Purdue University
Purdue e-Pubs

Open Access Dissertations | Theses and Dissertations

12-2016

Efficient processing of similarity queries with applications

Mingjie Tang
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_dissertations

 Part of the Computer Sciences Commons

Recommended Citation
Tang, Mingjie, "Efficient processing of similarity queries with applications" (2016). *Open Access Dissertations*. 1014.
https://docs.lib.purdue.edu/open_access_dissertations/1014

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Graduate School Form
30 Updated 12/26/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Mingjie Tang

Entitled

EFFICIENT PROCESSING OF SIMILARITY QUERIES WITH APPLICATIONS

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

WALID G. AREF Chair	ELISA BERTINO
SUNIL PRABHAKAR	SONIA FAHMY

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): WALID G. AREF

Approved by: SUNIL PRABHAKAR/WILLIAM J. GORMAN 9/6/2016

Head of the Departmental Graduate Program Date

EFFICIENT PROCESSING OF SIMILARITY QUERIES WITH APPLICATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Mingjie Tang

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2016

Purdue University

West Lafayette, Indiana

To my family

ACKNOWLEDGMENTS

I am thankful to my advisor, Walid G. Aref, who leads me to explore and understand the beauty of database system. He is a brilliant researcher who always pushes me to think further, and shares his experience about the research process. I will not forget these days he revised our paper until the middle night, even if he was sick. I am especially fortunate to work with this great person and benefit from his perspective.

The work in this dissertation is the result of collaboration with many other people. Chapter 2 was a joint work with Ruby Y. Tahboub, Walid G. Aref, Mikhail Atallah, Qutaibah Malluhi, Mourad Ouzzani and Yasin. Siva [1]. Chapter 3 introduces the new index for efficient hamming distance query processing developed with Yongyang Yu, Walid G. Aref, Qutaibah Malluhi, and Mourad Ouzzani [2]. Chapter 4 was a joint work with Yongyang Yu, Walid G. Aref, Ahmed R. Mahmood, Qutaibah Malluhi, and Mourad Ouzzani [3, 4].

I am also very fortunate to invite Professor Bertio Elisa, Professor Sunil Prabhakar and Professor Sonia Fahmy to be my commit members, and learn comments from them. These suggestions enhance my dissertation greatly. Beyond direct collaborators on this project, many other people contributed to my graduate work and made Purdue an unforgettable experience. The friends at Purdue database research group include Yongyang Yu, Ahmed M. Aly and Ahmed R. Mahmood always offer their help to improve this project. Pei He, Yefei Sun, Bo Sang, Shandian Zhe, He Zhu, Dong Su, Ziang Ding and Youhan Fang were great fun to hang out with and great collaborators on some neat ideas. I also want to express my best regards to my friend Weihang Wang, she helped me to overcome many unpredictable difficulties in the last eight years.

Last but not least, I want to thank my father Guanglun Tang, mother Shirong Chen, sister Yeyao Tang, brother Zaiye Chen and other friends for their unwavering support throughout my PhD. Without their support, I can not do anything.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Tang, Mingjie PhD, Purdue University, December 2016. Efficient Processing of Similarity Queries with Applications. Major Professor: Walid G.Aref.

Today, a myriad of data sources, from the Internet to business operations to scientific instruments, produce large and different types of data. Many application scenarios, e.g., marketing analysis, sensor networks, and medical and biological applications, call for identifying and processing similarities in ”big” data. As a result, it is imperative to develop new similarity query processing approaches and systems that scale from low dimensional data to high dimensional data, from single machine to clusters of hundreds of machines, and from disk-based to memory-based processing. This dissertation introduces and studies several similarity-aware query operators, analyzes and optimizes their performance.

The first contribution of this dissertation is an SQL-based Similarity Group-by operator (SGB, for short) that extends the semantics of the standard SQL Group-by operator to group data with similar but not necessarily equal values. We realize these SGB operators by extending the Standard SQL Group-by and introduce two new SGB operators for multi-dimensional data. We implement and test the new SGB operators and their algorithms inside an open-source centralized database server (PostgreSQL).

In the second contribution of this dissertation, we study how to efficiently process Hamming-distance-based similarity queries (Hamming-distance select and Hamming-distance join) that are crucial to many applications. We introduce a new index, termed the HA-Index, that speeds up distance comparisons and eliminates redundancies when performing the two flavors of Hamming distance range queries (namely, the selects and joins).

In the third and last contribution of this dissertation, we develop a system for similarity query processing and optimization in an in-memory and distributed setup for big spatial data. We propose a query scheduler and a distributed query optimizer that use a new cost

model to optimize the cost of similarity query processing in this in-memory distributed setup. The scheduler and query optimizer generates query execution plans that minimize the effect of query skew. The query scheduler employs new spatial indexing techniques based on bloom filters to forward queries to the appropriate local sites. The proposed query processing and optimization techniques are prototyped inside Spark, a distributed main-memory computation system.

1 INTRODUCTION

1.1 Motivation

The large amount of data accumulated from sensors, social networks, computational sciences, and location-aware services calls for advanced similarity query processing techniques. Consider the following application scenarios for similarity query processing.

