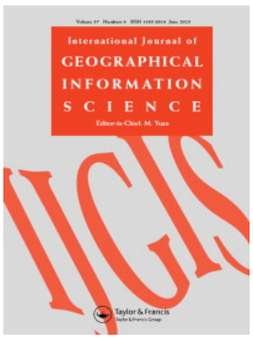

International Journal of Geographical Information Science

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/tgis20

Scheduling distributed multiway spatial join queries: optimization models and algorithms

Thiago Borges de Oliveira, Fábio M. Costa, Les R. Foulds & Humberto J. Longo

To cite this article: Thiago Borges de Oliveira, Fábio M. Costa, Les R. Foulds & Humberto J. Longo (2023) Scheduling distributed multiway spatial join queries: optimization models and algorithms, International Journal of Geographical Information Science, 37:6, 1388-1419, DOI: 10.1080/13658816.2023.2170380

To link to this article: https://doi.org/10.1080/13658816.2023.2170380

INTERNATIONAL JOURNAL OF GEOGRAPHICAL INFORMATION SCIENCE
2023, VOL. 37, NO. 6, 1388–1419
https://doi.org/10.1080/13658816.2023.2170380

Taylor & Francis
Taylor & Francis Group

RESEARCH ARTICLE

Scheduling distributed multiway spatial join queries: optimization models and algorithms

Thiago Borges de Oliveira[a], Fábio M. Costa[b], Les R. Foulds[b] and Humberto J. Longo[b]

[a]Unidade Acad. de Ciências Exatas e Tecnológicas, Universidade Federal de Jataí, Jataí, Brazil;
[b]Instituto de Informática, Universidade Federal de Goiás, Goiânia, Brazil

ABSTRACT

Multiway spatial joins are a commonly occurring and fundamental type of query for spatial data processing. This article presents models and algorithms to schedule this type of query in distributed database systems while attempting to strike a balance between makespan and communication costs. We propose three algorithms based on combinatorial optimization methods: the well-known linear relaxation technique of rounding a solution generated by linear programming (LP), a more sophisticated Lagrangian Relaxation method (LR), as well as a greedy heuristic (GR) for baseline comparison. Our evaluation shows that a schedule built using GR consumes, on average, 22% more processing and communication resources than a more elaborate schedule constructed via the LR method, when scheduling a query for 64 machines. The schedule provided by LR is also, on average, an order of magnitude closer to the optimal schedule for a query compared to GR. We show that scheduling Gigabyte-size multiway queries before execution can reduce its processing time by an order of magnitude compared to state-of-the-art frameworks for spatial data processing that do not have this capability, and can significantly reduce the amount of shuffled data in the network.

ARTICLE HISTORY
Received 14 February 2022
Accepted 16 January 2023

KEYWORDS
Multiway spatial join; Distributed query scheduling; Lagrangian relaxation

1. Introduction

The amount of spatial data available in many areas of human endeavor has significantly increased with the popularization of GPS-enabled devices. This includes geotagged satellite images and maps, sensor data from IoT (Internet of Things) devices, open data, and census data. Typically, such data are continuously collected and organized in thematic datasets to, for instance, support decision-making and improve the efficiency of market intelligence and logistics. An important kind of spatial query used to process such significant amounts of spatial data is the *multiway spatial join* (Mamoulis and Papadias 2001a). Multiway spatial join queries (MSJQ) are essential in

CONTACT Thiago Borges de Oliveira thborges@ufj.edu.br
Supplemental data for this article is available online at https://doi.org/10.1080/13658816.2023.2170380
This article has been republished with minor changes. These changes do not impact the academic content of the article.

several application fields, including geography [e.g. finding the animal species that survived in a preservation area damaged by fire or finding all the forests crossed by a river in each state (Du *et al.* 2017)], VLSI [e.g. identifying circuits that constitute a particular topological configuration (Mamoulis and Papadias 2001a)], and digital medical imaging [e.g. analyzing microscopy whole slide images (WSI) of the brain to identify tumor subtypes and characteristics, with a typical image containing 10^{10} pixels, hundreds of millions of features, and thousands of images being generated daily in a moderate-size healthcare operation (Aji *et al.* 2012)].

The complexity of the computational geometry algorithms used in MSJQ often causes long query execution times (i.e. makespan). For practical spatial datasets, this usually necessitates the processing of the related queries in distributed systems and the partitioning of the datasets using spatial columns to split the processing cost among many machines, consequently reducing query response time (Vu *et al.* 2020). However, there are open issues related to the processing of multiway spatial join queries in distributed environments (Huang *et al.* 2011). One such important issue, which is addressed in this paper, refers to the efficient scheduling of query fragments to improve the overall processing and communication costs. The partitioning of spatial datasets often creates significant challenges due to their skewed nature, which may cause unbalanced query execution. Further, besides considering the local CPU and I/O costs in a distributed system, the selection of execution plans must take into account the effect of data partitioning on the communication between the processors. Thus, the challenge when choosing a schedule for a query is to find an appropriate division of all work among processors, with regard to both the bandwidth limit of the network interface and the load on the CPUs.

Selecting execution plans and identifying query schedules to efficiently process MSJQ in distributed systems are critical 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 improve the scheduling of the execution plans must be specified, taking into account the specifics of spatial data and the characteristics of distributed systems. Spatial data analysis in such situations can significantly improve the scalability of spatial data processing, especially in today's environment of cloud computing platforms, taking advantage of elasticity, and pay-as-you-go offers.

The issue addressed here relates to the copy selection and sub-query allocation problem described in the distributed database literature, dealing with the allocation and copying of entire relations or fragments of horizontally partitioned relations (Özsu and Valduriez 2011). Early work suggested the use of exhaustive enumeration or heuristics to cope with the NP-hard complexity of the problem (Yu and Chang 1984). In general, the proposed solutions assume a controlled number of disjoint relation fragments and a small number of replicas.

In this paper, we consider a generalization of this problem in which even a single step of a multiway query may have a relatively large number of data partitions to handle, given the size of spatial datasets involved. Furthermore, the fragments (or data partitions) are non-disjoint by nature, due to the intrinsic characteristics of spatial data, and the processing of each query predicate may be split into several processors in a distributed system. This generalized version of the problem adheres to recently

proposed models for data processing in distributed platforms, such as MapReduce (Dean and Ghemawat 2008) and Spark (Zaharia *et al.* 2016). However, the need for methods that consider both the optimization of queries and more flexible ways to partition data in such platforms has been previously identified as a future research direction in the literature (Doulkeridis and Nørvåg 2014).

In this direction, we propose a formal bi-objective linear model of the problem of scheduling distributed multiway spatial join query plans. The model aims to minimize the weighted sum of a query's makespan and the total communication cost. We present three methods, based on combinatorial optimization, to identify viable solutions for the model: a greedy heuristic algorithm (GR), similar to those commonly used in related work; the linear relaxation technique of rounding a solution generated by linear programming, known as Linear Relaxation (LP) (Vazirani 2001); and the more sophisticated Lagrangian Relaxation (LR) method (Fisher 2004).

We also measured the benefits of query scheduling using an experimental distributed query engine. We ran some Gigabyte-size queries frequently reported in the literature and compared the execution time with that reported for related software using state-of-the-art frameworks, such as MapReduce (Dean and Ghemawat 2008) and its in-memory and more general counterpart, the Spark engine (Zaharia *et al.* 2016). We show that scheduling query plans before query execution can significantly reduce the execution time of queries in a distributed environment.

Throughout the text, we assume a basic understanding of linear programming (LP) and some elementary properties of linear models. We attempt to achieve a balance between formalism and application, and present a brief introduction to Lagrangian Relaxation in Section 2.4. For more comprehensive coverage of linear optimization, the reader is referred to Bazaraa *et al.* (2009) and Vazirani (2001, Chapter 12).

The remainder of this article is organized as follows. In Section 2, we present background concepts on multiway spatial join, its processing in distributed systems, previously reported work on spatial query processing built on top of the MapReduce and Spark frameworks, as well as some of the concepts of linear optimization used in our algorithms. We propose new models and solution algorithms for scheduling multiway spatial join queries in Section 3 and present their evaluation in Section 4. Finally, we state our conclusions and ideas for future work in Section 5.

2. Background and related work

2.1. Multiway spatial join

A simple or pairwise spatial join query performs 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 $a\ \theta\ b$ (Brinkhoff *et al.* 1996).

A multiway spatial join query, in turn, is a set of interconnected spatial join queries with an arbitrary number n of input datasets, $n > 2$ (Papadias and Arkoumanis 2002b). It can be represented as a graph $G = (D, P)$ with node set D and edge set P, where each node represents a distinct dataset, and each edge represents a join predicate. Formally, given a set of datasets $D = \{D_1, ..., D_n\}$, each containing a set of records

$r^i_1, \ldots, r^i_{m_i}$, $1 \leq i \leq n$ and m_i being the cardinality of D_i, and a set of spatial predicates, $P = \{\theta_{ij} \mid \forall\ i,j,\ 1 \leq i,j \leq n\}$, the query retrieves all n-tuples $(r^1_p, \ldots, r^i_k, \ldots, r^j_l, \ldots, u^n_r)$ such that each predicate θ_{ij} holds when applied to its respective elements in the n-tuple, with p, k, l, and r referring to specific records of its respective datasets, $1 \leq k \leq m_i$, $1 \leq l \leq m_j$, and analogously for p and r (Papadias *et al.* 1999, 2001, Mamoulis and Papadias 2001a, Papadias and Arkoumanis 2002a).

There are many distinct ways to process a multiway spatial join query, called execution plans. Each execution plan defines a distinct order of processing the datasets and which algorithms to apply in each step to compute the final result of the query. Mamoulis and Papadias (2001a) showed that the number of ways to process a query on serial processors (i.e. non-parallel, non-distributed) is a function of the query type, the number of input datasets, and the number of different join algorithms at each query step. They concluded that the number of equivalent execution plans for a query is exponential in the number of datasets. Although they are all equivalent and preserve the same query semantics, each of them requires a different amount of computing resources to produce the final result. An inexpensive execution plan for a query is, in general, orders of magnitude better than an expensive plan, regarding its processing cost. Thus, a great amount of effort has been dedicated to proposing cost-based optimizers that can select a relatively cheap execution plan for a query based on its estimated computational costs (Mishra and Eich 1992).

Some authors have proposed functions that predict the cost of spatial join queries, such as those by Sivasubramaniam (2001), Fornari *et al.* (2006), and Roh *et al.* (2010), as well as methods to combine them to predict the cost of multiway spatial join queries (Mamoulis and Papadias 2001a). Such methods and functions, i.e. a *cost model*, predict the I/O and CPU resources needed by the join algorithms applied to an execution plan, assuming that the data fills the spatial extent uniformly. However, the uniformity assumption does not hold for practical spatial datasets and may cause the selection of bad execution plans, especially in the presence of dataset skewness. To improve plan cost estimation in this condition, Mamoulis and Papadias (2001b) proposed a uniform (grid) histogram that divides the spatial extent of the dataset into disjoint cells of fixed size that accounts the density of spatial objects and other metadata about them, such as its average length. Additionally, many histogram techniques were proposed to improve the selectivity estimation—the main component of plan cost estimation (e.g. Acharya *et al.* 1999, Sun *et al.* 2006, Cheng *et al.* 2013).

Careful use of the cost model can improve estimates by gathering additional metadata for complex spatial objects, such as the kind of object stored, the area of polygons and length of polylines, and the number of spatial data points to estimate the communication volume in distributed systems. Furthermore, specific estimation formulas and precise histogram techniques are often decidedly helpful for improving selectivity estimation when joining polygons and polylines, as reported by de Oliveira *et al.* (2017) and improved by de Oliveira (2017).

