a query through LR is significant. Although it is a more computationally intensive method, we presented scenarios for which it may be applied. One such scenario is a system that focuses on the processing of mid-size or large spatial datasets that often require a query running time sufficiently large to amortize the optimization execution time. Another one is a system that repeatedly runs stored queries, where the plan can be cached and reused. In these scenarios, the proximity of the approximate schedules to the optimum and the improvement of resource consumption (processing and communication costs) that the LR method produces often result in a considerable increase in system throughput. LP and GR are recommended for scenarios where small *ad-hoc* queries are predominant.

A major feature of the query scheduling method is the ability to control the trade-off between processing and communication costs. Our evaluation showed that attempting to minimize both processing and network utilization creates conflicting objectives, in the sense that to achieve a better balance in the query execution (and consequently a reduced makespan), an extra cost is incurred to transfer partitions to machines that are underutilized. We investigated how to control the scheduling behavior concerning the usage of these resources and proposed a method based on the post-optimality theory of integer linear programming, which can identify all the distinct schedules for a given numerical instance of SM. The method requires the computing of optimal schedules for various sub problems within the instance. To address the scenario where optimal schedules are unavailable or difficult to find, we also introduced a method to provide an upper bound on the exact schedule, based on approximate schedules. For every instance tested, the schedule determined by LR, was very close to the optimal schedule. By adapting the execution of a query to the limits of the processing power and network bandwidth imposed, these two LR-based methods can usually be used to provide answers to practical questions that arise when scheduling queries using the SM model.

The applicability of the models for query scheduling and the accompanying method for resource consumption control is not limited to multiway spatial join queries. The proposed models (FM and SM) are suitable for the scheduling of tasks composed of

data partitions previously distributed in a number of interconnected machines that need to be aligned. Indeed, they are general enough to be used in other distributed system as well, such as a load-balancing mechanism within the existing platforms for distributed data processing (MapReduce, Spark, etc.).

The complete methodology for query cost estimation, query scheduling, and query plan selection was used to implement a query execution engine that can execute multiway spatial join queries in distributed systems, following the schedules provided by the query optimizer. Our evaluation showed that the estimated communication cost followed the pattern of the real data transfer measured when running the query plans on the execution engine. The query optimizer was able to choose good execution plans for all the queries tested in our experiments with public spatial datasets (that have a significant range of sizes). Further, it was possible to establish a consistent relative ordering of all plans with respect to resource consumption. Also, sets of "good" and "bad" plans for a given query were identified with a reasonable level of distinction.

8.2 Limitations of our Approach

While we believe that the models and methods in this thesis are useful, they have been designed under some assumptions and suffer from some limitations when considering more general applications. These limitations are now discussed.

One assumption we have made is that the machines in the cluster are used exclusively to process just one query at a time, without interference from other queries. This does not allow for the processing of multiple queries concurrently in a multi-query workload where queries share cluster resources. In a multi-query scenario, the execution engine must be able to decide where and when to run queries concurrently and how to share cluster resources among them.

Regarding multiway query types, our cost model considers only the estimation of chain queries. Clique queries have distinct predicate checking costs due to the non-transitivity of spatial operators and should be considered in a complete query optimizer for multiway queries. Bushy queries can be decomposed into a set of chain and clique sub-queries. The modification required is limited to the cost model and the execution engine. The other proposed methods do not require modifications in this regard. Further, we concentrated our efforts on proposing a cost model for the intersection predicate. Other predicates should be considered to extend the applicability of the proposed model.

Furthermore, we focused on planning the query execution and implemented an execution engine only to serve the purpose of experimenting with our methods. The present engine does not allow for some requirements of distributed systems, such as fault-tolerance, replication, security, etc. Implementing these requirements will not necessarily

degrade the performance of our methods, as similar previously reported execution engines implemented some of these requirements without a significant reduction in performance [92].

In addition, some sources of processing costs are not represented in the proposed models, such as operating system costs (which can interfere with the execution of a query plan and increase its makespan). Although the most relevant costs for distributed systems are considered in our models, there might be scenarios where other parameters and variables should be considered. In these cases, the schedules we provided would have to be adapted or extended. For instance, one such scenario arises when local costs incurred because data partitions are stored on local disks. Also, the initial load of data from disks, or the final storage of query results, may be relevant to consider in query costs.

Another limitation concerns the methods utilized to compute schedules for FM. We reasoned in this thesis that the possibility of the existence of a polynomial time algorithm to solve FM is remote, i.e., unless P=NP. To address this limitation, we have investigated how fast and how well the optimum for FM can be approximated by exploring methods that solve a simplified version of it (SM). Usually, the best schedules for the SM instances that we tested were produced by the LR method. We explored how good each such SM schedule is when it is considered as a schedule for the corresponding FM instance. To this end, we applied exhaustive branch-and-bound (exact) algorithms to the FM instance whenever it could be done in reasonable running time. Sometimes there was a significant difference between the two schedules, implying there is room for improving the scheduling process even further.

In the next section, we suggest future work that could be carried out to address these limitations and discuss other potential improvements to our proposed models and methods.

8.3 Future Work

We see several promising directions for future work on multiway spatial join query optimization and execution in distributed systems. We indicate in the following possible enhancements to the performance and applicability of the models and methods proposed in this thesis.

It is well-known that dealing with the scheduling of tasks at the intra-operator level requires the use of a very accurate and detailed cost model [65]. In this regard, two main sources of imprecision in our proposed cost model should be addressed: (*a*) imprecision caused by naturally co-located events that increase the chance of intersections between two datasets, and (*b*) the divergence between the uniformity assumption in the proposed formulae and the spatial data itself. For (*a*), specifying a metadata field for

indicating the expected probability of an intersection can be of help. However, this requires prior knowledge about the dataset and about the join itself, not related to each dataset in particular. Another strategy to investigate is the use of query hints to improve cost estimation, a strategy already used in some traditional DBMSs for scalar data [11]. For (*b*), other metadata fields should be investigated in order to capture more information about the location skew of spatial objects inside each histogram cell.

Other sources of imprecision exist in the cost model, regarding the construction of multidimensional grid histograms, such as the handling of the boundary effect by the Proportional Overlap method. In this case, improved histogram construction techniques can be employed. Two of which we are aware of are the MinSkew histogram [1] and the improved version of Euler Histograms proposed in [81]. While investigating this improvement, one should adapt these histograms to be used in data partitioning and as a data access method for a distributed system.

Improvements to the cost model might help to reduce the unbalance that is sometimes experienced during the execution of large queries, despite the efficiency of our proposed methods. We believe this should be the preferred way to aid the unbalance as it is compatible with the reduction of the query resource consumption. However, as imprecision in a cost model is inevitable to some degree, to further reduce unbalanced execution, a runtime load-balancing mechanism could be considered at the end of query execution. That is, tasks from over-used machines can be reassigned to underused machines. This strategy will usually result in a more balanced execution, at the expense of an increase in network usage at the end of the plan execution. Regarding system throughput, it would be interesting to study how to identify the cost of processing the remaining load of a query and consider it in the scheduling of the next load by means of the u_i parameter that features in our proposed models.

Investigating techniques to amortize the footprint of the LR method may be worthwhile. A simple option is to cache and reuse the execution plan after the query optimization, a common strategy used for repetitive or stored queries in DBMSs [37]. More interestingly however, is the parallelization of LR by splitting the execution of the m Knapsack instances in each iteration.

Finally, and perhaps most importantly, the development of approximate methods to solve FM should be considered. Although the schedule computed for SM can be used to execute a query, it presents a gap in the communication cost, when compared with optimal schedules for FM, indicating that there is room for further improvement of the query schedule. However, recalling the prohibitive time to compute the schedule for only the root node, even simple Linear Relaxation cannot be directly applied in a reasonable time. Thus, more research is needed to discover an exploitable structure of FM, so that Lagrangian Relaxation or other techniques can be applied to solve FM directly.

Bibliography

[1] Acharya, S.; Poosala, V.; Ramaswamy, S. **Selectivity Estimation in Spatial Databases**. *SIGMOD Record*, 28(2):13–24, 1999.

[2] Aji, A.; Wang, F.; Saltz, J. H. **Towards Building a High Performance Spatial Query System for Large Scale Medical Imaging Data**. In: *Proceedings of the ACM International Symposium on Advances in Geographic Information Systems*, p. 309–318, Redondo Beach, CA, USA, 2012.

[3] Aji, A.; Wang, F.; Vo, H.; Lee, R.; Liu, Q.; Zhang, X.; Saltz, J. **Hadoop GIS: A High Performance Spatial Data Warehousing System over Mapreduce**. *Proceedings of the VLDB Endowment*, 6(11):1009–1020, 2013.

[4] Angel, E.; Bampis, E.; Kononov, A. **Algorithms - ESA 2001**, volume 2161 in **Lecture Notes in Computer Science**, chapter A FPTAS for Approximating the Unrelated Parallel Machines Scheduling Problem with Costs, p. 194–205. Springer, 2001.

[5] Bazaraa, M. S.; Jarvis, J. J.; Sherali, H. D. **Linear Programming and Network Flows**. Wiley, 4th edition, 2009.