Motivating Scenario 1: DNA Microarray Data Analysis

Molecular biologists strive to understand the biological processes (e.g., diseases) that are influenced by the activity of multiple gene expressions. One two-dimensional DNA microarray (also referred to as biochip) is a kind of data matrix, in which each cell contains a numeric value that corresponds to a given gene. A cell value is a correlated attribute that represents the intensity of a gene within a sample. Two-dimensional microarray cells are clustered into patterns (i.e., groupings of genes) that are used to explain the biological process of interest. In contrast to costly standalone clustering operators that are external to the database, biological databases can benefit from a Similarity Group By operator (SGB, for short) to efficiently group two-dimensional microarray data within the database.

Motivating Scenario 2: Spatial Computing

Spatial computing [5] is a fundamental problem for various areas, e.g., GIS, data mining, machine learning, and recommender systems. For example, a GPS allows people to know their locations, nearby facilities, and routes to reach place of interest. Uber (e.g., Uberpool) provides the service for drivers to share their cars during the traffic rush hour. Location-based augmented reality games (e.g., Pokeman go) enable players to use a mobile device's GPS capability to locate, capture, battle, and train virtual creatures. Recommendation systems suggest more relevance products (e.g., advertisements, news and friends) to users based on their spatial location information. Given the large volumes of captured

spatial data, a similarity query would retrieve data points that are inside a spatial range, or data points that are among the k nearest neighbors to the query's focal point.

Motivating Scenario 3: Location-based Social Networks

Location-based social networks allow users to share their current locations via check-in services in social networks, e.g., Foursquare and Facebook. Social network applications model human mobility patterns using geographic locations, time of movements, and social ties (i.e., friends). Multi-dimensional spatial and temporal data are correlated in nature. Supporting a similarity group-by operator over correlated multi-dimensional location data is critical to correctly detecting groups of moving objects that share similar movement patterns.

Motivating Scenario 4: Content-based Image Searching

Many companies, e.g., Google [1], Baidu [2], and Bing [3] have commercialized their image-based search engines that use similarity search over billions of images. Typically, features extracted from the images are modeled by high-dimensional vectors, e.g., color histograms, texture features, edge orientations, etc. Given a query image and its high-dimensional vector of extracted features, a similarity search is conducted to find the images that are similar to or that are k-closest to the query image.

Motivating Scenario 5: Content-based Webpage Analysis

Similarity search is widely used in detecting duplicate web pages over the internet. Detecting duplicate web pages is useful in many real-world applications, e.g., web mirroring, plagiarism, and spam detection [6]. Each web page is preprocessed to generate a high-dimensional vector of topics that appear in this page (typically around 250 topics per page, e.g., [7]). High-dimensional similarity search is involved to find documents in the web that are similar to the query document.

[1] http://www.google.com/imghp
[2] http://stu.baidu.com/
[3] https://www.bing.com/images/

1.2 Taxonomy of Similarity Query Processing

There is a wide variety of applications that rely on similarity query processing. These include databases [8, 9], data mining [10], machine learning [11–13], and GIS [5, 14, 15]. In these applications, given a similarity function and a set of objects, a similarity query returns a set of objects that satisfy a certain similarity criterion. Following the introduction for similarity querying in [16], we categorize similarity queries along six dimensions as illustrated in the taxonomy below.

- **Data Types**

  One dimension of the taxonomy is the data type where the concept of similarity search applies. Similarity query operators, e.g., Similarity Join, Similarity Group By, k-Nearest Neighbor (kNN, for short) and kNN Join have been proposed as basic database operators in [9] for numeric data using Euclidean distance to signify similarity. In addition, there is a rapidly growing amount of multimedia data (e.g., text/documents, images, fingerprints and videos), and queries asking for objects similar to a given one based on content similarity (e.g., [11, 16, 17]). In biological databases, DNA/RNA or protein sequences are the basic objects, where an object is modeled as a string. Similarity search becomes the problem to find a given sequence of characters inside a longer sequence that is within a certain edit distance threshold from the query [18–20]. Recently, there has been an explosion in the amounts of location data produced by various devices, e.g., smart phones, satellites, and medical devices [5, 14]. For example, NASA satellite data archives have exceeded 500 TB and have been continuously growing. Twitter data with location information, e.g., latitude and longitude, are accumulating more than 20 million tweets per day.

- **Data Dimensionality**

  The second aspect of the taxonomy is that of data dimensionality. Each object can be represented or modeled by one-dimensional [9], multi-dimensional [1, 4], or high-dimensional vector data [2].

- **Distance Metric**

  The distance metric used to detect similarity between two objects needs to be carefully chosen to be suitable for certain application scenarios. For example, Euclidean Distance and Manhattan Distance are widely used in business intelligence [1, 9]. Hamming Distance is being used for content-based search [2], and cosine distance is being used to measure the similarity of two documents for web search. Edit Distance is adopted for the similarity of two strings (biological sequences) [18, 19]. More details on metric space distance functions can be found in [10, 16].

- **Data Operators**

  Three types of similarity queries have been receiving particular attention, namely the Similarity Range Select, k-Nearest Neighbor (kNN), and Similarity Group By (SGB). Similarity Range Select retrieves all elements within a distance threshold to the query. kNN retrieves the k objects nearest to the query. The similarity between Range Select and kNN can be easily extended to Similarity-Range-Join (a.k.a. Spatial-Range-Join) and kNN-Join. SGB operators divide data objects with similar relationship but not necessarily equivalent relationship into the same groups.

- **Data Storage**

  The data can be stored on disk or in main memory. Traditionally, data has resided on the disk. The similarity computation time mainly counts the transmission time between CPU and disk. This is due to the fact that the I/O accessing time is the dominant factor in some applications. Recently, with the advancement in memory technology, memory has become cheaper and larger in size, which makes it possible to keep data in the main memory. This makes it possible to process data once it arrives.