2.2. Distributed execution of multiway spatial join

With the increasing availability of large spatial datasets, many algorithms have been proposed to execute spatial join and multiway spatial join queries in parallel and distributed environments (e.g. Patel and DeWitt 2000, Luo *et al.* 2002, Gupta and Chawda 2014, Du *et al.* 2017, Sabek and Mokbel 2017, Yu *et al.* 2018, Eldawy *et al.* 2021). Most of these algorithms use a disjoint data partitioning strategy (declustering) to create groups of spatial objects, called *data partitions*. A frequently proposed way of generating data partitions is to group objects by location, i.e. their intrinsic geographic location. Before or during the join execution, a routine assigns a set of partitions (query fragments) to a particular processor that performs the query over it. Thus, the number of data partitions is a key parameter of a distributed query processing system as it determines the level of parallelism. However, in general, this number is determined empirically, for the data being processed. Eldawy *et al.* (2015) presented and evaluated a comprehensive set of data partitioning schemes for spatial data.

In our work, the cost model mentioned in the previous section was used to compute a spatial grid histogram for each dataset, with a specific granularity based on its metadata. The histogram, in turn, establishes the number of data partitions for the distributed system. 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. Furthermore, the strategy does not require the repartitioning of data in each join query performed, in contrast to some of the related work investigated. Further, the histogram acts as a distributed index structure, which stores the metadata about the datasets—used to estimate the cost of processing a query—and also supports the creation and assignment of data partitions in the distributed environment. Due to space limitations, we refer the reader to de Oliveira (2017) for further details.

Another issue regarding the distributed processing of MSJQ is how to schedule the query fragments, or jobs, to processors in a way that achieves both a load-balanced query execution and low network usage. The main difficulty in this regard is that a data partition must be aligned with others, i.e. data partitions of distinct datasets but from the same spatial region must be processed together on the same processor. The model we propose in this paper addresses this issue and is described in Section 3.

We focus here on a hybrid optimization technique that splits the execution plan into two parts: (i) a static plan to determine the access methods to use and the order of dataset processing, and (ii) an execution plan, generated at run time, to determine where the jobs will be executed and, consequently, where the data partitions will be copied from. This way, we can focus on query scheduling (ii) and use previously reported results for well-studied problems in (i), such as a plan enumeration explicitly designed for multiway spatial joins (Mamoulis and Papadias 2001a).

2.3. Earlier work on the processing of spatial data in distributed systems

Recently, increasing attention has been focused on the processing of spatial data in distributed systems and several studies on this topic have been published (see Eldawy and Mokbel (2016) for a survey). In this section, we briefly describe previously reported work on the topic that relates to the issues addressed in this article.

Earlier studies based on MapReduce focused on how to make design decisions concerning the underlying 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 reduction phase. Examples of such studies can be found in SJMR (Zhang *et al.* 2009), VegaGiStore (Zhong *et al.* 2012), and SpatialHadoop (Eldawy and Mokbel 2015). These studies, however, only support the processing of pairwise spatial join queries, not providing strategies to process multiway queries. The work presented by Aji *et al.* (2012), and later improved on by the same authors (Aji *et al.* 2013), although addressing multiway spatial join, used the default MapReduce load balancer,[1] and focused on dividing the load into evenly-sized tasks. This is also the case for the work of Gupta *et al.* (2013), improved by Gupta and Chawda (2014). The default load balancer algorithm of MapReduce assigns tasks to available slots in the cluster in a greedy way and requires tasks to have evenly spread loads to perform a balanced execution (Kwon *et al.* 2012). As discussed earlier, spatial data is, by nature, non-uniform and thus, the occurrence of straggler tasks is expected, resulting in unbalanced execution. Although other algorithms that improve the execution of non-uniform tasks in MapReduce frameworks are available (e.g. Afrati and Ullman 2011, Moseley *et al.* 2011, Verma *et al.* 2012, Bhattu *et al.* 2020), all reported studies maintain a focus on providing equally-sized-tasks.

The work on Spark for spatial data processing mainly supports the execution of pairwise spatial joins (e.g. You *et al.* 2015, Yu *et al.* 2015, Xie *et al.* 2016, Yu *et al.* 2018, Eldawy *et al.* 2021). However, it is possible to process a multiway join query in a system designed to process pairwise join queries by cascading the result of a previous step to the next. Nevertheless, it often incurs additional costs in collecting and redistributing the intermediate results. As far as we know, Du *et al.* (2017) proposed the only work that extends Spark to process multiway spatial join queries and does not incur such costs. Similarly to MapReduce, Spark also requires evenly-sized tasks to perform a balanced execution. Even so, the Spark API allows an application to set the preferred locations for a task, a feature that enables the scheduling of data to process on specific machines. However, all these studies focus on providing evenly-sized tasks and do not mention the API for preferred locations.

There is also a significant body of research on distributed database technology with some DDBMS supporting features for spatial data handling. However, the traditional focus of DDBMS systems is on supporting multi-query workloads at the inter-operator and inter-query levels of parallelism (Özsu and Valduriez 2011). Since our scope focuses on intra-operator parallelism and does not include multi-query workloads, we limited our comparison in this regard. An exception, however, is Distributed Secondo (Nidzwetzki and Güting 2017), a distributed, general-purpose DBMS which considers both the intra-operator level of parallelism and query optimization. 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 query execution in which underutilized machines are used to reduce the load of busy machines by randomly reassigning tasks. This strategy is similar to that used in MapReduce and Spark, and follows the principle of performing computation in the same location where the data is originally stored (i.e. reducing the

need to move data across the network). Although this scheduling strategy focuses on reducing communication, Moseley *et al.* (2011) and Verma *et al.* (2012) showed that, due to the combinatorial nature of the problem, constructing optimized job schedules can greatly reduce the makespan. In contrast, we consider in this article both the minimization of makespan and communication costs.

In general, the proposed techniques for the processing of MSJQ using the MapReduce and Spark frameworks (e.g. Aji *et al.* 2013, Du *et al.* 2017, Eldawy *et al.* 2017, Sabek and Mokbel 2017, Yu *et al.* 2018, Eldawy *et al.* 2021) do not consider the selection or even the scheduling of execution plans, which are well-established strategies to process multiway queries in traditional database systems. In turn, the kind of parallelism implemented by these frameworks, known as intra-operator parallelism, is a fundamental design principle responsible for the high scalability achieved and, generally, is not implemented in traditional distributed database systems as they focus mainly on intra-query parallelism (Özsu and Valduriez 2011). Recently, the lack of attention to database theory in the above-mentioned emerging frameworks has been criticized (e.g. Pavlo *et al.* 2009, Stonebraker *et al.* 2010) and a few surveys propose the integration of query optimizers as future work (e.g. Doulkeridis and Nørvåg 2014).

2.4. Lagrangian relaxation

In terms of computational complexity, the problem considered in this article is NP-Hard, which means that nontrivial, general, numerical instances of it are notoriously hard to solve. A common approach to deal with such problems is to solve a simplified version of it to obtain approximate solutions and bounds. The well-known technique of Lagrangian relaxation (LR) is suitable for such problems if their constraints can be divided into two sets:

- 'simple' constraints—when the problem consists of only these, it can be solved relatively easily, and
- 'difficult' constraints—when these are added, the problem becomes very hard to solve.

More detailed descriptions of LR have been given by several authors, including Fisher (1985, 2004), Bertsimas and Tsitsiklis (1997), and Klau and Reinert (2007). The main idea of LR is to relax the problem by removing the difficult constraints and placing them in the objective function, where they are assigned weights (the Lagrangian multipliers). Each weight represents a penalty that is added to the cost of any solution that does not satisfy the corresponding constraint. LR is often used for efficiently finding a bound on the value of the optimal solution, Z, to such problems. Sometimes, the bound equals Z and the use of LR leads to an optimal solution. Consider the following combinatorial optimization problem expressed as an integer linear program:

$$\begin{aligned} \text{Minimize } Z &= \boldsymbol{c}^{\boldsymbol{T}}\boldsymbol{x}, \\ \text{subject to} & \end{aligned} \tag{2.1}$$

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{2.2}$$

$$\boldsymbol{D}\boldsymbol{x} \leq \boldsymbol{e}, \tag{2.3}$$

$$\boldsymbol{x} \in \mathbb{Z}_{+}^{\boldsymbol{n}}. \tag{2.4}$$

where $\boldsymbol{x}$, $\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{D}$, and $\boldsymbol{e}$ are integral, with dimensions $n \times 1, m \times n, m \times 1, n \times 1, k \times n$, and $k \times 1$, respectively.

Assuming that the constraints given by (2.2) are difficult, the original problem (2.1)–(2.4) becomes intractable. When (2.2) is removed, the remaining problem, including the easy constraints (2.3), is assumed to be relatively easy to solve compared to the original problem. To attempt to solve (2.1)–(2.4) by LR, a vector of non-negative variables $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_m)$, termed Lagrangian multipliers, is introduced into the objective function (2.1). This creates the following relaxed problem in which the difficult constraints (2.2) have been relocated and weighted with a set of fixed values from $\boldsymbol{\mu}$:

$$\begin{aligned} \text{Minimize } Z_D(\boldsymbol{\mu}) &= \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{\mu}^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}), \\ \text{subject to} & \\ \boldsymbol{D}\boldsymbol{x} &\leq \boldsymbol{e}, \qquad (2.6) \\ \boldsymbol{x} &\in \mathbb{Z}_+^{\boldsymbol{n}}. \qquad (2.7) \end{aligned} \tag{2.5}$$

Since (2.6) is a set of easy constraints, it is assumed that there exists an algorithm that can be used to efficiently solve the relaxed problem (2.5)–(2.7) in polynomial or pseudo-polynomial time. For any set of given non-negative values $\boldsymbol{\mu}$, it is straightforward to show that $Z_D(\boldsymbol{\mu}) \leq Z$, i.e. the value of the solution to (2.5)–(2.7) is a lower bound on the value of the solution to (2.1)–(2.4). Frequently, $Z_D(\boldsymbol{\mu})$ is a tighter lower bound on Z than that provided by solving the linear relaxation of the model (2.1)–(2.4). The widespread interest in LR stems from the fact that in some cases, $Z_D(\boldsymbol{\mu}) = Z$, i.e. the optimal solution to (2.5)–(2.7) is actually the optimal solution to (2.1)–(2.4).

Studying (2.5)–(2.7) raises the obvious question: What are the best possible values for the entries of $\boldsymbol{\mu}$, i.e. those that provide the tightest possible bound on Z (for which either $Z_D(\boldsymbol{\mu}) = Z$ or, at least, $Z_D(\boldsymbol{\mu})$ is quite close below Z)? This question can be answered by solving the following problem, termed the Lagrangian dual:

$$\begin{aligned} Z_D &= \max_{\boldsymbol{\mu} \geqslant \boldsymbol{0}} Z_D(\boldsymbol{\mu}) \\ &= \max_{\boldsymbol{\mu} \geqslant \boldsymbol{0}} \min_{p} \boldsymbol{c}^T\boldsymbol{x}^{\boldsymbol{p}} + \boldsymbol{\mu}^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{\boldsymbol{p}}), \qquad (2.8) \\ &= \max_{\boldsymbol{\mu} \geqslant \boldsymbol{0}} \boldsymbol{\mu}^T\boldsymbol{b} + \min_{p} (\boldsymbol{c}^T - \boldsymbol{\mu}^T\boldsymbol{A})\boldsymbol{x}^{\boldsymbol{p}}, \qquad (2.9) \end{aligned}$$

where $\{\boldsymbol{x}^{\boldsymbol{p}}, p = 1, \ldots, P\}$ is the set of feasible solutions to (2.5)–(2.7), assumed to be of finite cardinality P.