[6] Bertsimas, D.; Tsitsiklis, J. **Introduction to linear programming**, volume 1. Athena Scientific, p. 267, 1997.

[7] Bittorf, M.; Bobrovytsky, T.; Erickson, C. C. A. C. J.; Hecht, M. G. D.; Kuff, M. J. I. J. L.; Leblang, D. K. A.; Robinson, N. L. I. P. H.; Rus, D. R. S.; Wanderman, J. R. D. T. S.; Yoder, M. M. **Impala: A modern, open-source SQL engine for Hadoop**. In: *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, p. 1–10, Asilomar, CA, USA, 2015.

[8] Blair, C. **Advances in Sensitivity Analysis and Parametric Programming**, volume 6 in **International Series in Operations Research & Management Science**, chapter Integer and Mixed-Integer Programming, p. 9.1–9.25. Springer US, 1st edition, 1997.

[9] Brinkhoff, T.; Kriegel, H. P.; Seeger, B. **Efficient Processing of Spatial Joins Using R-trees**. *SIGMOD Record*, 22(2):237–246, 1993.

[10] Brinkhoff, T.; Kriegel, H.-P.; Seeger, B. **Parallel Processing of Spatial Joins Using R-trees**. In: *Proceedings of the IEEE International Conference on Data Engineering*, p. 258–265, New Orleans, LA, USA, 1996.

[11] Bruno, N.; Chaudhuri, S.; Ramamurthy, R. **Power Hints for Query Optimization**. In: *Proceedings of the IEEE International Conference on Data Engineering*, p. 469–480, Shanghai, China, 2009.

[12] Campbell, J. E.; Shin, M. **Geographic Information Systems Basics**. CRC Press, 2001.

[13] Carey, M. J.; Lu, H. **Load Balancing in a Locally Distributed DB System**. *SIGMOD Record*, 15(2), 1986.

[14] Chung, W.; Park, S.-Y.; Bae, H.-Y. **Embedded Software and Systems**, volume 3605 in **Lecture Notes in Computer Science**, chapter Efficient Parallel Spatial Join Processing Method in a Shared-Nothing Database Cluster System, p. 81–87. Springer, 2005.

[15] Conforti, M.; Cornuéjols, G.; Zambelli, G. **Integer Programming**, volume 271 in **Graduate Texts in Mathematics**. Springer, 2014.

[16] Cormode, G.; Garofalakis, M.; Haas, P. J.; Jermaine, C. **Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches**. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[17] Corral, A.; Manolopoulos, Y.; Theodoridis, Y.; Vassilakopoulos, M. **Multi-Way Distance Join Queries in Spatial Databases**. *GeoInformatica*, 8(4):373–402, 2004.

[18] Cunha, A. R.; de Oliveira, S. S. T.; Aleixo, E. L.; de C. Cardoso, M.; de Oliveira, T. B.; do Sacramento Rodrigues, V. J. **Processamento Distribuído da Junção Espacial de Múltiplas Bases de Dados - Multi-way Spatial Join**. In: *Anais do XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, p. 165–178, Vitória, ES, Brazil, 2015.

[19] De Berg, M.; Cheong, O.; Van Kreveld, M.; Overmars, M. **Computational Geometry: Algorithms and Applications**. Springer, 3rd edition, 2008.

[20] de Oliveira, S. S. T.; do Sacramento Rodrigues, V. J.; Cunha, A. R.; Aleixo, E. L.; de Oliveira, T. B.; de C. Cardoso, M.; Junior, R. R. **Processamento Distribuído de Operações de Junção Espacial com Bases de Dados Dinâmicas para Análise de Informações Geográficas**. In: *Anais do XXXI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, p. 1009–1022, Brasília, Brazil, 2013.

[21] de Oliveira, T. B.; Costa, F. M.; Rodrigues, V. J. S. **Definição de Planos de Execução Distribuídos para Consultas de Junção Espacial usando Histogramas Multidimensionais**. In: *Proceedings of the Brazilian Symposium on Databases*, p. 89–100, Petrópolis, RJ, Brazil, 2015.

[22] de Oliveira, T. B.; Costa, F. M.; Rodrigues, V. J. S. **Distributed Execution Plans for Multiway Spatial Join Queries using Multidimensional Histograms**. *Journal of Information and Data Management*, 7(3):199–214, 2017.

[23] de Oliveira, T. B.; do Sacramento Rodrigues, V. J.; de Oliveira, S. S. T.; de Albuquerque Lima, P. I.; de C. Cardoso, M. **DSI-RTree - Um Índice R-Tree Escalável Distribuído**. In: *Anais do XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, p. 719–732, Campo Grande, MS, Brazil, 2011.

[24] Dean, J.; Ghemawat, S. **MapReduce: Simplified Data Processing on Large Clusters**. *Communications of the ACM*, 51(1):107–113, 2008.

[25] Dittrich, J.-P.; Seeger, B. **Data Redundancy and Duplicate Detection in Spatial Join Processing**. In: *Proceedings of the IEEE International Conference on Data Engineering*, p. 535–546, San Diego, CA, USA, 2000.

[26] Doulkeridis, C.; Nørvåg, K. **A Survey of Large-scale Analytical Query Processing in MapReduce**. *The VLDB Journal*, 23(3):355–380, 2014.

[27] Du, Z.; Zhao, X.; Ye, X.; Zhou, J.; Zhang, F.; Liu, R. **An Effective High-Performance Multiway Spatial Join Algorithm with Spark**. *ISPRS International Journal of Geo-Information*, 6(4), 2017.

[28] Eldawy, A.; Mokbel, M. F. **SpatialHadoop: A MapReduce framework for spatial data**. In: *Proceedings of the IEEE International Conference on Data Engineering*, p. 1352–1363, Seoul, South Korea, 2015.

[29] Eldawy, A.; Mokbel, M. F. **The Era of Big Spatial Data: A Survey**. *Foundations and Trends in Databases*, 6(3-4):163–273, 2016.

[30] Eldawy, A.; Sabek, I.; Elganainy, M.; Bakeer, A.; Abdelmotaleb, A.; Mokbel, M. F. **Advances in Spatial and Temporal Databases**, volume 10411 in **Lecture Notes in Computer Science**, chapter Sphinx: Empowering Impala for Efficient Execution of SQL Queries on Big Spatial Data, p. 65–83. Springer, 2017.

[31] Evrendilek, C.; Dogac, A.; Nural, S.; Ozcan, F. **Multidatabase Query Optimization**. *Distributed and Parallel Databases*, 5(1):77–114, 1997.

[32] Fisher, M. L. **The Lagrangian Relaxation Method for Solving Integer Programming Problems**. *Management Science*, 50(12 Supplement):1861–1871, 2004.

[33] Fornari, M.; Comba, J. L. D.; Iochpe, C. **Advances in Geoinformatics**, chapter A Rule-Based Optimizer for Spatial Join Algorithms, p. 73–90. Springer, 2007.

[34] Fornari, M. R.; Comba, J. L. D.; Iochpe, C. **Query Optimizer for Spatial Join Operations**. In: *Proceedings of the ACM International Symposium on Advances in Geographic Information Systems*, p. 219–226, Arlington, Va, USA, 2006.

[35] Garey, M.; Johnson, D. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. Freeman, 1979.

[36] Geoffrion, A.; Nauss, R. **Exceptional Paper – Parametric and Postoptimality Analysis in Integer Linear Programming**. *Management Science*, 23(5):453–466, 1977.

[37] Graefe, G. **Query Evaluation Techniques for Large Databases**. *ACM Computing Surveys*, 25(2):73–169, 1993.

[38] Gupta, H.; Chawda, B. **Web Information Systems Engineering**, volume 8787 in **Lecture Notes in Computer Science**, chapter ϵ-Controlled-Replicate: An Improved Controlled-Replicate Algorithm for Multi-way Spatial Join Processing on Map-Reduce, p. 278–293. Springer, 2014.

[39] Gupta, H.; Chawda, B.; Negi, S.; Faruquie, T. A.; Subramaniam, L. V.; Mohania, M. **Processing Multi-way Spatial Joins on Map-reduce**. In: *Proceedings of the International Conference on Extending Database Technology*, p. 113–124, Genoa, Italy, 2013.

[40] Güting, R. H.; Behr, T.; Düntgen, C. **SECONDO: A Platform for Moving Objects Database Research and for Publishing and Integrating Research Implementations**. *icdebulletin*, 33(2):56–63, 2010.

[41] Hall, J. A. J. **Towards a Practical Parallelisation of the Simplex Method**. *Computational Management Science*, 7(2):139–170, 2010.

[42] Held, M.; Karp, R. M. **The traveling-salesman problem and minimum spanning trees: Part II**. *Mathematical programming*, 1(1):6–25, 1971.

[43] Hormann, K.; Agathos, A. **The Point in Polygon Problem for Arbitrary Polygons**. *Computational Geometry*, 20(3):131–144, 2001.

[44] Ibaraki, T.; Kameda, T. **On the Optimal Nesting Order for Computing N-relational Joins**. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.