- **Data Platform**

  Lately, the platform to compute the similarity queries on has undergone dramatic changes. The join operation for two big tables is usually very expensive and takes massive response time under the setting of one machine. Therefore, there is a trend

to move from computation platform from one single machine to clusters of hundreds of machines (e.g., using Hadoop MapReduce [4], Hbase [5], Spark [6]), and to develop parallel computation frameworks to support similarity joins in the corresponding distributed platforms.

1.3 Summary of Results

1. Similarity Group By

   - We introduce two new SGB operators in the multi-dimensional space. We define the class of order-independent SGB operators that produce the same results regardless of the order in which the input data is presented to them. Using the notion of interval graphs borrowed from graph theory, we prove that, for certain SGB operators, there exist order-independent implementations. For each of these operators, we provide a sample algorithm that is order-independent. Also, we prove that for other SGB operators, there does not exist an order-independent implementation for them, and hence these SGB operators are ill-defined and should not be adopted in extensions to SQL to realize similarity group-by.
   - We present new definitions of SGB operator over relational data. The first operator is the clique (or distance-to-all) SGB, where all the tuples in a group are within some distance ϵ from each other. The second operator is the distance-to-any SGB, where a tuple belongs to a group if the tuple is within some distance ϵ from any other tuple in the group. When a tuple satisfies the membership criterion of multiple groups, we support three different semantics that would either eliminate the tuple, put the tuple in any one group, or create a new group for this tuple.

[4] http://hadoop.apache.org/
[5] http://hbase.apache.org/
[6] https://spark.apache.org/

- We realize and test new SGB operators and their algorithms inside PostgreSQL, an open source SQL-based database management system. The experiments demonstrate that the proposed algorithms can achieve up to three orders of magnitude enhancement in performance over the baseline approaches. Moreover, the performance of the proposed SGB operators is comparable to that of relational Group-by, and outperform state-of-the-art clustering algorithm (i.e., *K-means* [21], *DBSCAN* [22] and *BIRCH*) [23] from one to three orders of magnitude.

2. Hamming distance-based Similarity querying

   - We focus on two variants of the Hamming distance-based similarity querying, namely Hamming-distance-based select and Hamming-distance-based join (for short, Hamming-select and Hamming-join, respectively). We propose a new index, termed the HA-Index, that is designed to reduce redundant and duplicate distance computations during the Hamming-distance search. The HA-Index assumes that the underlying datasets are preprocessed; data is mapped from the high-dimensional space into one-dimensional binary codes that are fixed-length strings of 0's and 1's. Then, the binary codes are sorted using the Gray ordering [24]. Sorting the binary codes in this way helps group together multiple binary codes that share a common substring or non-contiguous yet similar sequences of bits. By computing the distances between the query binary code and similar substrings, many redundant distance computations can be avoided based on properties of binary codes.
   - We introduce two approaches to improve the performance of Hamming-select and Hamming-join. The first approach uses a simple Radix-tree index from the literature. The second approach is based on the HA-Index with both a static and a dynamic version. We also introduce the maintenance operations, i.e., build, insert, update, and search operations, for the dynamic HA-Index. For Hamming-joins over large and skewed data, we propose an efficient data

partitioning technique for balancing data computations among servers, and introduce a distributed version of the HA-Index to reduce data shuffling inside MapReduce.

- We conduct an extensive experimental study using real datasets and demonstrate that the HA-Index (i) enhances the performance of Hamming-select and Hamming-join by two orders of magnitude over state-of-the-art techniques, and (ii) saves memory usage by more than one order of magnitude. We also evaluate how the proposed index improves approximate algorithms for kNN-select and kNN-join operations.

3. Similarity query processing and optimization for in-memory distributed computing environments

   - We develop a new computing system for processing and optimizing similarity queries over in-memory distributed spatial data.
   - We address data and query skew issues to improve load balancing while executing similarity operators, e.g., spatial-range-join and kNN-join, by generating cost-optimized query execution plans over in-memory distribute spatial data.
   - We introduce a new light-weight yet efficient spatial Bloom filter to reduce communication cost.
   - We realize the introduced query processing and optimization techniques inside of Spark. We use the developed prototype system to conduct a large-scale evaluation on real spatial data and common benchmark algorithms and compare our system against state-of-the-art distributed spatial data processing systems. Experimental results show enhancements in performance by up to an order of magnitude over existing in-memory and distributed spatial systems.

Table 1.1.: Contributions of this dissertation based on the aforementioned taxonomy for similarity query processing

	Similarity Group By	**Hamming-Distance-Based Similarity Search**	**Distributed In-memory Similarity Query Processing and Optimization**
Data Types:	Numerical	Text and Image	Spatial data
Data Dimensionality:	Multi-Dimensional	High-Dimensional	Multi-Dimensional and Polygons
Data Distance:	Euclidean and Manhattan Distance	Hamming Distance	Euclidean and Cosine
Data Operators:	Similarity Group By	Similarity-select and Similarity-Join	Spatial-range-select, Spatial-range-join kNN-select, kNN-join
Data Storage:	Disk	Disk	Memory
Data Platform:	Centralized (e.g., PostgreSQL)	Centralized and Parallel (e.g., MapReduce)	Parallel (e.g., Spark)

1.4 Dissertation Plan

We have published parts of the work presented in this dissertation [1–4, 25]. The work on Similarity Group-by is presented in [1]. The study of the order independent properties for SGB operator is presented in [25]. The Hamming-distance-based similarity query processing is presented in [2]. The similarity query processing over in-memory distributed spatial data is presented in [3, 4]. The contributions of this dissertation based on the aforementioned taxonomy for similarity query processing is given in Table 1.1.