Due to the integrality requirement (2.4), problems (2.1)–(2.4) and (2.8) are not, in general, equivalent, and thus $Z_D(\boldsymbol{\mu}) \leq Z$ (with equality not necessarily holding). Let $\boldsymbol{\mu}^*$ be the optimal multipliers for (2.5)–(2.7). As can be seen from (2.5), $Z_D(\boldsymbol{\mu})$ is the lower envelope of a finite family of linear functions, but $Z_D(\boldsymbol{\mu})$ is not, in general, differentiable at any $\boldsymbol{\mu}^*$. This causes major difficulties in solving (2.8), making the steepest ascent gradient solution method invalid. Instead, the so-called subgradient optimization method (Anstreicher and Wolsey 2009) is commonly used, where the gradients are replaced by subgradients of the form $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{\boldsymbol{p}}$, $p = 1, \ldots, P$.

2.4.1. Solving the Lagrangian dual

Geoffrion (1974) showed that $Z_{LP} = Z_D(\boldsymbol{\mu}^*)$ and that $Z_D(\boldsymbol{\mu}*) \geqslant Z^*$, $\boldsymbol{\mu}^* \geqslant \boldsymbol{0}$, where $\boldsymbol{\mu}^*$ is the multiplier vector regarding the optimal solution of (2.5)–(2.7), and Z^* and Z_{LP} are

the values of the optimal solutions of the problem (2.1)–(2.4) and its linear relaxation, respectively. Geoffrion also showed that a vector $\boldsymbol{x}$ is an optimal solution to (2.1)–(2.4) if, given a certain set $\boldsymbol{\mu}$ of multipliers, it satisfies the following conditions:

i. $\boldsymbol{x}$ is optimal in (2.9);
ii. $\boldsymbol{Ax} = \boldsymbol{b}$; and
iii. $\boldsymbol{\mu}(\boldsymbol{b} - \boldsymbol{Ax}) = 0$.

Defining a vector $\Delta = \boldsymbol{c^T} - \boldsymbol{\mu^T A}$, then an optimal solution to (2.9) is obtained by fixing:

$$x_j = \begin{cases} 1, & \text{if } \Delta_j < 0; \\ 0 \text{ or } 1, & \text{if } \Delta_j = 0; \\ 0, & \text{if } \Delta_j > 0. \end{cases} \tag{2.10}$$

One of the available methods to solve (2.9) is described in the next section.

2.4.2. Subgradient optimization method

The Subgradient Optimization Method, proposed by Held and Karp (1971), begins with a vector of multipliers $\boldsymbol{\mu^0}$ and then, iteratively, calculates directions and steps to obtain a sequence of vectors $\boldsymbol{\mu^k}$, which converges to the vector $\boldsymbol{\mu}^*$ that maximizes (2.9). Each vector $\boldsymbol{\mu^k}$ is closest to the optimal vector $\boldsymbol{\mu}^*$ (in terms of the norm $||\boldsymbol{\mu^k} - \boldsymbol{\mu}^*||$) than its predecessor $\boldsymbol{\mu^{k-1}}$, despite the fact that the objective function does not increase monotonically. The procedure below summarizes the method in pseudocode.

SUBGRADIENT-OPT()

1 Determine a vector $\boldsymbol{\mu^0}$ of multipliers
2 **for** $k = 0$ **to** t
3 Solve (2.9) with the vector $\boldsymbol{\mu^k}$
4 Calculate the subgradients $\boldsymbol{\sigma_k} = \boldsymbol{b} - \boldsymbol{Ax^p}$
5 Calculate the step $t_k = \frac{\lambda_k(Z^* - Z_D(\boldsymbol{\mu^k}))}{||\boldsymbol{\sigma_k}||}$
6 Do $\boldsymbol{\mu^{k+1}} = \max(\boldsymbol{0}, \boldsymbol{\mu^k} + \boldsymbol{t_k\sigma_k})$
7 **return** $\boldsymbol{\mu^t}$

The natural choice for the initial vector multiplier is $\boldsymbol{\mu^0} = \boldsymbol{0}$. However, the convergence can be accelerated by making $\boldsymbol{\mu^0} = \boldsymbol{u}$, where $\boldsymbol{u}$ is a solution to the dual of the linear relaxation problem (2.1)–(2.4). In this method, there is no way to prove that the optimal solution is reached unless it is obtained via a vector of multipliers $\boldsymbol{\mu^k}$ such that $Z_D(\boldsymbol{\mu^k}) = Z^*$. Thus, the stopping criteria, in general, is a limited number of iterations (t, line 2). The justification for how the step t_k is performed and an explanation about the factor λ_k can be found in Held and Karp (1971), Held *et al.* (1974), and Goffin (1977). Held *et al.* (1974) validated the commonly-used step size for subgradient optimization (t_k, line 5), where the denominator is the square of the norm of the subgradient vector $\boldsymbol{b} - \boldsymbol{Ax^p}$.

3. MSJQ models and algorithms

3.1. The SM model

In this section, we describe a proposed model for scheduling multiway spatial join queries in distributed systems, called SM (Spatial Multiway), together with solution algorithms for it. To conform with the related literature, we use the term *machine* as a synonym of *a processor* in a distributed system.

We assume that multiway spatial join queries are processed in a pairwise fashion, two datasets at a time. We also assume that each dataset has been previously partitioned and that the resulting data partitions have been distributed to the machines. However, no prior knowledge of the queries is assumed (i.e. during data distribution we do not know the set of queries that will be performed over the data).

A multiway spatial join can be thought of as a set of steps, each of them composed by a set of jobs $\boldsymbol{J}$, $n = |\boldsymbol{J}|$, with each job defined by a pair of data partitions that are aligned by a spatial predicate between two spatial datasets. The pair of partitions that compose a job needs to be processed in the same physical machine, as the spatial objects that constitute them need to be evaluated against the spatial predicate algorithm specified in the query step. Each job must be processed on exactly one of a given number m of nonidentical and unrelated machines, running in parallel.

The model includes parameters that represent the datasets and their allocation in the distributed system. The communication cost of processing a job j on a machine i, c_{ij}, is defined as the data transfer cost that is incurred when moving a data partition from the machine where it is currently located to the machine where it is assigned for processing. No communication cost incurs if the data partition is processed on a machine to which it has been previously assigned, whether it is the original data partition or a replica.

The processing cost w is defined as the 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 u for a machine, arising from a prior unbalanced query execution, or any other particularity of the system. This residual load is useful when scheduling multiway queries, as it accounts for the imbalance of a prior step, and hence may induce a better balance for the entire query.

The c_{ij} and w parameters are estimated by a cost model for spatial query processing. These metadata are gathered when datasets are loaded and spread in the underlying distributed data system. We focus here on the problem of scheduling the multiway query and assume that these values are previously computed. For a comprehensive set of data structures, methods, and estimation formulas that can provide these parameters, i.e. a cost model for multiway spatial join queries, please see de Oliveira (2017).

The objective of the scheduling is to perform job allocation in such a way that the query load is somewhat evenly distributed among the machines, i.e. reducing the makespan, but acknowledging that the communication cost incurred must also be controlled. However, these are two conflicting objectives in the sense that to achieve a better balance in query execution we may incur unacceptably high costs in transferring data partitions to idle machines. To this end, we introduce a parameter f, to

Table 1. Symbols used in the SM Model and algorithms.

Symbol	Description
J	The set of jobs
Indices	
j	The job in set J, $1 \leq j \leq n$
i	The machine, $1 \leq i \leq m$
Parameters (all nonnegative and finite)	
n	The number of jobs, $\|J\|$
m	The number of machines, $(m \leq n)$
w_j	The processing cost of j on any machine
c_{ij}	The cost of communication incurred when j is processed in machine i
u_i	The residual load of i in a previous query or multiway step
f	A factor that converts the communication cost into a processing cost, enabling the two expressions in (3.1) to be measured in a common unit of cost.
Decision variables	
x_0	The makespan for the query step
x_{ij}	$\begin{cases} 1, & \text{if job } j \text{ is processed on machine } i, \\ 0, & \text{otherwise.} \end{cases}$

specify the desired emphasis on a balanced schedule or on a low usage of network capacity. Table 1 summarizes the indices, parameters, and decision variables used in the SM Model and in the algorithms presented in the following sections.

Then we have the following problem:

$$\text{Minimize } Z_{SM} = fx_0 + \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij}, \tag{3.1}$$

$$\text{subject to}$$

$$\sum_{i=1}^{m} x_{ij} = 1, j = 1, \ldots, n; \tag{3.2}$$

$$\sum_{j=1}^{n} w_j x_{ij} + u_i \leq x_0, i = 1, \ldots, m; \tag{3.3}$$

$$x_{ij} \in \{0, 1\}, j = 1, \ldots, n; i = 1, \ldots, m. \tag{3.4}$$

Function (3.1) represents the weighted objective of minimizing the makespan and the sum of the communication costs. Constraint family (3.2) expresses the requirement that each job must be processed on exactly one machine. Constraint family (3.3) is a set of logical inequalities arising from the need to minimize the makespan. Constraint family (3.4) represents the usual integrality constraints, indicating if a job is or is not processed by a particular machine (x_{ij}).

Clearly, computational performance is sensitive to the relative value of f. If f is set to zero, the makespan is of no importance and the problem reduces to one of only processing cost minimization. If f is set to a relatively high value, the total processing cost is of little importance and the problem reduces to one of makespan minimization. Either of these reduced problems are easier to solve than the case where we have an intermediate value of f. The weighting factor f can be adjusted so that for an optimal solution to any numerical instance, the total communication cost is approximately equal to the makespan. That is, the total communication cost and the weighted

makespan are of roughly the same importance. From now on we assume the latter, more challenging case.

Although determining a suitable value for f is nontrivial and instance-dependent, it is possible to specify a value that establishes a compromise between makespan and communication through a parametric analysis. Such analysis retrieves the nature of solutions and their values as a function of f. Due to the focus on scheduling methods and space limitations, we present the parametric analysis in the Supplementary Material (de Oliveira *et al.* 2023). Also, we direct the reader to de Oliveira (2017, Chapter 5) for a complete explanation with examples. Although f is continuous, we have shown that it is possible to establish bounds and breakpoints to it, i.e. there is a small set of values for f that changes the query schedule when solving SM.

3.2. Observations on the SM model

The SM model represents a problem that is an extension of the machine scheduling problem $R|pmtn|C_{\max}$, i.e. minimize the makespan with a number of unrelated machines running in parallel. $R|pmtn|C_{\max}$ is $\mathcal{NP}$-complete (by reduction from 3-PARTITION). Hence the problem related to SM is also $\mathcal{NP}$-complete, so the likelihood of the existence of a pseudo-polynomial exact algorithm for it is remote.

SM focuses on which processor a particular task is to be processed on. For instances with many given tasks having large processing times, the memory requirements and the solution times will be high. This makes it extremely hard to solve practical instances of the model exactly by standard integer programming methods, such as branch-and-bound or branch-and-cut. This has been confirmed by computational experiments with a branch-and-cut algorithm for the problem of minimizing the makespan on parallel processors (cf Martello *et al.* 1997). An analysis of the distribution of the total computation time over the various components of the branch-and-cut algorithm by Martello *et al.* (1997) revealed that most of the time was spent solving linear programs. This result reinforces the quest for efficient approximate solution methods for SM.

3.3. Solution methods

3.3.1. Linear programming relaxation of SM

Linear relaxation of SM consists of removing the integrality constraints (3.4), thus letting x_{ij} assume fractional values in the solution. Despite removing (3.4), the constraint family (3.2) imposes an upper limit such that each job is still scheduled once $\left(\sum_{i=1}^{m} x_{ij} \leq 1\right)$.

The optimal solution for the linear relaxation can be computed by a Linear Programming method, such as the well-known Simplex algorithm (Bazaraa *et al.* 2009). 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 the schedule. We denote this procedure by LP in the following.

Both LP and the method resulting from the Lagrangian relaxation use the same repairing procedure (Repair-Partial-Solution), which we introduce in Section 3.3.3.