[45] Ioannidis, Y. E.; Kang, Y. **Randomized Algorithms for Optimizing Large Join Queries**. *SIGMOD Record*, 19(2):312–321, 1990.

[46] Jacox, E. H.; Samet, H. **Spatial Join Techniques**. *ACM Transactions on Database Systems*, 32(1):1–44, 2007.

[47] Knuth, D. E. **The Art of Computer Programming**, volume 2: Seminumerical Algorithms. Addison-Wesley, p. 232, 3rd edition, 1998.

[48] Kossmann, D. **The State of the Art in Distributed Query Processing**. *ACM Computing Surveys*, 32(4):422–469, 2000.

[49] Kwon, Y.; Balazinska, M.; Howe, B.; Rolia, J. **SkewTune: Mitigating Skew in Mapreduce Applications**. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, p. 25–36, Scottsdale, Arizona, USA, 2012.

[50] Lakshman, A.; Malik, P. **Cassandra: a decentralized structured storage system**. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[51] Lawler, E. L.; Lenstra, J. K.; Kan, A. H. R.; Shmoys, D. B. **Sequencing and scheduling: Algorithms and complexity**. *Handbooks in operations research and management science*, 4:445–522, 1993.

[52] Lenstra, J. K.; Shmoys, D. B.; Tardos, É. **Approximation Algorithms for Scheduling Unrelated Parallel Machines**. *Mathematical Programming*, 46(1-3):259–271, 1990.

[53] Lo, M.-L.; Ravishankar, C. V. **Spatial Hash-Joins**. *SIGMOD Record*, 25(2):247–258, 1996.

[54] Longley, P. A.; Goodchild, M. F.; Maguire, D. J.; Rhind, D. W. **Geographic Information Systems and Science**. Wiley, 2nd edition, 2005.

[55] Mamoulis, N.; Papadias, D. **Integration of Spatial Join Algorithms for Processing Multiple Inputs**. *SIGMOD Record*, 28(2):1–12, 1999.

[56] Mamoulis, N.; Papadias, D. **Advances in Spatial and Temporal Databases**, volume 2121 in **Lecture Notes in Computer Science**, chapter Selectivity Estimation of Complex Spatial Queries, p. 155–174. Springer, 2001.

[57] Mamoulis, N.; Papadias, D. **Multiway Spatial Joins**. *ACM Transactions on Database Systems*, 26(4):424–475, 2001.

[58] Mishra, P.; Eich, M. H. **Join Processing in Relational Databases**. *ACM Computing Surveys*, 24(1):63–113, 1992.

[59] Morrison, J. L. **Elements of Spatial Data Quality**, chapter Spatial Data Quality, p. 1–12. Elsevier, New York, 1995.

[60] Moseley, B.; Dasgupta, A.; Kumar, R.; Sarlós, T. **On scheduling in map-reduce and flow-shops**. In: *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, p. 289–298, San Jose, CA, USA, 2011.

[61] Mutenda, L.; Kitsuregawa, M. **Parallel R-tree Spatial Join for a Shared-Nothing Architecture**. In: *Proceedings of the International Symposium on Database Applications in Non-Traditional Environments*, p. 423–430, Kyoto, Japan, 1999.

[62] Naughton, J.; Ellmann, C. **A non-blocking parallel spatial join algorithm**. In: *Proceedings of the IEEE International Conference on Data Engineering*, p. 697–705, San Jose, CA, USA, 2002.

[63] Nidzwetzki, J. K.; Güting, R. H. **Distributed SECONDO: An Extensible and Scalable Database Management System**. *Distributed and Parallel Databases*, p. 1–52, 2017.

[64] Noltemeier, H. **Sensitivitätsanalyse bei diskreten linearen Optimierungsproblemen**, volume 30. Springer-Verlag, 2013.

[65] Özsu, M. T.; Valduriez, P. **Principles of Distributed Database Systems**. Springer, p. 245–293, 3rd edition, 2011.

[66] Pang, H.; Carey, M. J.; Livny, M. **Multiclass Query Scheduling in Real-time Database Systems**. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):533–551, 1995.

[67] Papadias, D.; Mamoulis, N.; Theodoridis, Y. **Constraint-Based Processing of Multiway Spatial Joins**. *Algorithmica*, 30(2):188–215, 2001.

[68] Papadias, D.; Mamoulis, N.; Theodoridis, Y. **Processing and Optimization of Multiway Spatial Joins Using R-trees**. In: *Proceedings of the ACM Symposium on Principles of Database Systems*, p. 44–55, Philadephia, PA, USA, 1999.

[69] Patel, J. M.; DeWitt, D. J. **Partition Based Spatial-Merge Join**. *SIGMOD Record*, 25(2):259–270, 1996.

[70] Patel, J. M.; DeWitt, D. J. **Clone Join and Shadow Join: Two Parallel Spatial Join Algorithms**. In: *Proceedings of the ACM International Symposium on Advances in Geographic Information Systems*, p. 56–61, McLean, VA, USA, 2000.

[71] Pavlo, A.; Paulson, E.; Rasin, A.; Abadi, D. J.; DeWitt, D. J.; Madden, S.; Stonebraker, M. **A Comparison of Approaches to Large-scale Data Analysis**. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, p. 165–178, Providence, Rhode Island, USA, 2009.

[72] Pisinger, D. **A Minimal Algorithm for the 0-1 Knapsack Problem**. *Operations Research*, 45(5):758–767, 1997.

[73] Ray, S.; Simion, B.; Brown, A. D.; Johnson, R. **Skew-resistant Parallel In-memory Spatial Join**. In: *Proceedings of the International Conference on Scientific and Statistical Databases Management*, p. 1–12, Aalborg, Denmark, 2014.

[74] Rigaux, P.; Scholl, M.; Voisard, A. **Spatial Databases: With Application to GIS**. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2001.

[75] Roh, Y. J.; Kim, J. H.; Chung, Y. D.; Son, J. H.; Kim, M. H. **Hierarchically Organized Skew-tolerant Histograms for Geographic Data Objects**. In:

Proceedings of the ACM SIGMOD International Conference on Management of Data, p. 627–638, Indianapolis, IN, USA, 2010.

[76] Shchepin, E. V.; Vakhania, N. **An Optimal Rounding Gives a Better Approximation for Scheduling Unrelated Machines**. *Operations Research Letters*, 33(2):127–133, 2005.

[77] Shi, W.; Fisher, P.; Goodchild, M. F. **Spatial Data Quality**. CRC Press, 2003.

[78] Shmoys, D. B.; Tardos, É. **An Approximation Algorithm for the Generalized Assignment Problem**. *Mathematical Programming*, 62(1-3):461–474, 1993.

[79] Sivasubramaniam, A. **Selectivity Estimation for Spatial Joins**. In: *Proceedings of the IEEE International Conference on Data Engineering*, p. 368–375, Berlin, Heidelberg, Germany, 2001.

[80] Stonebraker, M.; Abadi, D.; DeWitt, D. J.; Madden, S.; Paulson, E.; Pavlo, A.; Rasin, A. **MapReduce and parallel DBMSs: friends or foes?** *Communications of the ACM*, 53(1):64–71, 2010.

[81] Sun, C.; Agrawal, D.; Abbadi, A. E. **Advances in Database Technology**, chapter Selectivity Estimation for Spatial Joins with Geometric Selections, p. 609–626. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[82] Theodoridis, Y.; Sellis, T. **A Model for the Prediction of R-tree Performance**. In: *Proceedings of the ACM Symposium on Principles of Database Systems*, p. 161–171, Montreal, Quebec, Canada, 1996.

[83] Theodoridis, Y.; Stefanakis, E.; Sellis, T. **Cost Models for Join Queries in Spatial Databases**. In: *Proceedings of the IEEE International Conference on Data Engineering*, p. 476–483, Orlando, Florida, USA, 1998.

[84] Trott, M. **The Area of a Random Triangle**. *Mathematica Journal*, 7(2):189–198, 1998.

[85] Vazirani, V. V. **Approximation Algorithms**. Springer Science & Business Media, 2001.

[86] Verma, A.; Cherkasova, L.; Campbell, R. H. **Two Sides of a Coin: Optimizing the Schedule of Mapreduce Jobs to Minimize their Makespan and Improve Cluster Performance**. In: *Proceedings of IEEE International*

Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), p. 11–18, Washington, DC, USA, 2012.

[87] Williamson, D. P.; Shmoys, D. B. **The Design of Approximation Algorithms**. Cambridge University Press, 2011.

[88] Xie, D.; Li, F.; Yao, B.; Li, G.; Zhou, L.; Guo, M. **Simba: Efficient In-memory Spatial Analytics**. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, p. 1071–1085, San Francisco, CA, USA, 2016.

[89] You, S.; Zhang, J.; Gruenwald, L. **Spatial Join Query Processing in Cloud: Analyzing Design Choices and Performance Comparisons**. In: *Proceedings of the 44th International Conference on Parallel Processing Workshops*, p. 90–97, Beijing, China, 2015.

[90] Yu, C. T.; Chang, C. **Distributed Query Processing**. *ACM Computing Surveys*, 16(4):399–433, 1984.