This rest of this dissertation is organized as follows. Chapter 2 introduces the Similarity Group-By Operator for the relational data. Chapter 3 covers techniques for Hamming-distance-based query processing. Chapter 4 presents the system to support distributed in-memory similarity query processing for spatial data. Chapter 5 concludes this dissertation and discusses possible areas for future work.

2 SIMILARITY GROUPBY OPERATORS FOR MULTI-DIMENSIONAL RELATIONAL DATA

2.1 Introduction

The deluge of data accumulated from sensors, social networks, emerging computational sciences, and location-aware services calls for advanced querying and analytic that are often dependent on efficient aggregation and summarization techniques. The SQL group-by operator of a relational database is one main construct that is used in conjunction with aggregate operations to partition the data into groups and produce useful summaries. Grouping is usually performed by aggregating into the same groups tuples with equal values on a certain subset of the attributes. However, many applications are often interested in grouping based on *similar* rather than strictly equal values.

Clustering [10] is a well-known technique for grouping similar data items in the multi-dimensional space. In most cases, clustering is performed outside of the database system. Moving the data outside of the database to perform the clustering and then back into the database for further processing results in a costly impedance mismatch. Moreover, based on the needs of the underlying applications, the output clusters may need to be further processed by SQL to filter out some of the clusters and perform further SQL operations on the remaining clusters. Hence, it is of great benefit to develop practical and fast similarity group-by operators that can be embedded within SQL to avoid the impedance mismatch and to benefit from the processing power of all the other SQL operators.

SQL-based Similarity Group-by (SGB) operators have been proposed in [9] to support several semantics to group similar but not necessarily equal data. Although many applications can benefit from using existing SGB over Group-by, a key shortcoming of SGB operators is that they focus on one-dimensional data. Consequently, data can only be approximately grouped based on one attribute at a time. As a result, clusters reflecting cor-

related spatial and multi-attributes cannot be detected properly to form meaningful groups. Furthermore, supporting SGB operators using existing SQL constructs and procedures may result in high execution times in contrast to a native database intergrated SGB operator.

In this chapter, we introduce new similarity-based group-by operators that group multi-dimensional data using various metric distance functions. More specifically, we propose two SGB operators, namely SGB-All and SGB-Any, for grouping multi-dimensional data. SGB-All forms groups such that a tuple or a data item, say o, belongs to a group, say g, if and only if o is within a user-defined threshold from all other data items in g. In other words, each group in SGB-All forms a clique of nearby data items in the multi-dimensional space. For example, all the two-dimensional points (a-e) in Figure 2.1a are within Distance 3 from each other and hence form a clique. They are all reported as members of one group as they are all part of the output of SGB-All. In contrast, SGB-Any forms groups such that a tuple or a data item, say o, belongs to a group, say g, if and only if o is within a user-defined threshold from at least one other data item in g. For example, all the two dimensional points in Figure 2.1b form one group. Point a is within Distance 3 from Point c that in turn is within Distance 3 from Points b, d, and f. Furthermore, Point e is within Distance 3 from Point d, etc. Therefore, Points a-h of Figure 2.1b are reported as members of one group as part of the output of SGB-Any.

Notice that in the SGB-All operator, a data item may qualify the membership criterion of multiple groups. For example, Data item c in Figure 2.1a forms a clique with two groups. In this case, we propose three semantics namely, *on-overlap join-any*, *on-overlap eliminate*, and *on-overlap form-new-group* for handling these overlapped data items. We provide efficient algorithms for handling correlated multi-dimensional attributes using each of the two SGB operators, along with the three alternatives for handling tuples that overlap multiple groups. The proposed algorithms use a filter-refine paradigm. In the filter step, a fast yet conservative check is performed to identify the data items that are candidates to form groups. Some of the data items resulting from the filter step will end up being false-positives that will be discarded. The refinement step eliminates the false-positives to produce the final output groups. Notice that for the case of SGB-Any, a data item cannot

belong to multiple groups. For example, consider a data item, say o, that is a member of two groups, say g_1 and g_2, i.e., o is within distance $epsilon$ from at least one other data item in each of g_1 and g_2. In this case, based on the semantics of SGB-Any, Groups g_1 and g_2 merge into one encompassing bigger group that contains all members of g_1, g_2 and common data item o.

The contributions of this chapter are summarized as follows:

1. We investigate new similarity group-by semantics and introduce two new operators, namely SGB-All and SGB-Any, for grouping multi-dimensional data from within SQL.

2. We present an extensible algorithmic framework to accommodate the various semantics of SGB-All and SGB-Any along with the various options to handle the overlapping data among multiple groups. We introduce effective optimizations for both operators.

3. We analyze the complexity of all the proposed algorithms. In addition, we prototype the two operators inside PostgreSQL and study their performance using the TPC-H benchmark. The experiments demonstrate that the proposed algorithms can achieve up to three orders of magnitude enhancement in performance. Moreover, the performance of the proposed SGB operators is comparable to that of relational Group-by.

The rest of this chapter proceeds as follows.