3.3.2. Lagrangian relaxation

Following Section 2.4, one possible Lagrangian dual for SM is obtained by dualizing constraints (3.2) into the objective function (3.1) using the Lagrangian multipliers $\boldsymbol{\mu} = (\mu_j | j \in \boldsymbol{J})$. The resulting dual model is:

$$\begin{aligned} Z_D &= \max_{\boldsymbol{\mu} \geqslant \mathbf{0}} Z_D(\boldsymbol{\mu}), \\ &\text{subject to } (3.3) \text{ and } (3.4), \end{aligned} \tag{3.5}$$

where

$$\begin{aligned} Z_D(\boldsymbol{\mu}) &= \min_{x} fx_0 + \sum_{j=1}^{n}\sum_{i=1}^{m} c_{ij}x_{ij} + \sum_{j=1}^{n} \mu_j \left(\sum_{i=1}^{m} x_{ij} - 1 \right) \\ &= \min_{x} fx_0 + \sum_{j=1}^{n}\sum_{i=1}^{m} (c_{ij} + \mu_j)x_{ij} - \sum_{j=1}^{n} \mu_j. \end{aligned} \tag{3.6}$$

Problem (3.5) reduces to m knapsack problems, one for each constraint in (3.3). A knapsack problem (Kellerer *et al.* 2004) consists in 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 a given knapsack capacity v. The problem can be solved in pseudo-polynomial time (Kellerer *et al.* 2004). A knapsack model is given in (K.1)–(K.3) for reference. (K.1) is the objective function, (K.2) ensures that the total weight of the selected items does not exceed v, and (K.3) defines the integrality constraints for x_j to indicate the selected items:

$$Z_K = \max \sum_{j=1}^{\hat{n}} p_j x_j, \tag{K.1}$$

subject to

$$\sum_{j=1}^{\hat{n}} \hat{w}_j x_j \leq v, \tag{K.2}$$

$$x_j \in \{0, 1\}, j = 1, \ldots, \hat{n}. \tag{K.3}$$

The m knapsack problems for LR are obtained in the following way: Let $K_{LR}(i, \boldsymbol{\mu})$, $i = 1, \ldots, m$ be the i-knapsack problem for LR. The number of items to select from is $\hat{n} = n$, one item for each $j \in \boldsymbol{J}$. The weights $\hat{w}$ for each problem are obtained from w values, and profits p from c and $\boldsymbol{\mu}$. A lower bound for v can be determined by $\sum_{j \in \boldsymbol{J}} w_j / m \leq x_0$. The complete model for a $K_{LR}(i, \boldsymbol{\mu})$ is defined by (3.7) to (3.9). The inversion of the sign for the sum of profits in (3.7) is due to knapsack being a maximization problem, while SM is a minimization problem.

$$Z_{K_{LR}}(i, \boldsymbol{\mu}) = \max - \sum_{j=1}^{n} (c_{ij} + \mu_j)x_{ij}, \tag{3.7}$$

subject to

$$\sum_{j=1}^{n} w_j x_{ij} \leq v - u_i, \tag{3.8}$$

$$x_{ij} \in \{0, 1\}, j = 1, \ldots, n. \tag{3.9}$$

Next, we present a procedure to compute $\boldsymbol{\mu}$ and to obtain feasible solutions to SM, based on the iterative Subgradient Optimization Method. Algorithm 3.1 shows the steps of the method in pseudocode. After setting initial values in lines 1–3, we compute and set the initial upper bound on Z_{SM} (line 4), by calling a greedy algorithm that is presented next. The value of v (line 5) is set by using a best-fit heuristic that provides an upper bound by ordering the jobs according to decreasing order of makespan and allocating them to the least-used machine. The lower bound for v is set in line 6. The algorithm then iterates from $k = 0$ to a specific number of iterations t and while the conditions for t_k and λ are not met. In lines 9 and 10 the method iteratively solves the m knapsack problems. Next, in line 11, the vector of subgradients $\boldsymbol{\sigma}$ is computed. If the value of $Z_D(\boldsymbol{\mu^k})$ increases above the upper bound limit Z^U (line 12), we reduce the value of v by an arbitrary small percentage (line 13) and return to solve the knapsack problems again. Otherwise, we call the procedure Repair-Partial-Solution to repair the partial solution $\boldsymbol{x}$, transforming it into a feasible solution to SM (line 15). Next, based on a possibly improved upper bound Z^U (line 16), a new t_k and $\boldsymbol{\mu^{k+1}}$ for the next iteration are computed (lines 17 and 18). Line 19 updates λ if a better Z_D is not found in λ^i iterations. The method returns the best feasible solution found ($\hat{\boldsymbol{x}}$). We refer to this procedure as the LR method.

Algorithm 3.1 Procedure to compute a feasible solution to SM through LR-relaxation.

Solve-LR-Relaxation$(\boldsymbol{c}, \boldsymbol{w}, \boldsymbol{u}, f)$

1 $\boldsymbol{\mu^0} = \boldsymbol{0}$
2 $\lambda = 2$
3 $t_0 = 1$
4 Z^U is set with an upper bound on Z_{SM}
5 $v =$ best-fit UB for $\boldsymbol{x_0}$.
6 $v^{lb} = \sum_{j \in J} w_j / m$
7 $k = 0$
8 **while** $(k < t)$ **and not** $\left(t_k < 1 \times 10^{-4} \text{ and } \lambda < 1 \times 10^{-4}\right)$
9 **for** $i = 1$ **to** m
10 Solve $K_{LR}(i, \boldsymbol{\mu}^k)$ and partially set $\boldsymbol{x}$
11 $\sigma_j = \sum_{i=1}^{m} x_{ij} - 1, \forall j \in \boldsymbol{J}$
12 **if** $\left(v > v^{lb} \text{ and } Z_D(\boldsymbol{\mu^k}) > Z^U\right)$
13 reduce v
14 **else**
15 $\hat{\boldsymbol{x}} =$ Repair-Partial-Solution$(\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{c}, \boldsymbol{u}, f)$
16 $Z^U = min\left(Z^U, Z^{\hat{x}}\right)$
17 $t_k = \dfrac{\lambda\left(Z^U - Z_D(\boldsymbol{\mu^k})\right)}{\|\boldsymbol{\sigma}\|^2}$
18 $\boldsymbol{\mu^{k+1}} = \boldsymbol{\mu^k} + \boldsymbol{\sigma} t_k$
19 halve λ if a better Z_D is not found in λ^i iterations
20 $k = k + 1$
21 **return** best $\hat{\boldsymbol{x}}$

3.3.3. Repairing heuristic

This section introduces a heuristic to repair a partial schedule provided by either LP or LR by transforming it into a feasible solution. The partial LP solution can have fractionally assigned jobs, where $0 < x_{ij} < 1$, which we round up, eventually turning them into multiple assigned jobs. The partial LR solution, in turn, does not have fractionally assigned jobs but can present multiple assigned jobs, as well. Let $\boldsymbol{x}$ be the partial solution provided by the LP or LR method, and let us partition the jobs into three sets defined by:

$$S_1 = \{j \in \boldsymbol{J} | \sum_{i=1}^{m} \lceil x_{ij} \rceil = 0\}, \tag{3.10}$$

$$S_2 = \{j \in \boldsymbol{J} | \sum_{i=1}^{m} \lceil x_{ij} \rceil = 1\}, \tag{3.11}$$

$$S_3 = \{j \in \boldsymbol{J} | \sum_{i=1}^{m} \lceil x_{ij} \rceil > 1\}, \tag{3.12}$$

where $\boldsymbol{S_1}$ is the set of unassigned jobs, $\boldsymbol{S_2}$ is the set of jobs that are correctly assigned, and $\boldsymbol{S_3}$ is the set of jobs that were multiply assigned. All $j \in \boldsymbol{S_1} \cup \boldsymbol{S_3}$ need to be repaired to transform $\boldsymbol{x}$ 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 as the difference between the maximum and the minimum cost that may be incurred if the job has been scheduled in the worst or the best possible machine, plus its load w_j weighted by f. Formally, r_j is defined by (3.13). We use this concept to sort the assignment of jobs, in a way that jobs that have a large load (fw_j) or a large regret are scheduled first.

$$r_j = fw_j + (\max_{1 \le i \le m} c_{ij} - \min_{1 \le i \le m} c_{ij}) \tag{3.13}$$

Algorithm 3.2 Procedure to repair a partial solution $\boldsymbol{x}$ to SM.

REPAIR-PARTIAL-SOLUTION$(\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{c}, \boldsymbol{u}, f)$

$\hat{\boldsymbol{x}} = \boldsymbol{0}$
$\boldsymbol{S_1} = \emptyset$
$\boldsymbol{S_3} = \emptyset$
for $i = 1$ **to** m
    $load[i] = u[i]$
for $j \in \boldsymbol{J}$
    $t = \sum_{i=1}^{m} \lceil x_{ij} \rceil$
    if $t == 0$
        $\boldsymbol{S_1} = \boldsymbol{S_1} \cup \{j\}$
    if $t > 1$
        $\boldsymbol{S_3} = \boldsymbol{S_3} \cup \{j\}$
    if $t == 1$
        Let i be the machine where j is allocated
        $\hat{x}_{ij} = 1$
        $load[i] = load[i] + w_j$
SCHEDULE-UNASSIGNED-JOBS$(\boldsymbol{load}, \boldsymbol{S_1} \cup \boldsymbol{S_3}, \hat{\boldsymbol{x}}, \boldsymbol{w}, \boldsymbol{c}, f)$
IMPROVE-REPAIRED-SOLUTION$(\boldsymbol{load}, \hat{\boldsymbol{x}}, \boldsymbol{w}, \boldsymbol{c}, f)$
return $\hat{x}$

We repair $\boldsymbol{x}$ using the procedure Repair-Partial-Solution in Algorithm 3.2. Let $\hat{\boldsymbol{x}}$ be the feasible solution under construction. The procedure starts by setting the residual load of previous steps u (lines 4 and 5). Next, it computes the number of machines for which each $j \in \boldsymbol{J}$ is allocated (t) and uses it to build the sets $\boldsymbol{S_1}$ and $\boldsymbol{S_3}$ (lines 7 to 11). If j is correctly set ($t = 1$), $\hat{\boldsymbol{x}}$ is set accordingly (line 14), and its load w_j is added to the array of loads for each machine i (line 15). The remaining jobs $j \in \boldsymbol{S_1} \cup \boldsymbol{S_3}$, denoted as $\boldsymbol{S_u}$, are assigned to machines by the procedure Schedule-Unassigned-Jobs (line 16). After this call, $\hat{\boldsymbol{x}}$ has a feasible solution for SM. This solution is further improved by procedure Improve-Repaired-Solution (line 17). Next, we describe these two auxiliary procedures.

Procedure Schedule-Unassigned-Jobs, in Algorithm 3.3, starts by sorting the jobs in $\boldsymbol{S_u}$ in decreasing order of r_j (line 1). Next, for each item $j \in \boldsymbol{S_u}$, the procedure finds the machine s for which the assignment of j least increases the cost (lines 2 to 12), assigns j to it (line 13), and updates the load on machine s (line 14) for the next iteration. The procedure is terminated when all jobs are assigned to machines. For the sake of simplicity, we 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 14 if $load[s]$ exceeds the stored x_0 value.

Algorithm 3.3 Procedure to schedule unassigned jobs in S_u.

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

After the scheduling of the jobs in $\boldsymbol{S_u}$, we then 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 3.4, searches for a new machine s, in which to schedule j, $j \in \boldsymbol{J}$, such that the sum of the processing and communication costs are reduced by the most (lines 1 to 12). If there exists such machine s that

would cause a positive cost reduction (line 13), j is moved from p_j to s (lines 14 and 15) and the loads 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.