[91] Yu, J.; Wu, J.; Sarwat, M. **GeoSpark: A Cluster Computing Framework for Processing Large-scale Spatial Data**. In: *Proceedings of the SIGSPATIAL International Conference on Advances in Geographic Information Systems*, p. 70:1–70:4, Bellevue, WA, USA, 2015.

[92] Zaharia, M.; Xin, R. S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M. J.; others. **Apache Spark: A Unified Engine for Big Data Processing**. *Communications of the ACM*, 59(11):56–65, 2016.

[93] Zhang, S.; Han, J.; Liu, Z.; Wang, K.; Xu, Z. **SJMR: Parallelizing Spatial Join with Mapreduce on Clusters**. In: *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, p. 1–8, New Orleans, LA, USA, 2009.

[94] Zhong, Y.; Han, J.; Zhang, T.; Li, Z.; Fang, J.; Chen, G. **Towards Parallel Spatial Query Processing for Big Spatial Data**. In: *Proceedings of the International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, p. 2085–2094, Shanghai, China, 2012.

APPENDIX A

Detailed Results for Schedule Methods

This appendix presents additional material referred to in Chapter 4. Section A.1 present the values of f used when performing the experiments with the J and M queries. Section A.2 presents the complete set of plots for the characterization of the queries, regarding the communication cost and the makespan of the queries. Finally, Section A.3 presents two tables reporting the total execution time required by each scheduling method proposed in this thesis.

A.1 Challenging Values of f

Tables A.1 and A.2 present the challenging values of f, which require schedules that balance communication cost and makespan, for each query studied and number of machines m used in the experiments presented in Chapter 4.

Table A.1: *Challenging values of f used for join queries J.*

Query	m=4	m=8	m=16	m=32	m=64	**Query**	m=4	m=8	m=16	m=32	m=64
J_1	1.14	2.45	5.16	10.75	21.79	J_{11}	0.15	0.35	0.75	1.58	3.20
J_2	0.84	1.87	3.99	8.14	17.26	J_{12}	0.39	0.89	1.87	3.91	7.65
J_3	0.48	1.55	3.76	13.27	38.01	J_{13}	0.76	1.67	3.48	7.11	14.46
J_4	0.63	3.53	7.54	21.33	51.39	J_{14}	1.29	2.85	6.04	12.80	26.57
J_5	0.59	1.40	3.00	6.17	12.66	J_{15}	0.89	1.94	4.04	8.57	16.40
J_6	0.53	1.19	2.80	7.22	16.95	J_{16}	1.15	2.49	5.19	10.38	21.67
J_7	0.34	0.87	1.83	7.65	30.12	J_{17}	1.54	3.36	7.12	14.96	30.00
J_8	0.74	1.65	3.72	8.83	20.67	J_{18}	1.30	2.85	5.96	11.88	24.80
J_9	0.52	1.29	2.89	9.52	32.37	J_{19}	1.59	3.49	7.48	15.64	32.28
J_{10}	1.02	2.29	5.39	11.03	12.99	J_{20}	0.99	2.21	4.74	9.89	20.86

Table A.2: *Challenging values of f for multiway queries M*

Query	m=4	m=8	m=16	m=32	m=64	**Query**	m=4	m=8	m=16	m=32	m=64
$M_{1.1}$	1.18	2.82	5.41	8.73	9.17	$M_{4.2}$	1.27	2.68	6.42	12.56	11.61
$M_{1.2}$	1.94	4.28	8.38	6.47	5.65	$M_{4.3}$	6.74	17.14	32.56	64.25	117.39
$M_{1.3}$	6.44	13.87	30.32	55.19	53.79	$M_{5.1}$	3.93	8.90	18.31	31.45	33.28
$M_{2.1}$	0.74	1.66	3.44	7.00	14.93	$M_{5.2}$	3.48	6.30	6.92	7.82	7.90
$M_{2.2}$	5.82	12.70	26.62	55.77	114.00	$M_{5.3}$	5.83	13.13	27.83	54.18	50.15
$M_{3.1}$	1.43	3.14	6.54	13.41	26.39	$M_{6.1}$	2.61	5.75	12.17	24.91	51.54
$M_{3.2}$	3.70	8.33	17.41	36.70	72.90	$M_{6.2}$	3.25	7.24	15.31	32.00	63.98
$M_{4.1}$	1.38	3.33	7.09	13.98	27.23	$M_{6.3}$	4.31	9.91	20.36	42.25	86.29

A.2 Characteristics of Schedules Computed

Figures A.1 and A.2 present a series of plots for each query tested, depicting the characteristics of schedules generated by GR, LP, and LR, with $m = 64$ and $f = 0$, $f = 100.000$, and a challenging value of f that makes the contribution of makespan and communication cost about the same on Z_{SM}. The points in the top-left, bottom-right, and bottom-left, represent the schedule with the minimum communication cost, minimum makespan, and balanced makespan and communication cost using the values of f presented in Tables A.1 and A.2, respectively. The evaluation of these plots was presented in Section 4.11.1.

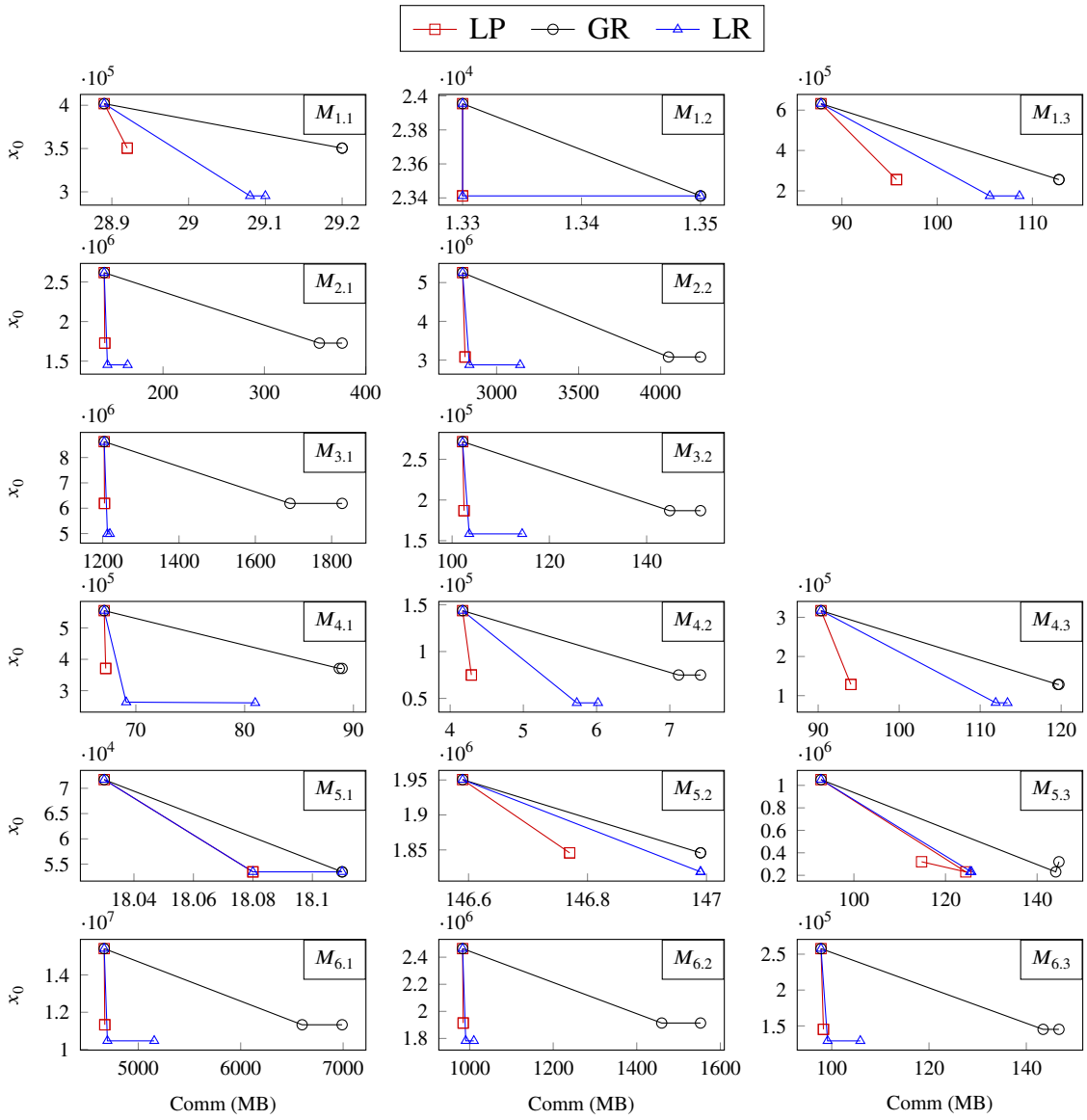


Figure A.1: *Schedule costs for tested instances using distinct values of f for each step of queries from M_1 to M_6.*