Section 2.2 provides background material. Section 2.3 introduces the new SGB operators. Section 2.4 presents application scenarios that demonstrate the use and practicality of the various proposed semantics for SGB operators. Sections 2.5 and 2.6 introduce the algorithmic frameworks for SGB-All and SGB-Any operators, respectively. Section 2.7 provides the complexity analysis of the proposed algorithms. Section 2.9 discusses the related work. Section 2.8 describes the in-database extensions to support the two operators and their performance evaluation from within PostgreSQL. Section 2.10 contains concluding remarks.

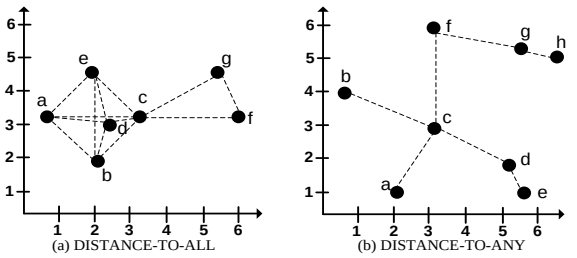


Figure 2.1.: The semantics of similarity predicates $\epsilon = 3$.

2.2 Preliminaries

In this section, we provide background definitions and formally introduce similarity-based group-by operators.

Definition 1 *A **metric space** is a space* $\boldsymbol{M} = \langle \mathbb{D}, \delta \rangle$ *in which the distance between two data points within a domain* $\mathbb{D}$ *is defined by a function* $\delta : \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{R}$ *that satisfies the properties of symmetry, non-negativity, and triangular inequality.*

We use the Minkowski distance L_p as the distance function δ. We consider the following two Minkowski distance functions. Let p_x be a data point in the multi-dimensional space of the form $p_x : \langle x_1, ..., x_d \rangle$ and p_{xy} is the value of the y^{th} dimension of p_x. Then,

- The Euclidean distance
  $L_2 : \delta_2(p_i, p_j) = \sqrt{\sum_y (p_{iy} - p_{jy})^2}$
- The maximum distance
  $L_\infty : \delta_\infty(p_i, p_j) = \max_y |p_{iy} - p_{jy}|$.

Definition 2 *A **similarity predicate** $\xi_{\delta,\epsilon}$ is a Boolean expression that returns TRUE for two multi-dimensional points, say p_i and p_j, iff the distance δ between p_i and p_j is less than or equal to ϵ, i.e., $\xi_{\delta,\epsilon}(p_i, p_j) : \delta(p_i, p_j) \leq \epsilon$. In this case, the two points are said to be similar.*

Definition 3 *Let T be a relation of tuples, where each tuple, say t, is of the form $t = \{GA_1, ..., GA_k, NGA_1, ..., NGA_l, \}$, the subset $GA_c = \{GA_1, ..., GA_k\}$ be the grouping*

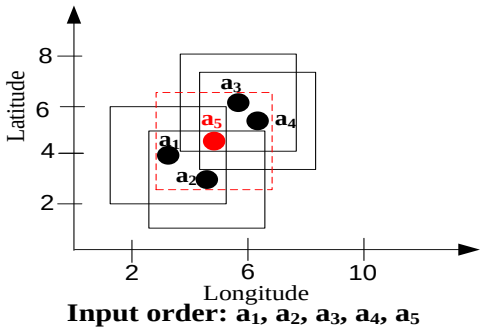


Figure 2.2.: Data points using $\epsilon = 3$ and L_∞.

attributes, the subset $NGA = \{NGA_1, ..., NGA_l\}$ *be the non-grouping attributes, and* $\xi_{\delta,\epsilon}$ *be a similarity predicate. Then, the* ***similarity Group-by operator*** $\mathcal{G}_{\langle GA_c,(\xi_{\delta,\epsilon})\rangle}(R)$ *forms a set of answer groups* G_s *by applying* $\xi_{\delta,\epsilon}$ *to the elements of* GA_c *such that a pair of tuples, say* t_i *and* t_j*, are in the same group iff* $\xi_{\delta,\epsilon}(t_{i\cdot GAc}, t_{j\cdot GAc})$.

Definition 4 *Given a set of groups* $G = \{g_1, ..., g_m\}$*, the* ***Overlap Set*** $Oset$ *is the set of tuples formed by the union of the intersections of all pairs of groups* $(g_1, ..., g_m)$*, i.e.,* $Oset = \cup_{(i,j)\in\{1..m\}}(g_i \cap g_j)$*, where* $i \neq j$*. In other words,* $Oset$ *contains all the tuples that belong to more than one group.*

For simplicity, we study the case when the set of grouping attributes, GA_c, contains only two attributes. In this case, we can view tuples as points in the two-dimensional space, each of the form $p{:}(x_1, x_2)$. We enclose each group of points by a bounding rectangle $R{:}(p_l, p_r)$, where points p_l and p_r correspond to the upper-left and bottom-right corners of R, respectively.

2.3 Similarity Group-By Operators

This section introduces the semantics of the two similarity-based group-by operators, namely, SGB-All and SGB-Any.