3.3.4. 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 identify a complete schedule, starting with no scheduled jobs in $\hat{\boldsymbol{x}}$, $\boldsymbol{S_u} = \boldsymbol{J}$, and an empty ***load*** array.

There are three purposes in using it in this way: (i) to compare the performance of the combinatorial methods (LP and LR) with the performance of an intuitively appealing but simple method; (ii) to use it when the limit of time imposed on the query optimization is critical, for example, for queries with small run times; and (iii) to use it as a baseline for comparison, observing that there is no other established method to compare against, and checking that it is similar to the greedy algorithm used in related work. We refer to this way of using these procedures as the GR method since it constitutes a greedy heuristic to provide solutions to SM.

Algorithm 3.4 Procedure to improve the feasible solution $\hat{x}$.

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

```
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. Evaluation data and results

We chose a set of public spatial datasets, obtained from the Brazilian Institute of Geography and Statistics[2] (IBGE), from the LAPIG Laboratory[3] of the Institute of Social and Environmental Studies (IESA) at UFG, from the Digital Chart of the World[4] (DCW),

Table 2. Datasets used in experiments.

Name	Abrev.	Type	Cardinality	SHP/GDB 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	5564	38.8
Vegetation	V	Polygons	2140	4.7
World-wide datasets (DCW)				
Rivers	RI	Lines	943,638	243.2
Railways	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
Tiger datasets for gigabyte-size experiments				
Primary roads	PR	Lines	13,373	47 (.csv 77 MB)[a]
Area land mark	LM	Polygons	129,252	132 (.csv 406 MB)
Area water	AW	Polygons	2,292,811	821 (.csv 6.5 GB)
Linear water	LW	Lines	5,825,479	2103 (.csv 18.3 GB)
Edges	ED	Lines	69,572,173	14,558 (.csv 62.0 GB)

[a]It is common to find these datasets converted to .csv format in related work that uses the Hadoop or Spark frameworks. Here, we depict both the original binary size and the .csv size reported in related work for a complete understanding of the experiment volume. As can be seen, the binary size is smaller, but when loaded, these datasets occupy more work memory. The inverse can be the case for the .csv format, depending on the implementation details of each system.

and from the TIGER 2015 spatial database (Bureau 2015). Table 2 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 (SHP) or FileGDB binary formats, and used the GDAL[5] and GEOS[6] libraries to extract and process the geometry of each spatial object contained in them.

The first and second groups of datasets, in Table 2, were used to perform the set of experiments involving the scheduling methods due to their manageable sizes. We observe that what determines the execution time of the optimization is not the size of datasets, but the number of jobs and machines, as discussed in Section 3.2. The number of jobs, n, is determined by the number of data partitions. Larger datasets should have larger data partitions as this improves query execution time by reducing the number of control messages in the system, resulting in a similar number of jobs. The Gigabyte-size experiments, in Section 4.3, combine the third group of datasets to form multiway queries and measure their run time.

To evaluate our proposed methods, we employed the datasets listed in Table 2 to build a set of pairwise and multiway spatial join queries. Each pairwise spatial join provides one scheduling instance, i.e. a set of jobs $\boldsymbol{J}$ with their corresponding costs (w_j and c_{ij}). Each multiway spatial join provides the same number of scheduling instances as its number of steps. Additionally, in multiway instances, the imbalance from a prior step is represented by the parameter $\boldsymbol{u}$.

Table 3 displays the pairwise join queries used in the experiments. There are 20 queries, constructed using the datasets listed in Table 2 and providing an all-to-all combination of the first group of datasets and also an all-to-all combination of the second group of datasets. This set comprises queries that illustrate all possible combinations of the two types of spatial objects, i.e. line $\bowtie$ line, line $\bowtie$ polygon, and

Table 3. Pairwise spatial join queries used in experiments.

Name	Query	Jobs	Name	Query	Jobs
J_1	A ⋈ H	8082	J_{11}	RI ⋈ RA	5572
J_2	A ⋈ R	8082	J_{12}	RI ⋈ HI	10,298
J_3	A ⋈ C	8082	J_{13}	RI ⋈ EC	10,019
J_4	A ⋈ V	8082	J_{14}	RI ⋈ CR	6630
J_5	H ⋈ R	7125	J_{15}	RA ⋈ HI	4614
J_6	H ⋈ C	7587	J_{16}	RA ⋈ EC	4588
J_7	H ⋈ V	7755	J_{17}	RA ⋈ CR	4209
J_8	R ⋈ C	2139	J_{18}	HI ⋈ EC	8495
J_9	R ⋈ V	2160	J_{19}	HI ⋈ CR	5624
J_{10}	C ⋈ V	114	J_{20}	EC ⋈ CR	5106

Table 4. Multiway instances with intermediate results and the number of jobs n of each step.

		Jobs n in each step		
Query		$M_{i.1}$	$M_{i.2}$	$M_{i.3}$
M_1	$((A \bowtie RI) \bowtie RA) \bowtie CR$	148	69	69
M_2	$(RI \bowtie RA) \bowtie EC$	5572	5477	–
M_3	$(RI \bowtie HI) \bowtie RA$	10,298	5544	–
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	9870	5450

polygon ⋈ polygon, as well as distinct cardinality results. The join predicate used is *intersect*. In what follows, we refer to these queries using their numbers, ranging from J_1 to J_{20}.

Table 4 displays the set of multiway spatial join queries for the scheduling experiments. We refer to the steps of a multiway query as $M_{i.j}$, where i is the query number and j is the step. For example, M_1 has three steps referred to as $M_{1.1}$, $M_{1.2}$, and $M_{1.3}$. Tables 3 and 4 also present the number of jobs n for each query (or query step). The resulting 36 experiment instances, i.e. 20 spatial join queries and 16 steps of multiway spatial join queries, were scheduled on $m = (4, 8, 16, 32, 64)$ machines, that is, 180 schedules were tested for each method.

Before query execution, we distributed the data partitions to the machines, individually for each dataset, using a round-robin algorithm. In the resulting data distribution, often a data partition from two distinct datasets with overlapping geographic regions happened to be assigned to distinct machines. Our purpose in using this distribution is to force the scheduling algorithm to find a way to reduce the communication cost by strategically assigning the jobs to machines. We determined the parameter f for each query step following the methods in the Supplementary Material (de Oliveira *et al.* 2023), such that the total communication cost is approximately equal to the makespan, i.e. the total communication cost and the weighted makespan are of roughly the same importance.

The necessary parameters were set as $\lambda^i = 50$ and $t = 3000$, when executing the procedure Solve-LR-Relaxation (Algorithm 3.1). The maximum number of iterations t was reached for 11 experiment instances. For the others, the number of iterations remained 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,[7] with optimization flags `-Ofast -march=native`. Pisinger's minknap algorithm[8] (Pisinger 1997) was used to solve the m knapsack problems 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,[9] version 12.6.1. The model and its parameters are set in the CPLEX optimization module through C API calls, and the extreme point solution is captured after the optimization process terminates.

The experiments for Section 4.1 were performed in m4.4xlarge Amazon EC2 virtual machines, with Intel(R) Xeon(R) CPUs, E5-2686 v4 model, running at 2.30 GHz with 64GB of RAM. The operating system used was the standard Debian 11 offered by the provider through an AMI image. Experiments that display execution time, in Section 4.2, were performed in a controlled local environment, using an AMD Phenom™ II X6 1055 T 2.8 GHz processor with 8 G of RAM, a Debian 8.7 distribution, and Linux Kernel version 3.18.1. The environment for the distributed Gigabyte-size query experiments will be described in Section 4.3.

4.1. Quality of generated schedules

In this section, we compare each schedule provided by the GR, LP, and LR methods 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 an ideal schedule for each instance of SM. To calculate Z^{lb}_{SM}, we processed SM via the CPLEX software and left its MIP (Mixed Integer Programming) solver to run from the root node, applying all possible cuts. We present the distance between the proposed schedule and the lower bound, computed as $gap = (Z^{+}_{SM} - Z^{lb}_{SM})/Z^{lb}_{SM}$, where the $+$ sign indicates the method used in each case, e.g. Z^{GR}_{SM}.

Figure 1 presents the results. There are five charts, one for each cluster size (i.e. 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 (and sometimes equal) to the optimum. The gaps for GR are the largest, almost all fitting in the range of 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 ($\sigma = 31.71\%$). LP improved over the GR schedule in almost all instances. Example instances for which it performed worse than GR are $M_{1.1-1.3}$ for $m = 16$, and $M_{4.1-4.3}$ for $m = 64$. The main implication for LP is that the schedules were worse than GR 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, i.e. jobs that were partially scheduled in more than one machine, increased with the number of machines. We present some statistical values in Table 5 where it is possible to check this behavior. The average of all gaps for LP is 6.38% ($\sigma = 10.74\%$).

LR achieved the smallest gaps. Observing Figure 1, for $m = 4$, there are 21 out of 36 instances with gap $\leq 0.01\%$, with the other 15 instances having gap $\leq 1\%$ (check the dotted line at gap $= 1$). The gaps increased when m increased, but not

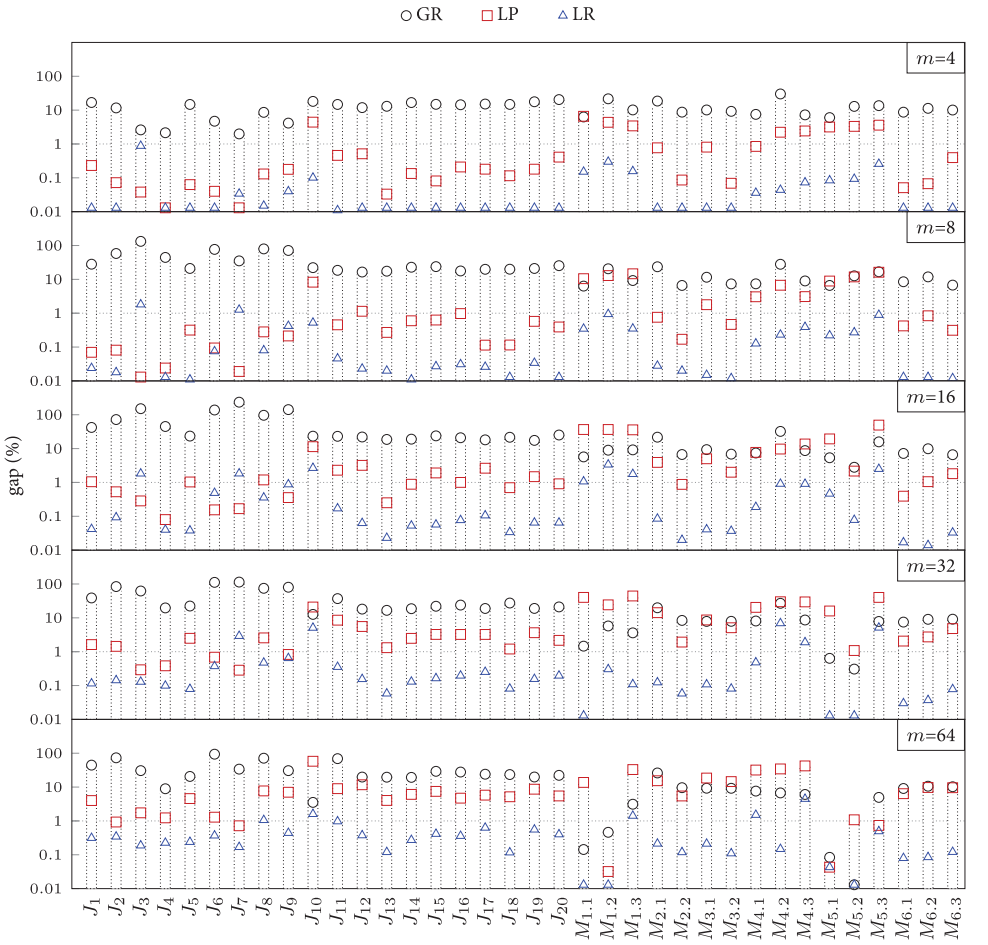

Figure 1. Gap between each schedule provided by GR, LP, and LR and a known lower bound of Z^*_{SM}, for $m \in \{4, 8, 16, 32, 64\}$. The y axis is logarithmic to emphasize the near-optimal schedules.

Table 5. Average and standard deviation for gaps in Figure 1.

	Average for m					Standard deviation for m				
Method	4	8	16	32	64	4	8	16	32	64
GR	11.96	26.8	37.3	27.08	22.18	5.92	26.11	50.97	29.6	22.37
LP	1.1	2.98	7.16	9.75	10.88	1.65	4.7	12.36	12.63	12.97
LR	0.06	0.23	0.57	0.75	0.51	0.15	0.4	0.87	1.61	0.79

significantly (see LR line in Table 5). For $m = 8$, 34 instances still presented gaps $\leq 1\%$ and two had a gap $> 1\%$ (≤ 1.8). For $m = 64$, 31 instances achieved a gap $\leq 1\%$ and the other five instances were such that $1 \leq \text{gap} \leq 4.5\%$. Of all the 180 runs, there are only 10 cases for which LR generated 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_7 for $m = 32$). LR achieved the best gap in all instances when $m = 64$ and also, the best gap for all M instances. The average for all gaps was 0.43% ($\sigma = 0.94\%$) with a worst case of 4.5%.

4.2. Comparison of the execution time to produce a schedule

This section presents the execution time for GR, LP, and LR, and reports the computational experience in solving the practical numerical instances previously mentioned. In the experiments involving the LR method, the 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, were 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, job enumeration, and other cleanup routines, such as memory release.