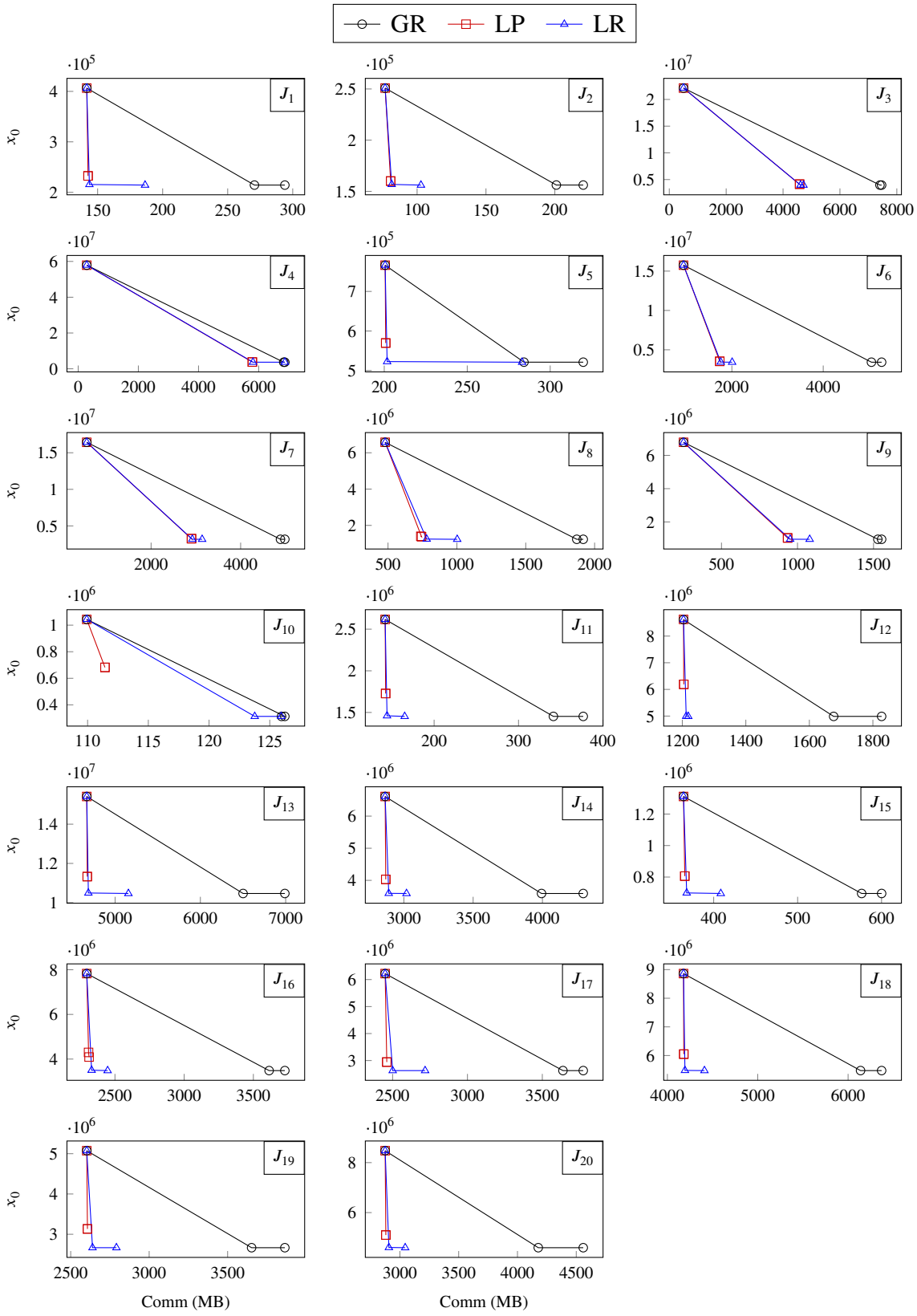


Figure A.2: *Schedule costs for tested instances using distinct values of f for queries J_1 to J_{20}.*

A.3 Execution Times of GR, LP and LR for all Queries

Tables A.3 and A.4 present the execution time required to produce a schedule by GR, LP, and LR for each query J and M. A more comprehensive evaluation of these values was presented in Section 4.11.3.

Table A.3: *Execution times to produce a schedule using GR, LP and LR for m = 4, 8, and 16. Values are in milliseconds.*

Query	m = 4			m = 8			m = 16		
	GR	LP	LR	GR	LP	LR	GR	LP	LR
J_1	3	87	1968	5	172	3707	7	426	5476
J_2	3	84	1676	4	172	3377	7	845	6127
J_3	3	67	341	4	138	787	6	940	1031
J_4	3	63	326	4	1465	3114	6	1554	6935
J_5	3	77	2127	4	140	1980	6	363	4924
J_6	3	73	1662	4	131	2934	6	356	4712
J_7	3	64	1436	4	157	3519	6	481	5775
J_8	1	18	925	1	35	1745	1	88	3048
J_9	1	17	741	1	38	1410	2	96	3244
J_{10}	0	1	902	0	2	1706	0	4	3010
J_{11}	2	53	1453	3	106	2543	5	226	5192
J_{12}	4	133	2845	6	206	4227	10	457	7645
J_{13}	4	110	2628	6	214	4029	9	521	7024
J_{14}	3	70	1728	4	151	3789	6	284	6141
J_{15}	2	47	1382	2	92	2216	4	195	4650
J_{16}	2	46	1358	2	88	1223	4	186	2452
J_{17}	2	41	693	2	84	1292	3	173	2414
J_{18}	4	100	1614	5	186	2681	8	361	6424
J_{19}	2	59	1005	3	113	1571	5	258	5961
J_{20}	2	47	1399	3	88	1613	4	204	4894
$M_{1.1}$	0	2	96	0	3	237	0	4	504
$M_{1.2}$	0	1	61	0	2	99	0	2	194
$M_{1.3}$	0	1	71	0	2	98	0	3	168
$M_{2.1}$	2	56	1012	3	101	1930	5	234	2956
$M_{2.2}$	2	58	1260	3	100	1665	5	240	3318
$M_{3.1}$	4	123	2323	6	231	3721	10	445	6487
$M_{3.2}$	2	60	1040	3	111	1858	5	250	2759
$M_{4.1}$	0	4	105	0	8	332	0	19	826
$M_{4.2}$	0	2	141	0	4	219	0	9	370
$M_{4.3}$	0	2	36	0	4	79	0	7	214
$M_{5.1}$	0	1	79	0	2	143	0	3	315
$M_{5.2}$	0	1	93	0	2	155	0	3	114
$M_{5.3}$	0	1	54	0	2	148	0	4	211
$M_{6.1}$	4	111	2020	5	215	3213	10	529	5659
$M_{6.2}$	4	119	2513	5	220	3303	10	473	5440
$M_{6.3}$	2	56	1064	3	115	1467	5	239	2605

Table A.4: *Execution times to produce a schedule using GR, LP and LR for m = 32 and 64. Values are in milliseconds.*

Query	m = 32			m = 64		
	GR	**LP**	**LR**	**GR**	**LP**	**LR**
J_1	9	1230	12249	13	3523	13589
J_2	9	1255	8119	12	6265	21971
J_3	8	4785	11281	11	12960	30409
J_4	8	3183	28915	11	7396	43145
J_5	8	933	6071	11	2541	23928
J_6	7	1842	9419	10	7494	12184
J_7	8	2595	10993	11	14909	22766
J_8	2	417	5592	3	848	13517
J_9	2	540	2052	3	1031	3822
J_{10}	0	9	243	0	24	145
J_{11}	6	564	9615	8	2126	28335
J_{12}	12	1265	15771	17	3202	34902
J_{13}	12	1209	15252	15	3432	27108
J_{14}	8	874	11557	11	2906	20745
J_{15}	5	461	10407	7	1409	19719
J_{16}	5	473	7807	7	1767	19017
J_{17}	5	498	8188	6	1389	15535
J_{18}	10	1171	7951	13	2880	16041
J_{19}	6	706	11487	8	1930	19732
J_{20}	6	619	11479	8	2462	18071
$M_{1.1}$	0	9	360	0	24	2067
$M_{1.2}$	0	4	66	0	7	2559
$M_{1.3}$	0	6	266	0	14	125
$M_{2.1}$	6	537	8137	8	1444	16889
$M_{2.2}$	6	681	5299	7	2442	9158
$M_{3.1}$	12	1175	11326	17	3179	18646
$M_{3.2}$	6	656	4873	8	1867	11137
$M_{4.1}$	1	42	1273	1	111	1894
$M_{4.2}$	0	17	357	0	56	1736
$M_{4.3}$	0	18	5754	0	51	1481
$M_{5.1}$	0	8	327	0	20	647
$M_{5.2}$	0	6	222	0	17	251
$M_{5.3}$	0	7	339	0	26	999
$M_{6.1}$	12	1249	10111	16	3549	18187
$M_{6.2}$	11	1110	12034	15	2962	19981
$M_{6.3}$	6	586	5700	7	1826	9810

APPENDIX B

A Complete Example of Parametric Analysis

In this appendix, we illustrate the complete parametric analysis (PA) process applied to a non-trivial SM instance (M_{pa}), which is a modified version of a practical multiway spatial join query. We changed the values of the processing and communication costs to improve the illustration of the process. The instance has integer w_j's and c_{ij}'s, $n = 69$ jobs, $m = 16$ machines, and $u_i = 0$, $i = 1, \ldots, 16$. Table B.3 presents the values of each parameter of this instance. For the sake of completeness, we restate here the first iteration of the process, also described in Chapter 5.