2.3.1 Similarity Group-By All (SGB-All)

Given a set of tuples whose grouping attributes form a set, say P, of two-dimensional points, where $P = \{p_1, ..., p_n\}$, the SGB-All operator $\check{\mathcal{G}}_{all}$ forms a set, say G_m, of groups of points from P such that $\forall g \in G_m$, the similarity predicate $\xi_{\delta,\epsilon}$ is TRUE for all pairs of points $\langle p_i, p_j \rangle \in g$, and g is maximal, i.e, there is no group g' such that $g \subseteq g'$. More formally,

$$\check{\mathcal{G}}_{all} = \{g \mid \forall p_i, p_j \in g,\ \xi_{\delta,\epsilon}(p_i, p_j) \wedge g\ is\ maximal\}$$

Figure 2.1 gives an example of two groups (a-e) and (c,f,g), where all pairs of elements within each group are within a distance $\epsilon \leq 3$. The proposed SQL syntax for the SGB-All operator is as follows:

```
SELECT column, aggregate-func(column)
FROM table-name
WHERE condition
GROUP BY column DISTANCE-TO-ALL [L2 | LINF] WITHIN ϵ
ON-OVERLAP [JOIN-ANY | ELIMINATE |FORM-NEW-GROUP]
```

SGB-All uses the following clauses to realize similarity-based grouping:

- DISTANCE-TO-ALL: specifies the distance function to be applied by the similarity predicate when deciding the membership of points within a group.
  - L2: L_2 (Euclidean distance).
  - LINF: L_∞ (Maximum infinity distance)
- ON-OVERLAP: is an arbitration clause to decide on a course of action when a data point is within Distance ϵ from more than one group. When a point, say p_i, matches the membership criterion for more than one group, say $g_1 \cdots g_w$, one of the three following actions are taken:
  - JOIN-ANY: the data point p_i is randomly inserted into any one group out of $g_1 \cdots g_w$.

- ELIMINATE: discard the data point p_i, i.e., all data points in $Oset$ (see Definition 4) are eliminated.
- FORM-NEW-GROUP: insert p_i into a separate group, i.e., form new groups out of the points in $Oset$.

Example 1 *The following query performs the aggregate operation $count$ on the groups formed by SGB-All on the two-dimensional grouping attributes GPSCoor-lat and GPSCoor-long. The L_∞ distance is used with Threshold $\epsilon = 3$.*

```
SELECT count(*)
FROM GPSPoints
GROUP BY GPSCoor-lat,GPSCoor-long DISTANCE-TO-ALL LINF
WITHIN 3
ON-OVERLAP <clause>
```

Consider Points a_1-a_5 in Figure 2.2 that arrive in the order $a_1, a_2, \cdots, a_5$ as in the figure. After processing a_4, the following groups satisfy the SGB-All predicates: $g_1\{a_1, a_2\}$ and $g_2\{a_3, a_4\}$. However, Data-point a_5 is within ϵ from a_1, a_2 in g_1 and a_3, a_5 in g_2. Consequently, an arbitration ON-OVERLAP clause is necessary. We consider the three possible semantics below for illustration.

With an ON-OVERLAP JOIN-ANY clause, a group is selected at random. If g_1 is selected, the resulting groups are $g_1\{a_1, a_2, a_5\}$ and $g_2\{a_3, a_4\}$, and the answer to the query is $\{3, 2\}$. With an ON-OVERLAP ELIMINATE clause, the overlapping point a_5 gets dropped; the resulting groups are $g_1\{a_1, a_2\}$ and $g_2\{a_3, a_4\}$, and the query output is $\{2, 2\}$. With an ON-OVERLAP FORM-NEW-GROUP clause, the overlapping point a_5 is inserted into a newly created group; the resulting groups are $g_1\{a_1, a_2\}$, $g_2\{a_3, a_4\}$, $g_3\{a_5\}$ and the query output is $\{2, 2, 1\}$.

2.3.2 Similarity Group-By Any (SGB-Any)

Given a set of tuples whose grouping attributes from a set, say P, of two dimensinal points, where $P = \{p_1, ..., p_n\}$, the SGB-Any operator $\check{\mathcal{G}}_{any}$ clusters points in P into a set

of groups, say G_m, such that, for each group $g \in G_m$, the points in g are all connected by edges to form a graph, where an edge connects two points, say p_i and p_j, in the graph iff they are within Distance ϵ from each other, i.e,. $\xi_{\delta,\epsilon}(p_i, p_j)$. More formally,

$$\check{\mathcal{G}}_{any} = \{g \mid \forall p_i, p_j \in g, (\xi_{\delta,\epsilon}(p_i, p_j) \vee (\exists\ p_{k1}, ..., p_{kn},\ \xi_{\delta\epsilon}(p_i, p_{k1}) \wedge ... \wedge \xi_{\delta\epsilon}(p_{kn}, p_j))) \wedge g\ is\ maximal\}$$

The notion of distance-to-any between elements within a group is illustrated in Figure 2.1b, where $\epsilon = 3$. All of the points (a-h) form one group. The corresponding SQL syntax of the SGB-Any operator is as follows:

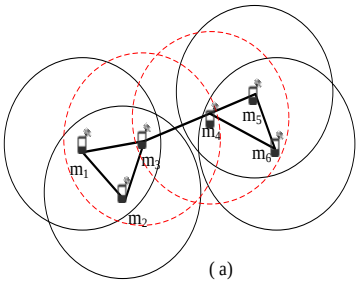
```
SELECT column, aggregate-func(column)
FROM table-name
WHERE condition
GROUP BY column DISTANCE-TO-ANY [L2 | LINF] WITHIN ϵ
```

SGB-Any uses the DISTANCE-TO-ANY predicate that applies the metric space function while evaluating the distance between adjacent points. When using the semantics for SGB-Any, the case for points overlapping multiple groups does not arise. The reason is that when an input point overlaps multiple groups, the groups merge to form one large group.