As expected, we find that solving the MIP numerical instances to optimality using CPLEX was very time-consuming compared to the execution times reported in the following. Moreover, the optimal solution did not show a significant benefit over LR as previously reported—the gaps are $<1\%$ in all instances. For reference, the larger instances, with respect to n and m, took from hours to days to achieve a gap $<10^{-4}$ from Z^*_{SM}. Thus, we decided not to include these timings in the charts.

Figure 2 presents the results. The y axis shows the execution time in seconds using a logarithmic 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. As expected, GR is the fastest, followed by LP and lastly by LR. GR has an average time per instance of 0.002 s for $m = 4$, and 0.007 s for $m = 64$. LP has an average time of 0.052 s for $m = 4$, and 2.7 s for $m = 64$. LR, in turn, has an average of 1.1 s for $m = 4$, and 14.4 s for $m = 64$. The maximum execution time for $m = 64$ for GR, LP, and LR are 0.017, 14.9, and 43.1 s, respectively. The small dot for each bar in (*b*) indicates the standard deviation and it shows that the LP execution time is less stable than for GR and LR (note the dot above the average for LP when $m \geqslant 8$, considering the logarithmic scale).

An important question is what determines the execution time of GR, LP, and LR. From a theoretical perspective, we can examine the complexity of the algorithms proposed. In general, they depend upon n and m. GR complexity is $\Theta(nm)$ for sufficiently large m, or $\Theta(n\ lg\ n)$ otherwise. LP complexity is also determined by n and m, as the underlying algorithm used to optimize the relaxed model (Simplex) is polynomial in

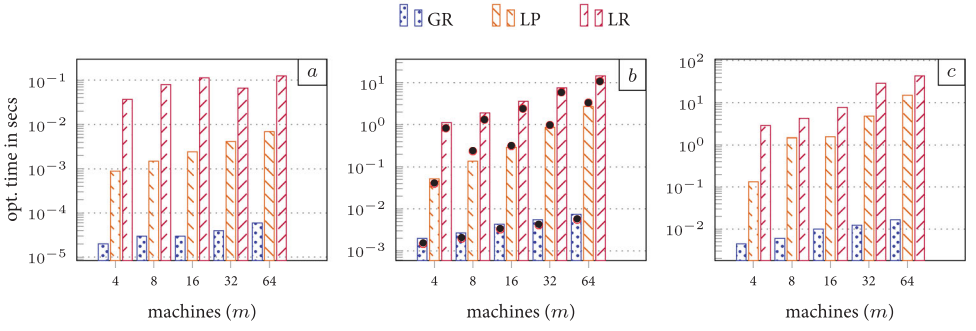

Figure 2. Execution time for GR, LP, and LR. (a) shows the minimum execution time, (b) the average, and (c) the maximum execution time for all J and M queries.

the number of constraints and variables (Hall 2010). Note that n and m determine the number of variables and constraints in the SM model. In turn, LR employs a pseudo-polynomial 0-1 knapsack algorithm whose complexity is $O(vn)$ and it is executed m times in each iteration for a constant number of iterations. Its amortized complexity is $O(vmn)$. The instance parameters w_j are used to compute the capacity v of the knapsacks. From a practical perspective, the number of jobs ranged from $69 \leq n \leq 10,298$ in this experiment. Indeed, queries with smaller n had lower optimization time and were prevalent to determine the minimums in Figure 2(a). Analogously, queries with larger n establish the maximums in Figure 2(c). The impact of m is also perceptible for all methods as the bars increase in size with increasing m. Finally, for a multiway query, the number of steps also determines the execution time as we have as many schedules to optimize as the number of steps.

Observing the execution times reported for this experiment and taking into consideration the quality of schedules from the previous section, we now discuss how to choose the optimization method for a query. In general, the time required to optimize a query is supposed to be smaller than the time required to execute it with a bad but fast-generated schedule. Observing this, the actual optimization method to be used may be chosen based on the expected execution time of the query, which can be estimated based on its estimated processing costs. For small *ad-hoc* queries, i.e. queries expected to have shorter execution time and meant to be executed just once, the best option is usually GR. For larger queries, however, the execution time can accommodate longer optimizations, and even benefit from the improved schedules provided by LR.

If we focus on system throughput, though, LR appears to be the most promising method, as it is the one that reduces the most the makespan of queries, as well as their communication cost. Additionally, a strategy to amortize its footprint may be worthwhile. One option is to cache and reuse the execution plan after the query optimization process, a common strategy used for repetitive or stored queries in non-spatial DBMSs (Graefe 1993). Other options to reduce the optimization time exist, as 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 depends upon the problem structure (Hall 2010).

4.3. Execution time comparison with gigabyte-size datasets, MapReduce, and spark

To investigate the effectiveness of the schedules provided by our proposed methods, called DGEO for short, we compared the execution time, shuffled data, and peak execution memory for three Gigabyte-size multiway spatial join queries executed using DGEO and Apache Sedona (Yu *et al.* 2018), formerly termed GeoSpark. Apache Sedona is an open-source project currently incubated at Apache Foundation and is a representative work on top of Spark. Although it is not able to execute partitioned MSJQ, we pipelined the results of pairwise joins and measured the resources spent with intermediate steps. We also consider the execution time reported by Du *et al.* (2017) for

MSJS and Hadoop ϵCR (Gupta and Chawda 2014). MSJS fully implements MSJQ without collecting and redistributing results for intermediate join steps. However, as it is closed-source software, we cannot reproduce their experiments in our environment. Sphinx (Eldawy *et al.* 2017) presented results only for pairwise spatial joins using two synthetic datasets with rectangles (not real spatial objects), and thus, it is not directly comparable to our realistic scenarios. Tsitsigkos *et al.* (2019) also simplified data to rectangles and is not considered. Finally, due to the differences in methodology, such as the multiway queries used, we only include the results presented by Nidzwetzki and Güting (2017) (Distributed Secondo) in perspective at the end.

To make this comparison possible, we implemented a small distributed query engine for query plan selection and execution, which can execute chain multiway spatial join queries following the selected plan and the respective schedule provided by the optimizer. We used two programming languages: C and Go.[10] The C language was used to code low-level algorithms that interface with the GDAL library, used to load the dataset files in the ESRI Shapefile (.shp) and ESRI FileGDB (.gdb). The code also interfaces with the GEOS library to process spatial predicates. Interoperability between C and Go code was achieved 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. We used queues of jobs that were implemented using the mechanism of *channels* provided by the language. The communication protocols used to provide the interaction between the modules run over TCP sockets, and the serialization of data structures was implemented using the GOB package provided by Go. Further detail about the query engine can be found in de Oliveira (2017).

The three multiway queries used the third group of datasets presented in Table 2, from the TIGER 2015 spatial database. The queries and their sizes are detailed in Table 6, both in binary size (SHP or GDB format) and the .csv size reported in related work. All queries process more than 2 million records. The last query, M_9, processes more than 77 million records and returns a set of more than 17.7 million results.

Our environment was composed of four m4.2xlarge Amazon EC2 instances, each with four CPU cores of an Intel(R) Xeon(R) CPU, E5-2686 v4, running at 2.30 GHz, two threads per core totaling eight vCPUs, and 32GB of RAM. According to the Amazon specification, each vCPU is a Hyper-thread of an Intel Xeon core.[11] The machines were allocated in the same data center, interconnected by a virtual network with a capacity of 10 Gbps for single-flow and 20 Gbps for multi-flow traffic in each direction (full duplex). The operating system used was a Debian 11 offered by the provider through an AMI image.

The execution environment used by Du *et al.* (2017) was composed of four Power Edge R720 Servers, each with an Intel Xeon E5-2630 v2 2.60 GHz processor with 32 GB of RAM, and runs a SUSE Linux enterprise server 11 SP2 operating system. According to the Intel documentation, the Xeon E5-2630 has six cores and 12 hyper-threads. The

Table 6. Gigabyte-size multiway spatial join queries used in experiments.

Name	Query	Binary size (GB)	.csv Size (GB)
M_7	PR $\bowtie$ LM $\bowtie$ AW	1.0	7.0
M_8	PR $\bowtie$ AW $\bowtie$ ED	15.4	68.6
M_9	AW $\bowtie$ LW $\bowtie$ ED	17.5	86.8

network capacity was not mentioned in their experiment. Version 2.6.0 of Apache Hadoop[12] and version 2.0.1 of Spark were used, both running on JDK 1.7. Thus, by comparing the specifications, the hardware used by Du *et al.* (2017) has the same memory size but has more processing power: (i) we used a virtualized environment that incurs the Hypervisor overhead, (ii) our environment CPU clock was smaller (2.3×2.6 GHz), and (iii) the number of hyperthreads per server is smaller (8×12). As a rough estimate, their cluster has ∼63% more computing power without considering Hypervisor overhead.

The chart in Figure 3(a) presents the execution time of each query for each system. M_7 was executed in 6 s by DGEO, 22 s by Sedona (3.7×), and 84 s by MSJS (14×). Similarly, M_8 was executed in 1 min and 32 s by DGEO, 67 min by Sedona (44.7×), and 22 min by MSJS (14.5×). DGEO executed M_9 in 17.3 min, compared with 90 min by Sedona (5×) and 28 min by MSJS (1.6×). The gap between the systems lessened for the largest query, M_9, but the difference in the amount of time remains significant: DGEO finished the execution 11 min earlier than MSJS. The unbalanced execution at the middle-end of query execution partially explains the higher execution time of Sedona: in M_8, the number of tasks (used CPU cores) starts decreasing at 40 min of query execution (12 of 32 possible tasks), and steadily decreases during the remaining 27 min; in M_9, the situation was even worse: only 10 of 32 tasks were running at 45 min—at the middle of query execution. The presence of straggler tasks occurred even when we controlled the number of partitions in the experiment (from 100 to 3000). In contrast, DGEO required only 17 s of unbalanced execution for M_8 and ∼3 min for M_9. We attribute this behaviour to the LR scheduling algorithm, and the small unbalancing at the end occurs due to imprecision in query cost estimates. Additionally, Sedona spent some time collecting and redistributing the intermediate step: 19 s for M_8 and 5 min for M_9 and also shuffled more data during query execution, as described later. Finally, compared to the others, the performance of Hadoop ϵCR can be explained by the use of data persistence in the disk, a design choice of the underlying MapReduce framework.

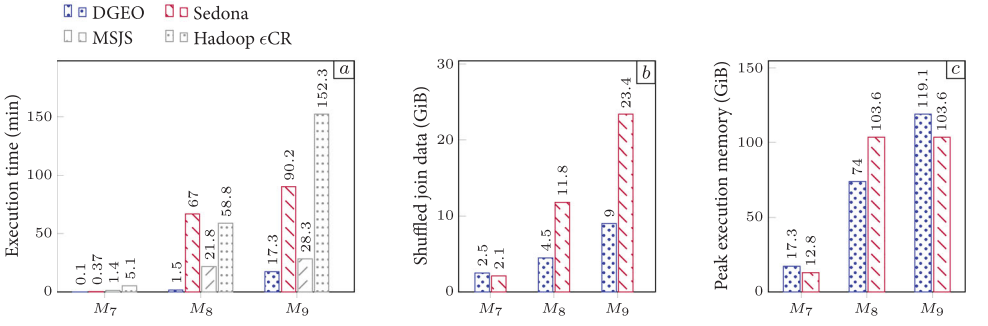

Figure 3. Comparison of the execution time (a), shuffled join data (b), and peak execution memory (c) for M_7, M_8, and M_9. In (a), the gray bars indicate the results reported for a similar but distinct cluster, with the same amount of memory but not virtualized and with ∼63% more processing power (see the textual description in this section and Du *et al.* 2017). We put it into perspective here as it was specifically targeted to MSJQ.