B.1 Finding Bounds for PA

In this section, we identify the range of f to which we shall apply the PA process. The upper bound (UB) of the makespan x_0^{UB} is obtained by setting $f = 0$ and solving SM_0 to identify a feasible schedule. As was observed in Chapter 5, SM_0 can be solved to optimality by greedily assigning each job to its lowest cost processor. The result here is a schedule with makespan $x_0^{UB} = 35$ and communication cost 7152. Using (5-1), we identify the lowest possible makespan of any feasible schedule, x_0^{LB}:

$$x_0^{LB} = \left\lceil \frac{\sum_{j=1}^{n} w_j}{m} \right\rceil = \left\lceil \frac{362}{16} \right\rceil = 23.$$

To identify f', the upper bound on f for the PA, we solve the instances SM_q for $q = 0, 1, 2$, that is, for $x_0 = 23, 24, 25$. The first feasible schedule occurs when $q_1 = 0$, with $(Z^*, x_0^*, C^*) = (8597, 23, 7470)$, i.e., a total cost of 8597, a makespan of $x_0 = x_0^{LB} = 23$, and a communication cost of 7470.

The second feasible schedule occurs when $q_2 = 2$, with $(Z^*, x_0^*, C^*) = (8597, 25, 7372)$, a total cost of 8597, a makespan of $x_0 = x_0^{LB} + 2 = 25$ and a communication cost of $Z_q(2) = 7372$. There is no feasible schedule for $q = 1$, with makespan $x_0 = 24$.

Using (5-3), we have that:

$$f' = \frac{Z_q(q_1) - Z_q(q_2)}{q_2 - q_1} = \frac{7470 - 7372}{2 - 0} = 49.$$

Therefore, the feasible schedule with $x_0 = 23$, communication cost 7470 and total cost $23f + 7470$ is optimal for all $f \geq 49$. So, f' is fixed at 49. As $x_0^{LB} = 23 < 35 = x_0^{UB}$, the proposed PA process is applied within the range $0 \leq f \leq 49$. The challenge is to find optimal schedules, their total costs $Z^*(f)$, makespans and communication costs, for each piecewise segment in this range.

B.2 Starting the PA Process

For the sake of brevity, let us denote as $(Z^*(f_i), x_0^i, C_i)$ the values for a feasible schedule obtained for f_i, where i indicates the sequence of f values investigated in the course of PA process. For example, for the first value of f below (f_1), we have $(Z^*(f_i), x_0^i, C_i) = (7152, 35, 7152)$.

Recall that for each iteration of the PA process we start with given feasible schedules for two distinct given values of f and compute an optimal schedule corresponding to an intermediate value of f. (Note, in some cases, the intermediate optimal schedule may turn out to be identical with one of two given feasible schedules). The result of the first iteration is presented in the following table. We already have two given (optimal) schedules, corresponding to $f_1 = 0$ and $f_2 = 49$. The schedules are shown in the second column of the table $(Z^*(f_i), x_0^i, C_i)$, and their respective points are in the third column $(P = (f_i, Z^*(f_i)))$. The initial interval of uncertainty for $Z^*(f)$ is the triangle $P_1P_4P_2$, depicted in Figure B.1A. By applying Proposition 5.7, we obtain UB(f), and by using Proposition 5.8, we get the LB(f)'s. The intermediate value of f (column interm. f) is obtained using (5-4), and finally, by computing the optimal schedule for it, we can determine which case of Figure 5.2 is pertinant.

f	$(Z^*(f_i), x_0^i, C_i)$	$P = (f_i, Z^*(f_i))$	**UB**(f)	**LB**(f)	**interm.** f
$f_1 = 0$	(7152, 35, 7152)	$P_1(0, 7152)$	$35f + 7152$	$29.49f + 7152$	$f_3 = 22$
$f_2 = 49$	(8597, 25, 7372)	$P_2(49, 8597)$	$25f + 7372$		

The optimal schedule for f_3 is:

f	$(Z^*(f_i), x_0^i, C_i)$	$P = (f_i, Z^*(f_i))$	**UB**(f)	**LB**(f)
$f_3 = 22$	(7870, 32, 7166)	$P_3(22, 7870)$	$32f + 7166$	P_1P_3: $32.64f + 7152$ P_3P_2: $26.92f + 7744$

The optimal schedule for f_3 provides a point $P_3 = (22, 7870)$ strictly between P_4 and P_5 (Figure B.1A), which corresponds to case (a) in Figure 5.2. Thus, by applying Propositions 5.7 and 5.8, we define new upper and lower bounds to reduce the area of uncertainty for $Z^*(f)$, as illustrated in Figure B.1B. The area of uncertainty for $Z^*(f)$ are now the triangles $P_1P_6P_3$ (Figure B.1C) and $P_3P_7P_2$ (Figure B.1D).

The PA process continues recursively in the triangles $P_1P_6P_3$ and $P_3P_7P_2$ until, with reference to Figure 5.2, case (b) occurs and a breakpoint is identified, or case (c) occurs and there is no breakpoint in the particular range of f at hand. For each iteration of the method described in the following sections, we employed a breadth-first search of f. These cases mentioned throught relate to those in Figure 5.2. To begin the process, we expanded the root node, resulting in two children nodes (one for each triangle). Both are visited in the second iteration.

B.3 The Second Iteration

The second iteration starts with the two triangles $P_1P_6P_3$ and $P_3P_7P_2$. The optimal solutions for f_1 and f_2 establish the upper bounds for the triangle $P_1P_6P_3$. By applying (5-4) to them, we have $f_4 = \frac{7166-7152}{35-32} = \frac{14}{3}$. The optimal schedule for f_4, presented in the following table, relates to the point $P_8 = (34, 7156)$, which corresponds to case (a). Thus, by applying Propositions 5.7 and 5.8, we identify new upper and lower

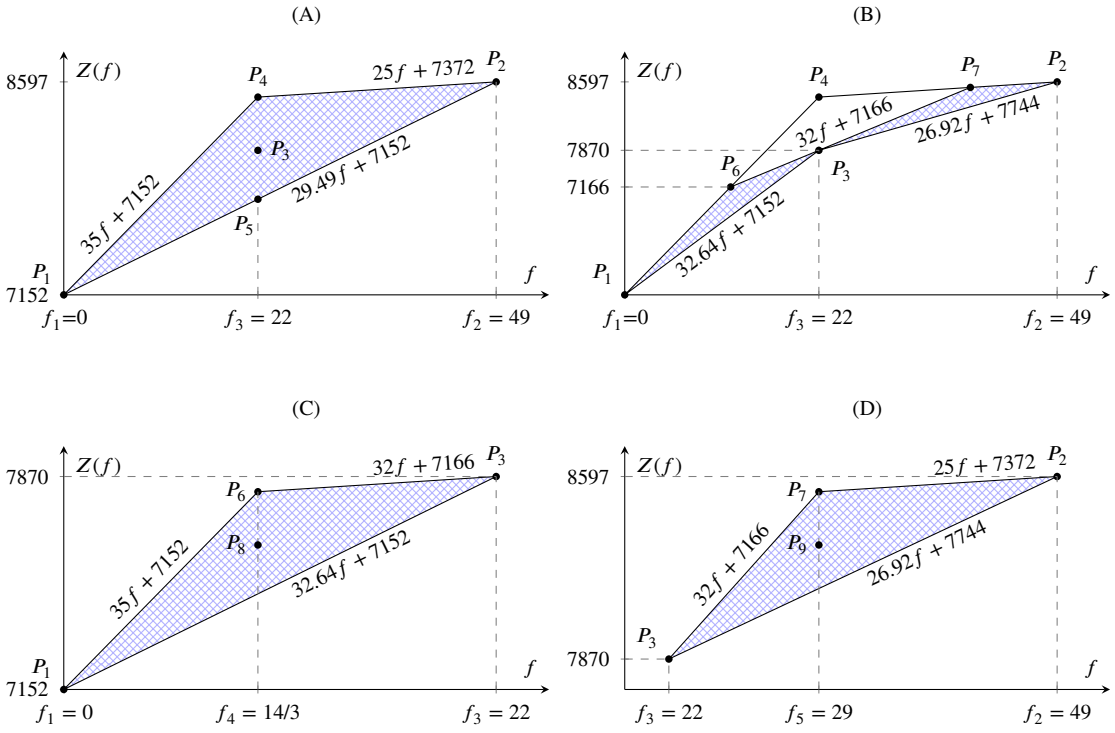


Figure B.1: *Uncertainty area in the first iteration of the parametric analysis for M_{pa}.*

bounds to reduce the area of uncertainty for $Z^*(f)$, as illustrated in Figure B.2A. The intervals of uncertainty of $Z^*(f)$ are now the triangles $P_1P_{10}P_8$ and $P_8P_{11}P_3$.