Example 2 *The following query performs the aggregate operation count on the groups formed by SGB-Any on the two-dimensional grouping attributes GPSCoor-lat and GPSCoor-long using the Euclidean distance with* $\epsilon = 3$.

```
SELECT count(*)
FROM GPSPoints
GROUP BY GPSCoor-lat and GPSCoor-long
DISTANCE-TO-ANY L2 WITHIN 3
```

Consider the example in Figure 2.2. After processing a_4*, the following groups are* $g_1\{a_1, a_2\}$ *and* $g_2\{a_3, a_4\}$*. Since Point* a_5 *is within* ϵ *from both* a_1, a_2 *in* g_1 *and* a_3, a_4 *in* g_2*, the two groups are merged into a single group. Therefore, the output of the query is*



Mobile Devices Table

MDID	Latitude	Longitude	...
m_1	lat_1	lon_1	
m_2	lat_2	lon_2	
...			

(b)

Figure 2.3.: (a) A mobile ad hoc network (MANET), (b) The mobile devices table.

$\{5\}$. *Any overlapping point will cause groups to merge and hence there is no need to add a special clause to handle overlaps.*

2.4 Applications

In this section, we present application scenarios that demonstrate the practicality and the use of the various semantics for the proposed Similarity Group-by operators.

Example 3 ***Mobile Ad hoc Network (MANET)*** *is a self-configuring wireless network of mobile devices (e.g., personal digital assistants). A mobile device in a MANET communicates directly with other devices that are within the range of the device's radio signal or indirectly with distant mobile devices using gateways (i.e., intermediate mobile devices, e.g., m_1 and m_2 in Figure 2.3a), note that the circle around each device is its signal rang. In a MANET, wireless links among nearby devices are established by broadcasting special messages. Radio signals are likely to overlap. As a result, uncareful broadcasting may result in redundant messages, contention, and collision on communication channels Consider the Mobile Devices table in Figure 2.3b that maintains the geographic locations of the mobile devices in a MANET. In the following, we give example queries that illustrate how MANETs can tremendously benefit from SGB-All and SGB-Any operators.*

Query 1 Geographic areas that encompass a MANET. *A mobile device, say m, belongs to a MANET if and only if m is within the signal range from at least one other device mobile.*

The SGB-ANY semantics identifies a connected group of mobile devices using signal range as a similarity grouping threshold.

```
SELECT ST_Polygon(Device-lat, Device-long)
FROM MobileDevices
GROUP BY Device-lat, Device-long
DISTANCE-TO-ANY L2 WITHIN SignalRange
```

Referring to the mobile devices in Figure 2.3a, the output of Query 1 returns a polygon that encompasses mobile devices m_1-m_6.

Query 2 Candidate gateway mobile devices. *A gateway represents an overlapping mobile device that connects two devices that are not within each other's signal range. The SGB-All FORM-NEW-GROUP inserts the overlapped devices into a new group. Therefore, those devices in the newly formed group are ideal gateway candidates.*

```
SELECT COUNT(*)
FROM MobileDevices
GROUP BY Device-lat , Device-long
DISTANCE-TO-ALL L2 WITHIN SignalRange
ON-OVERLAP FORM-NEW-GROUP
```

The output of Query 2 returns the number of candidate gateway mobile devices. Along the same line, identifying mobile devices that cannot serve as a gateway is equally important to a MANET. SGB-All ELIMINATE identifies mobile devices that cannot serve as a gateway by discarding the overlapping mobile devices.

Example 4 ***Location-based group recommendation in mobile social media****. Several social mobile applications, e.g., WhatsApp and Line, employ the frequent geographical location of users to form groups that members may like to join. For instance, users who reside in a common area (e.g., within a distance threshold) may share similar interests and are inclined to share news. However, members who overlap several groups may disclose information from one group to another and undermine the privacy of the overlapping groups. Query 3 demonstrates how SGB-ALL allows forming location-based groups without compromising privacy.*

Query 3 Forming private location-based groups. *The various SGB-All semantics form groups while handling ON-OVERLAP options that restrict members to join multiple groups. In Query 3, we assume that Table Users-Frequent-Location maintains the users' data, e.g., user-id and frequent location. The user-defined aggregate function List-ID returns a list that contains all the user-ids within a group.*

```
SELECT List-ID(user-id),
ST_Polygon(User-lat, User-long)
FROM Users − Frequent − Location
GROUP BY User-lat , User-long
DISTANCE-TO-ALL L2 WITHIN Threshold
[ON-OVERLAP JOIN-ANY | ELIMINATE | FORM-NEW-GROUP]
```

The output of Query 3 returns a list of user-ids for each formed group along with a polygon that encompasses the group's geographical location. The JOIN-ANY semantics recommends any one arbitrary group for overlapping members who in this case will not be able to join multiple groups. The ELIMINATE semantics drops overlapping members from recommendation, while FORM-NEW-GROUPS creates dedicated groups for overlapping members.

2.5 Efficient Algorithm for SGB Operator

In this section, we present an extensible algorithmic framework to realize similarity-based grouping using the distance-to-all semantics with the various options to handle the overlapping data among the groups.