Compared to DGEO, Sedona shuffled more data when running the two largest queries, as depicted in Figure 3(b). The numbers account only for the shuffled data in the join query execution—we did not account for the data loading as the data format used in each system differs (gdb/shp vs csv). Sedona has transferred ∼2.6 times more data through the network for M_8 and M_9. The shuffled data in the additional intermediate step partially justifies the difference for Sedona: 1.7 GiB for M_8 and 6 GiB for M_9. As we controlled the query optimization to maintain an equilibrium between execution time and communication cost in DGEO, through the *f* parameter, we attribute a significant part of the difference to the query optimization. The optimization also justifies the larger data shuffling in M_7: the scheduler identified a way to significantly reduce query execution time at the expense of a small increase in communication. Finally, there are also differences in the underlying technologies used to communicate between nodes: RPC (Netty) and Java Serialization for Sedona and GOB, a heavily optimized binary protocol of the Go Language, used by DGEO. Du *et al.* (2017) have not conducted this experiment.

Last but not least, Figure 3(c) presents the peak execution memory for DGEO and Sedona. The numbers in the chart show the memory used for join query processing, not accounting for supporting structures and other processes in the system. In M_7, peak memory by DGEO is 4.5 GiB larger than by Sedona due to a larger degree of parallelism during the execution in DGEO (100% CPU utilization for 6 s). In M_8, the inverse occurs: DGEO peak is 29.6 GiB smaller than the Sedona peak. In M_9, a particular situation occurred: both systems used almost all available memory in the cluster (128 GiB), but Sedona occupied a larger amount with supporting structures and processes (observe that the maximum is also reached at M_8). Observing that we reserved 2 GiB of memory to operating system processes per node, DGEO was able to use more memory in the query execution due to its lightweight and experimental query engine, and as such, having fewer features and structures.

Although the same queries were not executed in Distributed Secondo (Nidzwetzki and Güting 2017), we nevertheless put into perspective the results given by the authors. Nidzwetzki and Güting (2017) compared the performance of their system with those of SpatialSpark (You *et al.* 2015) and SpatialHadoop (Eldawy and Mokbel 2015). The times reported for a pairwise spatial join involving two datasets from Germany, called Roads and Buildings (size not mentioned in their study), generated from Open Street Maps,[13] are 9, 21, and 23 min, for SpatialSpark, Distributed Secondo, and SpatialHadoop, respectively. Considering this comparison and, due to the fact that Distributed Secondo also stores intermediate results on disk, its performance seems similar to that of MapReduce-based systems.

In summary, the reasons for the better performance achieved by DGEO stems from the focus on query planning, i.e. the proposed query scheduling method (LR). We observe that the execution time is an order of magnitude lower for two of the queries studied (M_7 and M_8). The execution time of DGEO is also significantly lower for M_9, a Gigabyte-size query with millions of results. Indeed, for all queries studied, the difference was sufficiently large to accommodate even the most time-consuming schedules generated by LR.

About query scheduling, we demonstrated in Section 4.1 the difference in makespan and communication costs between a naïve greedy algorithm and more efficient methods based on the theory of combinatorial optimization. The systems compared here use a load balancing mechanism rather than a query scheduling based on metadata. The load balancing is implemented by the underlying framework that is independent of spatial data and is based on the assignment of tasks to idle machines during query execution. This is a kind of greedy strategy, similar to the baseline GR method, that often results in sub-optimal cluster resource usage.

5. Conclusions

In this article, we dealt with the problem of assigning jobs to machines in a locally distributed system. We considered a given set of jobs defined by data partitions of 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 that embraces the minimization of both the makespan and the communication cost as objectives. We discussed the difficulty of solving the problem by exact integer methods and introduced approximate algorithms based on combinatorial methods: the well-known linear relaxation (LP) technique and the more sophisticated Lagrangian relaxation (LR), as well as a baseline greedy algorithm (GR), similar to those used in related work.

Our computational experiments showed that LR usually provided better solutions than either LP or GR. Although LR often requires significant time to identify a solution, it is interesting to observe how close its solutions often are to the optimum and to note the reduction in makespan and communication costs that is achieved. LP and GR are recommended for instances where small, ad-hoc queries are predominant.

We used the best schedules identified by our methods to run Gigabyte-size multiway spatial join queries, reported in the literature, in a realistic experiment that uses practical spatial objects (not simplifications like rectangles). We showed that the scheduling of a query before its execution can reduce the processing time in a distributed environment by an order of magnitude, while also shuffling significantly less data through the network when compared to state-of-the-art frameworks for spatial data processing that do not have this capability.

Although our models focus on multiway spatial join queries, they also apply to other kinds of problems in distributed data processing systems, notably those that require both the alignment of data partitions and the assignment of jobs to machines. As demonstrated in our experiments, even a pairwise spatial join can have a more balanced execution and reduced query cost by using the methods we propose to schedule the query fragments to machines. In this usage, a single instance of SM for the only existing pair of datasets will suffice, with the residual load of previous steps zeroed ($\boldsymbol{u} = 0$). Another application is the scheduling of MapReduce jobs. Each task has a set of key-value pairs generated by a *map* function, and each machine may report the same key, producing key slices. Slices with the same key are aligned and processed by only one *reduce* function (alignment of partitions), which computes the desired result by applying the predicate algorithm (*reduce* function). Therefore, we

believe that the generalization of our models and algorithms in this regard is a promising area for future work.

In our work, the number of jobs for a specific query is determined by the number of data partitions and the intersection of the extent area of the datasets. The number of jobs, in turn, is a major factor in the complexity of the studied methods. Although it is possible to reduce the number of partitions to in turn reduce the optimization time by concatenating histogram cells, this may reduce estimation accuracy and create more skewed partitions, often resulting in unbalanced query execution. Conversely, a large number of partitions increases the opportunity for using parallelism, which is often efficient when using large clusters to process large queries. As today's clusters commonly have hundreds of machines, improving the methods in this respect will be pertinent in future research.

Notes

1. The term scheduling in the MapReduce literature is reserved for the distribution of cluster resources to multi-user loads, similar to multi-query database loads.
2. https://mapas.ibge.gov.br
3. Image Processing and Geoprocessing Laboratory: https://lapig.iesa.ufg.br
4. http://gis-lab.info/qa/vmap0-eng.html
5. Geospatial Data Abstraction Library: www.gdal.org
6. Geometry Engine: https://libgeos.org
7. http://clang.llvm.org
8. http://www.diku.dk/pisinger/codes.html
9. https://www.ibm.com/products/ilog-cplex-optimization-studio
10. http://golang.org
11. https://aws.amazon.com/ec2/instance-types
12. https://hadoop.apache.org/
13. http://www.openstreetmap.org

Disclosure statement

No potential conflict of interest was reported by the author(s).

Funding

This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior–Brasil (CAPES)—Finance Code 001, FAPESP proc. 14/50937-1, and FAPESP proc. 15/24485-9.

Notes on contributors

Thiago Borges de Oliveira has a PhD from the Federal University of Goiás and is a professor of Computer Science at the Exact Sciences Academic Unit of the Universidade Federal de Jataí in Goiás, Brazil. His main area of research is on algorithms for distributed processing of spatial data. In this paper, he conceived the approach to query optimization, was responsible for code development and experimentation, and contributed to the article preparation.

Fábio M. Costa is a professor of Computer Science at the Instituto de Informática of the Federal University of Goiás at Goiânia, GO, Brazil. His research interests span different aspects of

middleware platforms to support distributed computing, including runtime adaptation, services computing, and computing in the edge-cloud continuum. His contribution to this paper relates to the distributed execution model for multiway join queries and article preparation.

Les R. Foulds was born in New Zealand, has a PhD from Virginia Tech, USA, and is a visiting professor at the Instituto de Informática of the Federal University of Goiás at Goiânia, GO, Brazil, where he lives permanently. He has travelled extensively and has taught and researched at universities in North and South America, Europe, Asia, Australia, and New Zealand. His main area of academic interest is in the development of models and techniques for decision-making and resource allocation in manufacturing, logistics, and scheduling. In this paper, he contributed with the Lagrangian relaxation method, the parametric analysis of *f*, and article preparation.

Humberto J. Longo is a titular professor at the Instituto de Informática of the Federal University of Goiás at Goiânia, GO, Brazil. His research interests include Algorithms and Combinatorial Optimization. In this paper, he proposed the use of the Subgradient Optimization method to solve the Lagrangian relaxation of the SM Integer Programming model, supported the development of the respective computational code, and contributed to the article preparation.

ORCID

Thiago Borges de Oliveira http://orcid.org/0000-0002-9603-0356
Fábio M. Costa http://orcid.org/0000-0003-1038-8873
Les R. Foulds http://orcid.org/0000-0001-7538-8263
Humberto J. Longo http://orcid.org/0000-0002-0712-7376

Data and codes availability statement

The set of public spatial datasets can be freely obtained from the Brazilian Institute of Geography and Statistics (IBGE) at https://mapas.ibge.gov.br, from the Image Processing and Geoprocessing Laboratory (LAPIG) of the Institute of Social and Environmental Studies (IESA)—UFG at https://lapig.iesa.ufg.br, from the Digital Chart of the World (DCW), at http://gis-lab.info/qa/vmap0-eng.html, and from the TIGER 2015 spatial database at Bureau (2015). The source code that support the findings of this study is available at https://doi.org/10.6084/m9.figshare.19166327 or https://github.com/thborges/dgeoijgis, and makes use of the Geospatial Data Abstraction Library (GDAL), available under an MIT style Open Source License at https://www.gdal.org, and the Geometry Engine (GEOS), available under the terms of GNU Lesser General Public License (LGPL) at https://libgeos.org. For solving the linear integer models by LP and LR methods, our code links to IBM ILOG CPLEX Optimization Studio, available under an academic licence at https://www.ibm.com/products/ilog-cplex-optimization-studio or to GNU Linear Programming Kit (GLPK), available under the GNU General Public License at https://www.gnu.org/software/glpk. LR method uses Pisinger's Minknap algorithm, available free of charge for academical purposes at http://www.diku.dk/∼pisinger/codes.html. The authors of these software are not affiliated with this research.