f	$(Z^*(f_i), x_0^i, C_i)$	$P = (f_i, Z^*(f_i))$	**UB**(f)	**LB**(f)
$f_4 = \frac{14}{3}$	(7314.67, 34, 7156)	P_8(14/3, 7314.67)	$34f + 7156$	P_1P_8: $34.86f + 7152$ P_8P_3: $32.04f + 7165.02$

The upper bounds for the triangle $P_3P_7P_2$ are defined by the optimal schedules for f_2 and f_3. By applying (5-4) in them, we have $f_5 = \frac{7372-7166}{32-25} = \frac{206}{7}$. For the sake of brevity, let we round it to $f_5 = 29$. The optimal schedule for f_5, presented in the following table, provides another case (a) again with the point $P_9 = (29, 8089)$. Thus, by tightening the upper and lower bounds (the UB(f)'s and LB(f)'s), we reduce the area of uncertainty for $Z^*(f)$. The intervals of uncertainty for this range of f are now the triangles $P_3P_{12}P_9$ and $P_9P_{13}P_2$, as illustrated in Figure B.2B.

f	$(Z^*(f_i), x_0^i, C_i)$	$P = (f_i, Z^*(f_i))$	**UB**(f)	**LB**(f)
$f_5 = 29$	(8089, 31, 7190)	P_9(29, 8089)	$31f + 7190$	P_3P_9: $31.29f + 7181.71$ P_9P_2: $25.4f + 7352.4$

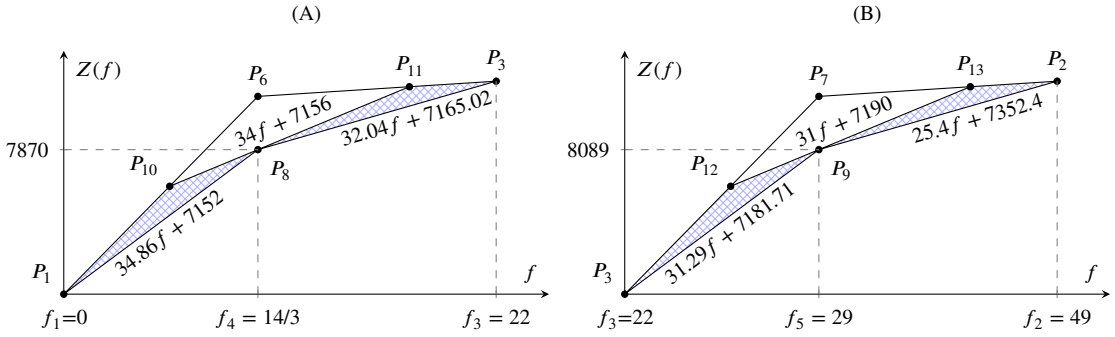


Figure B.2: *Reduction of the area of uncertainty in the second iteration of the parametric analysis for* M_{pa}.

B.4 The Last Iteration

We start this iteration with four triangles: (1) $P_1P_{10}P_8$, (2) $P_8P_{11}P_3$, (3) $P_3P_{12}P_9$, and (4) $P_9P_{13}P_2$. For (1), the upper bounds are given by the optimal schedules for f_1 and f_4 and by applying (5-4) to them we have $f_6 = \frac{7156-7152}{35-34} = 4$. Analogously, for (2), f_4 and f_3 define the upper bounds, and we have $f_7 = \frac{7166-7156}{34-32} = 5$. For (3), f_3 and f_5 define the upper bounds, and we have $f_8 = \frac{7190-7166}{32-31} = 24$. Finally, for (4), the upper bounds are given by the optimal schedules for f_5 and f_2, and we have $f_9 = \frac{7372-7190}{31-24} = \frac{182}{6} \approx 31$. The optimal schedules for the new intermediary f's, $f_{6,\ldots,9}$, are:

f	$(Z^*(f_i), x_0^i, C_i)$	$P = (f_i, Z^*(f_i))$
$f_6 = 4$	(7291, 35, 7152)	$P_{14}(4, 7291)$
$f_7 = 5$	(7326, 34, 7156)	$P_{15}(5, 7326)$
$f_8 = 24$	(7934, 32, 7166)	$P_{16}(24, 7934)$
$f_9 = 31$	(8147, 29, 7248)	$P_{17}(31, 8147)$

In this iteration, the PA identifies four breakpoints via case (b) (See Figure B.3). The point P_{14} coincides with P_{10}. Thus, there is a breakpoint at f=4 (Figure B.3A). The point P_{15} coincides with P_{11} and identifies a breakpoint at f=5 (Figure B.3B). The point P_{16} coincides with P_{12} determining a breakpoint at f=24 (Figure B.3C). Finally, point P_{17} coincides with P_{13}, defining a breakpoint at f=31 (Figure B.3D).

As the two points at $f = 24$ and $f = 31$ collapsed with case (b) and we have a distinct LB(f) for them, there is another breakpoint at $f = 29$. This does not occur for $f = 14/3$ and $f = 22$ because the LB(f)'s for them are the same. One last breakpoint occurs at $f = 49$, the endpoint of the range for the PA. As there is no more area of uncertainty to refine, the PA process is terminated. The next section presents the complete PA results.

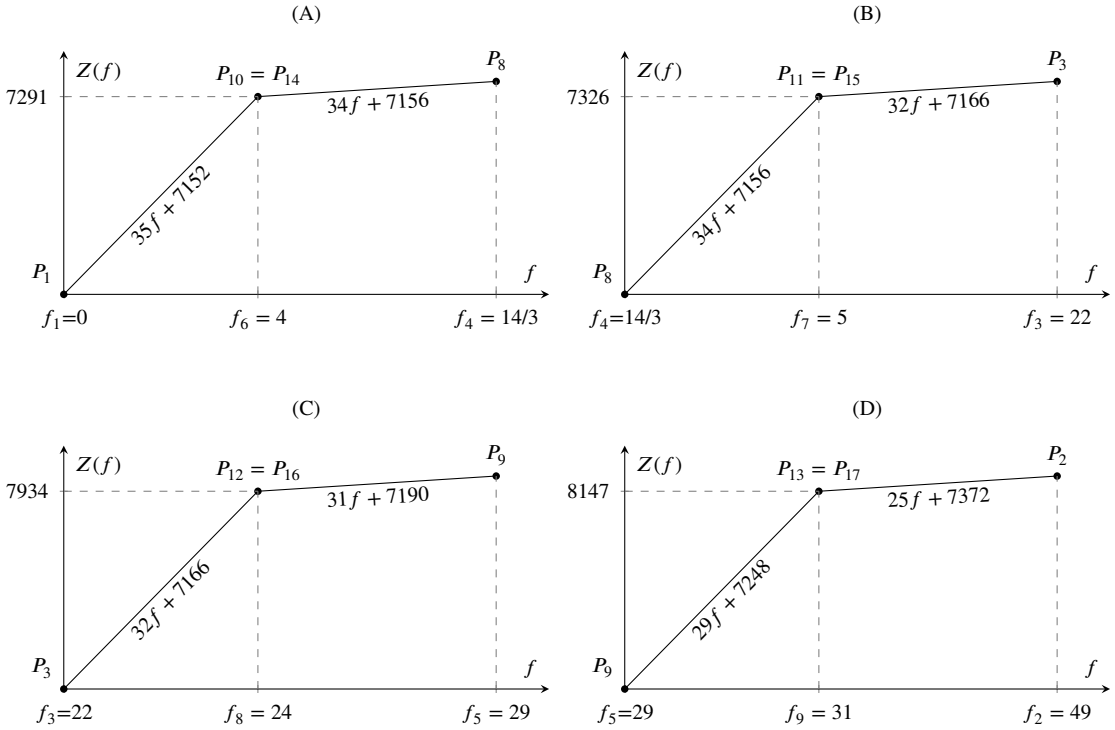


Figure B.3: *Breakpoints identified in the last iteration of the parametric analysis for M_{pa}.*

B.5 The Six Breakpoints

We found optimal schedules for nine values of f, as indicated in Table B.1. The process produced a complete PA of f for M_{pa}. There are six breakpoints: $f = 4$, 5, 24, 29, 31, and 49. At each of these breakpoints, there are two distinct optimal schedules. Table B.2 summarizes the results, and Figure B.4 depicts the shape of $Z^*(f)$.

Table B.1: *The fs examined in the parametric analysis of M_{pa}.*

Breakpoint?	f	$Z^*(f)$	x_0^*	C^*	x_0^*	C^*
No	0	7152	35	7152	-	-
Yes	4	7292	35	7152	34	7156
No	14/3	7314.67	34	7156	-	-
Yes	5	7326	34	7156	32	7166
No	22	7870	32	7166	-	-
Yes	24	7934	32	7166	31	7190
Yes	29	8089	31	7190	29	7248
Yes	31	8147	29	7248	25	7372
Yes	49	8597	25	7372	23	7470

Table B.2: *A summary of the parametric analysis of f for M_{pa}*

Range of f	x_0^*	$C(f)$	$Z^*(f)$	**Range of $Z^*(f)$**
$0 \leq f \leq 4$	35	7152	$35f + 7152$	$7152 \leq Z^*(f) \leq 7292$
$4 \leq f \leq 5$	34	7156	$34f + 7156$	$7292 \leq Z^*(f) \leq 7326$
$5 \leq f \leq 24$	32	7166	$32f + 7166$	$7326 \leq Z^*(f) \leq 7934$
$24 \leq f \leq 29$	31	7190	$31f + 7190$	$7934 \leq Z^*(f) \leq 8089$
$29 \leq f \leq 31$	29	7248	$29f + 7248$	$8089 \leq Z^*(f) \leq 8147$
$31 \leq f \leq 49$	25	7372	$25f + 7372$	$8147 \leq Z^*(f) \leq 8597$
$49 \leq f$	23	7470	$23f + 7470$	$8597 \leq Z^*(f)$

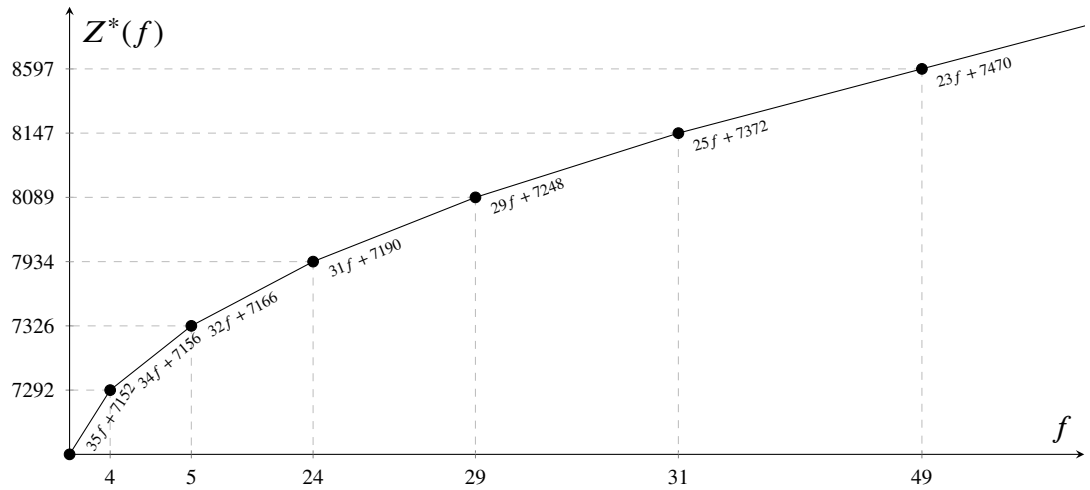


Figure B.4: *The parametric analysis of M_{pa}.*

Table B.3: *Values of w_j and c_{ij} in the M_{pa} instance.*

		c_{ij}'s															
j	w_j	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10	i=11	i=12	i=13	i=14	i=15	i=16
1	4	8	8	8	8	8	6	2	8	8	8	8	8	8	8	8	8
2	7	102	102	102	102	102	102	102	102	102	102	102	102	102	58	44	102
3	8	152	152	152	152	152	152	152	152	152	152	152	152	126	82	94	152
4	6	14	14	14	14	14	14	8	8	14	14	14	14	14	14	14	12
5	4	24	24	24	24	24	22	16	18	24	24	24	24	24	24	16	24
6	6	42	42	42	42	24	20	40	42	42	42	42	42	42	42	42	42
7	7	154	188	188	188	188	188	188	188	188	188	188	188	188	188	130	94
8	9	118	118	118	118	118	118	118	118	118	118	118	118	118	118	60	58
9	8	182	182	182	168	182	182	182	182	182	182	182	182	182	156	98	122
10	2	144	144	136	144	144	144	144	144	144	144	144	144	104	74	118	144
11	3	186	180	186	186	186	186	186	186	186	186	186	142	96	140	186	186
12	1	94	94	94	94	94	94	94	94	94	94	94	50	44	94	94	94
13	9	14	14	14	14	14	14	14	0	14	14	14	14	14	14	14	14
14	8	18	18	18	18	18	18	14	4	18	18	18	18	18	18	18	18
15	4	52	52	52	52	52	6	48	52	52	52	52	52	52	52	52	52
16	7	180	180	180	180	180	180	180	180	180	180	180	180	180	180	180	0
17	1	306	306	306	306	306	306	306	306	306	292	306	306	306	306	194	126
18	7	288	288	288	288	288	288	288	288	268	288	288	288	288	134	174	288
19	1	372	372	372	372	372	372	372	328	372	372	372	372	196	218	372	372
20	1	214	214	214	214	214	214	174	214	214	214	214	214	40	214	214	214
21	2	292	292	292	292	292	288	292	292	292	292	292	178	118	292	292	292
22	8	16	16	16	16	16	16	10	6	16	16	16	16	16	16	16	16
23	9	48	48	48	48	48	10	42	48	48	48	48	46	48	48	48	48
24	8	352	348	352	352	352	352	352	352	352	352	352	352	352	352	200	154
25	3	334	334	334	334	334	334	334	334	334	334	334	334	334	152	184	334
26	1	428	428	428	428	428	428	428	428	428	428	428	428	216	244	428	394
27	3	226	226	226	226	226	226	226	226	226	226	226	226	14	226	210	226
28	2	410	410	410	410	410	410	410	410	410	410	410	218	200	404	410	410
29	8	294	294	294	294	294	294	294	294	294	294	230	102	256	294	294	294
30	6	32	32	32	32	24	14	28	32	32	32	32	32	32	32	32	32
31	8	52	52	52	36	28	40	52	52	52	52	52	52	52	52	52	52
32	3	160	160	160	160	160	160	158	160	160	160	160	160	160	160	78	82
33	7	352	352	352	352	352	288	352	352	352	352	352	352	352	286	208	274
34	9	346	346	346	346	276	346	346	346	346	346	346	346	272	206	282	346
35	1	348	348	348	314	348	348	348	348	348	348	348	268	188	274	348	348
36	2	196	196	166	196	196	196	196	196	196	196	196	114	110	196	196	196
37	4	458	458	358	458	458	458	458	458	458	458	364	278	370	458	458	458
38	7	244	286	286	286	286	286	286	286	286	264	162	188	286	286	286	286
39	2	98	98	98	98	98	52	98	98	98	76	68	98	98	98	98	98
40	9	78	78	78	78	78	78	78	78	64	44	48	78	78	78	78	78
41	8	174	174	174	174	174	174	174	174	80	174	174	174	174	174	94	174
42	9	304	304	304	304	304	304	304	240	210	304	304	304	304	236	222	304
43	2	278	278	278	278	278	278	202	214	278	278	278	278	202	212	278	278
44	8	308	308	308	308	308	230	232	308	308	308	308	226	234	308	308	308
45	5	162	162	162	162	162	84	162	162	162	162	158	80	162	162	162	162
46	8	346	346	346	346	256	268	346	346	346	340	252	266	346	346	346	346
47	8	246	246	246	226	156	246	246	224	246	222	152	246	246	246	246	246
48	5	58	58	58	38	58	58	58	58	58	36	42	58	58	58	58	58
49	6	68	68	56	50	68	68	68	68	56	46	68	68	68	68	68	68
50	1	94	168	168	168	168	168	168	168	76	168	168	168	168	168	168	168
51	4	220	296	296	296	296	296	296	232	202	296	296	296	296	296	296	232
52	7	276	276	276	276	276	276	202	214	276	276	276	276	276	276	202	214
53	2	166	166	166	166	166	88	166	166	166	166	166	166	166	78	166	166
54	8	342	342	342	342	342	254	266	342	342	342	342	342	342	254	256	342	342
55	7	220	220	220	202	132	220	220	220	220	220	220	196	132	220	220	220
56	4	60	60	48	40	60	60	60	60	60	60	54	36	58	60	60	60
57	3	162	162	162	88	162	162	162	162	162	162	162	162	162	74	160	162
58	1	320	320	250	246	320	320	320	320	320	320	320	320	232	232	320	320
59	7	200	182	130	200	200	200	200	200	200	200	200	178	112	200	200	200
60	9	144	144	144	72	144	144	144	74	144	144	144	144	144	144	144	144
61	6	286	286	216	212	286	286	214	216	286	286	286	286	286	286	286	286
62	9	170	152	100	170	170	160	100	170	170	170	170	170	170	170	170	170
63	3	256	256	256	256	256	256	256	186	194	256	256	256	194	194	256	256
64	1	142	142	142	142	142	142	142	72	142	142	142	132	82	142	142	142
65	7	268	268	268	268	268	268	196	198	268	268	266	206	206	268	268	268
66	6	146	146	146	146	146	136	74	146	146	146	144	84	146	146	146	146
67	7	66	66	66	66	66	66	66	66	64	66	66	62	4	66	66	66
68	2	130	130	130	130	130	130	130	130	130	130	128	66	70	130	130	130
69	4	68	68	68	68	68	68	68	68	68	68	64	6	68	68	68	68