References

Acharya, S., Poosala, V., and Ramaswamy, S., 1999. Selectivity estimation in spatial databases. *Acm Sigmod Record*, 28 (2), 13–24.

Afrati, F.N. and Ullman, J.D., 2011. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering*, 23 (9), 1282–1298.

Aji, A., Wang, F., and Saltz, J.H., 2012. 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*, Redondo Beach, CA, USA, 309–318.

Aji, A., *et al.*, 2013. Hadoop GIS: a high performance spatial data warehousing system over MapReduce. *Proceedings of the VLDB Endowment*, 6 (11), 1009–1020.

Anstreicher, K.M. and Wolsey, L.A., 2009. Two "well-known" properties of subgradient optimization. *Mathematical Programming*, 120 (1), 213–220.

Bazaraa, M.S., Jarvis, J.J., and Sherali, H.D., 2009. *Linear programming and network flows*. 4th ed. Hoboken, NJ: Wiley.

Bertsimas, D. and Tsitsiklis, J., 1997. *Introduction to linear programming*. Vol. 1. Belmont, MA: Athena Scientific.

Bhattu, S.N., *et al.*, 2020. Generalized communication cost efficient multi-way spatial join: revisiting the curse of the last reducer. *GeoInformatica*, 24 (3), 557–589.

Brinkhoff, T., Kriegel, H.P., and Seeger, B., 1996. Parallel processing of spatial joins using R-trees. In: *Proceedings of the IEEE International Conference on Data Engineering*, New Orleans, LA, USA. 258–265.

Bureau, U.C., 2015. 2015 tiger/line shapefiles (machine-readable data files). Available from: https://www.census.gov/geographies/mapping-files/time-series/geo/tiger-line-file.2015.html.

Cheng, C., Song, X., and Zhou, C., 2013. Generic cumulative annular bucket histogram for spatial selectivity estimation of spatial database management system. *International Journal of Geographical Information Science*, 27 (2), 339–362.

de Oliveira, T.B., 2017. *Efficient processing of multiway spatial join queries in distributed systems*. Thesis (PhD). Instituto de Informática, Universidade Federal de Goiás. Available from: http://repositorio.bc.ufg.br/tede/handle/tede/8033.

de Oliveira, T.B., Costa, F.M., and Rodrigues, V.J.S., 2017. Distributed execution plans for multiway spatial join queries using multidimensional histograms. *Journal of Information and Data Management*, 7 (3), 199–214.

de Oliveira, T.B., *et al.*, 2023. Parametric analysis of *f*: Controlling the consumption of computational resources in query scheduling. 1–10.

Dean, J. and Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51 (1), 107–113.

Doulkeridis, C. and Nørvåg, K., 2014. A survey of large-scale analytical query processing in MapReduce. *The VLDB Journal*, 23 (3), 355–380.

Du, Z., *et al.*, 2017. An effective high-performance multiway spatial join algorithm with spark. *ISPRS International Journal of Geo-Information*, 6 (4), 96.

Eldawy, A., Alarabi, L., and Mokbel, M.F., 2015. Spatial partitioning techniques in SpatialHadoop. *Proceedings of the VLDB Endowment*, 8 (12), 1602–1605.

Eldawy, A., *et al.*, 2021. Beast: Scalable exploratory analytics on spatio-temporal data. In: *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, Queensland, Australia, 3796–3807.

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

Eldawy, A. and Mokbel, M.F., 2016. The era of big spatial data: a survey. *Foundations and Trends⋈ in Databases*, 6 (3–4), 163–273.

Eldawy, A., *et al.*, 2017. Sphinx: empowering Impala for efficient execution of SQL queries on big spatial data. *In*: *Advances in spatial and temporal databases, lecture notes in computer science*. Vol. 10411. Cham: Springer, 65–83.

Fisher, M.L., 1985. An applications oriented guide to Lagrangian relaxation. *Interfaces*, 15 (2), 10–21.

Fisher, M.L., 2004. The Lagrangian relaxation method for solving integer programming problems. *Management Science*, 50 (12 Supplement), 1861–1871.

Fornari, M.R., Comba, J.L.D., and Iochpe, C., 2006. Query optimizer for spatial join operations. *In*: *Proceedings of the ACM International Symposium on Advances in Geographic Information Systems*, Arlington, VA, USA, 219–226.

Geoffrion, A.M., 1974. Lagrangian relaxation for integer programming. In: *Approaches to integer programming*. Berlin, Heidelberg, Germany: Springer, 82–114.

Goffin, J.L., 1977. On convergence rates of subgradient optimization methods. *Mathematical Programming*, 13 (1), 329–347.

Graefe, G., 1993. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25 (2), 73–169.

Gupta, H. and Chawda, B., 2014. ϵ-Controlled-replicate: an improved controlled-replicate algorithm for multi-way spatial join processing on map-reduce. *In*: B. Benatallah, A. Bestavros, Y. Manolopoulos, A. Vakali, and Y. Zhang, eds. *Web information systems engineering. Lecture notes in computer science*. Vol. 8787. Cham, Switzerland: Springer, 278–293.

Gupta, H., *et al*., 2013. Processing multi-way spatial joins on Map-Reduce. *In*: *Proceedings of the International Conference on Extending Database Technology*, Genoa, Italy, 113–124.

Hall, J.A.J., 2010. Towards a practical parallelisation of the simplex method. *Computational Management Science*, 7 (2), 139–170.

Held, M. and Karp, R.M., 1971. The traveling-salesman problem and minimum spanning trees: part II. *Mathematical Programming*, 1 (1), 6–25.

Held, M., Wolfe, P., and Crowder, H.P., 1974. Validation of subgradient optimization. *Mathematical Programming*, 6 (1), 62–88.

Huang, Z., *et al*., 2011. Building the distributed geographic SQL workflow in the grid environment. *International Journal of Geographical Information Science*, 25 (7), 1117–1145.

Kellerer, H., Pferschy, U., and Pisinger, D., 2004. *Knapsack problems*. Berlin, Heidelberg, Germany: Springer Verlag.

Klau, G., and Reinert, K., 2007. Lagrangian relaxation: an overview. *Lecture Notes in Discrete Mathematics in Bioinformatics WS*, 7 (8), 14–21.

Kwon, Y., *et al*., 2012. SkewTune: mitigating skew in MapReduce applications. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Scottsdale, Arizona, USA, 25–36.

Luo, G., Naughton, J.F., and Ellmann, C.J., 2002. A non-blocking parallel spatial join algorithm. In: *Proceedings of the IEEE International Conference on Data Engineering*, San Jose, CA, USA, 697–705.

Mamoulis, N. and Papadias, D., 2001a. Multiway spatial joins. *ACM Transactions on Database Systems*, 26 (4), 424–475.

Mamoulis, N. and Papadias, D., 2001b. Selectivity estimation of complex spatial queries. In: C.S. Jensen, M. Schneider, B. Seeger, and V.J. Tsotras, eds. *Advances in spatial and temporal databases. Lecture notes in computer science*. Vol. 2121. Berlin, Heidberg, Germany: Springer, 155–174.

Martello, S., Soumis, F., and Toth, P., 1997. Exact and approximation algorithms for makespan minimization on unrelated parallel machines. *Discrete Applied Mathematics*, 75 (2), 169–188.

Mishra, P. and Eich, M.H., 1992. Join processing in relational databases. *ACM Computing Surveys*, 24 (1), 63–113.

Moseley, B., *et al*., 2011. On scheduling in map-reduce and flow-shops. *In*: *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, San Jose, CA, USA, 289–298.

Nidzwetzki, J.K. and Güting, R.H., 2017. Distributed SECONDO: an extensible and scalable database management system. *Distributed and Parallel Databases*, 35 (3–4), 197–248.

Özsu, M.T. and Valduriez, P., 2011. *Principles of distributed database systems*. 3rd ed. New York, NY: Springer.

Papadias, D. and Arkoumanis, D., 2002a. Approximate processing of multiway spatial joins in very large databases. In: *Advances in database technology — EDBT 2002*. Berlin; Heidelberg: Springer Berlin Heidelberg, 179–196.

Papadias, D. and Arkoumanis, D., 2002b. Search algorithms for multiway spatial joins. *International Journal of Geographical Information Science*, 16 (7), 613–639.

Papadias, D., Mamoulis, N., and Theodoridis, Y., 1999. Processing and optimization of multiway spatial joins using R-trees. *In*: *Proceedings of the ACM Symposium on Principles of Database Systems*, Philadephia, PA, USA, 44–55.

Papadias, D., Mamoulis, N., and Theodoridis, Y., 2001. Constraint-based processing of multiway spatial joins. *Algorithmica*, 30 (2), 188–215.

Patel, J.M. and DeWitt, D.J., 2000. Clone join and shadow join: two parallel spatial join algorithms. In: *Proceedings of the ACM International Symposium on Advances in Geographic Information Systems*, McLean, VA, USA, 56–61.

Pavlo, A., *et al.*, 2009. A comparison of approaches to large-scale data analysis. *In*: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Providence, RI, USA, 165–178.

Pisinger, D., 1997. A minimal algorithm for the 0-1 Knapsack problem. *Operations Research*, 45 (5), 758–767.

Roh, Y.J., *et al.*, 2010. Hierarchically organized skew-tolerant histograms for geographic data objects. *In*: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Indianapolis, IN, USA, 627–638.

Sabek, I. and Mokbel, M.F., 2017. On spatial joins in MapReduce. *In*: *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, New York, NY, USA.

Sivasubramaniam, A., 2001. Selectivity estimation for spatial joins. In: *Proceedings of the IEEE International Conference on Data Engineering*, Berlin, Heidelberg, Germany, 368–375.

Stonebraker, M., *et al.*, 2010. MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53 (1), 64–71.

Sun, C., *et al.*, 2006. Exploring spatial datasets with histograms. *Distributed and Parallel Databases*, 20 (1), 57–88.

Tsitsigkos, D., *et al.*, 2019. Parallel in-memory evaluation of spatial joins. *In*: *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, New York, NY, USA, 516–519.

Vazirani, V.V., 2001. *Approximation algorithms*. Berlin, Heidelberg, Germany: Springer Science & Business Media.

Verma, A., Cherkasova, L., and Campbell, R.H., 2012. 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)*, Washington, DC, USA, 11–18.

Vu, T., *et al.*, 2020. Using deep learning for big spatial data partitioning. *ACM Transactions on Spatial Algorithms and Systems*, 7 (1), 1–37.

Xie, D., *et al.*, 2016. Simba: efficient in-memory spatial analytics. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Francisco, CA, USA, 1071–1085.

You, S., Zhang, J., and Gruenwald, L., 2015. Spatial join query processing in cloud: analyzing design choices and performance comparisons. In: *Proceedings of the 44th International Conference on Parallel Processing Workshops*, Beijing, China, 90–97.

Yu, C.T. and Chang, C., 1984. Distributed query processing. *ACM Computing Surveys*, 16 (4), 399–433.

Yu, J., Wu, J., and Sarwat, M., 2015. GeoSpark: a cluster computing framework for processing large-scale spatial data. In: *Proceedings of the SIGSPATIAL International Conference on Advances in Geographic Information Systems*, Bellevue, WA, USA, 70:1–70:4.

Yu, J., Zhang, Z., and Sarwat, M., 2018. Spatial data management in Apache spark: the GeoSpark perspective and beyond. *GeoInformatica*, 23 (1), 37–78.

Zaharia, M., *et al.*, 2016. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59 (11), 56–65.

Zhang, S., *et al.*, 2009. SJMR: parallelizing spatial join with MapReduce on clusters. In: *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, New Orleans, LA, USA, 1–8.

Zhong, Y., *et al.*, 2012. Towards parallel spatial query processing for big spatial data. In: *Proceedings of the International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, Shanghai, China, 2085–2094.