2.5.1 Framework

Procedure 1 illustrates a generic algorithm to realize SGB-All. This generic algorithm supports the various data overlap semantics using one algorithmic framework. The algorithm breaks down the SGB-All operator into procedures that can be optimized independently. For each data point, the algorithm starts by identifying two sets (Line 2). The first

Algorithm 1: Similarity Group-By ALL Framework

Input: P: set of data points, ϵ: similarity threshold , δ: distance function , CLS: ON-OVERLAP clause, G set of existing groups

Output: Set of output groups

```
1 for each data element p_i in P do
2     (CandidateGroups, OverlapGroups) ← FindCloseGroups(p_i, G, ε, δ, CLS)
3     ProcessGroupingALL(p_i, CandidateGroups, CLS)
4     if CLS is not JOIN-ANY And sizeOf(OverlapGroups)!= 0 then
5         ProcessOverlap(p_i, OverlapGroups, CLS)
6     end
7 end
```

set, namely $CandidateGroups$, consists of groups that p_i can join. p_i can join a group, say g, in $CandidateGroups$ if the similarity predicate is true for all pairs $\langle p_i, p_i' \rangle \ \forall p_i' \in g$. The second set, namely $OverlapGroups$, includes groups that have some (but not all) of its data points satisfying the similarity predicate. A group, say g, is in $OverlapGroups$ if there exists at least two points p and q in g such that the similarity distance between p_i and p holds and the similarity distance between p_i and q does not hold. $OverlapGroups$ serves as a preprocessing step required to handle the semantics of ELIMINATE and FORM-NEW-GROUP encountered in later steps. Figure 2.4 gives four existing groups g_1-g_4 while Data-point x is being processed. In this case, $CandidateGroups$ contains $\{g_2, g_3\}$ and $OverlapGroups$ contains $\{g_1\}$.

Procedure $ProcessGroupingALL$ (Line 3 of Procedure 1) uses $CandidateGroups$ and the ON-OVERLAP clause CLS to either (i) place p_i into a new group, (ii) place p_i into existing group(s), or (iii) drop p_i from the output, in case of an ON-OVERLAP clause. Finally, Procedure $ProcessOverlap$ (Line 5) uses $OverlapGroups$ to verify whether additional processing is needed to fulfill the semantics of SGB-All.

2.5.2 Finding Candidate and Overlap Groups

In this section, we present a straightforward approach to identify $CandidateGroups$ and $OverlapGroups$. In Section 2.5.3, we propose a new two-phase filter-refine approach that utilizes a conservative check in the filter phase to efficiently identify the member groups in $CandidateGroups$. Then, in Section 2.5.4, we introduce the refine phase that is applied only if L_2 is used as the distance metric to detect the $CandidateGroups$ that falsely pass the filter step.

Procedure 2 gives the pseudocode for *Naive FindCloseGroups* that evaluates the distance-to-all similarity predicate between p_i and all the points that have been previously processed (Lines 6-15). The grouping semantics (Lines 16-20) identify how the two sets $CandidateGroups$ and $OverlapGroups$ are populated.

Processing New Points

The second step of the SGB-All Algorithm in Procedure 1 places p_i, the data point being processed, into a new group or into an existing group, or drops p_i from the output depending on the semantics of SGB-All specified in the query.

Procedure 3 (ProcessGroupingAll) proceeds as follows. First, it identifies the cases where $CandidateGroups$ is empty or consists of a single group. In these cases, p_i is inserted into a newly created group or into an existing group depending on p's distance from the existing group. Procedure $ProcessInsert$ places the data point p_i into a group. Next, the ON-OVERLAP clause CLS is consulted to determine the proper course of action. The JOIN-ANY clause arbitrates among the overlapping groups by inserting p_i into a randomly chosen group. The procedure $ProcessEliminate$ (Line 13) handles the details of processing the ELIMINATE clause. Consider the example illustrated in Figure 2.4, where $CandidateGroups$ consists of $\{g_2, g_3\}$. $ProcessEliminate$ drops Point x.

Finally, Procedure $ProcessNewGroup$ (Line 15) processes the FORM-NEW-GROUP clause. It inserts p_i into a temporary set termed S' for further processing. The SGB-All

Algorithm 2: Naive FindCloseGroupsALL

Input: p_i: data point, ϵ: similarity threshold , δ: distance function, CLS: ON-OVERLAP clause, G: set of existing groups

Output: $Candidate$, $OverlapGroups$

```
Candidate ← NULL
OverlapGroups ← NULL
for each group g_j in G do
    CandidateFlag = True
    OverlapFlag = False
    for each p_k in g_j do
        if (Distance(p_i, p_k, δ)⩽ ϵ) then
            OverlapFlag = True
        else
            CandidateFlag = False
            if CLS == JOIN-ANY then
                break
            end
        end
    end
    if CandidateFlag is True then
        insert g_j into Candidate
    else if CLS is not JOIN-ANY and CandidateFlag is False and OverlapFlag is True then
        insert g_j into OverlapGroups
    end
end
```

with FORM-NEW-GROUP option forms groups out of S' by calling SGB-All recursively until S' is empty.

Algorithm 3: ProcessGroupingALL

Input: p_i: data point, CLS: ON-OVERLAP clause, $CandidateGroups$

Output: updates $CandidateGroups$ based on CLS semantics

```
1  if sizeof(CandidateGroups)== 0 then
2      create a new group g_new
3      ProcessInsert(p_i, g_new)
4  else if sizeof(CandidateGroups) == 1 then
5      insert into existing group g_out
6      ProcessInsert(p_i, g_out)
7  else
8      switch CLS do
9          case JOIN-ANY
10             g_out ← GetRandomGroup(CandidateGroups)
11             ProcessInsert(p_i, g_out)
12         case ELIMINATE
13             ProcessEliminate(p_i, CandidateGroups)
14         case FORM-NEW-GROUP
15             ProcessNewGroup(p_i, CandidateGroups)
16     endsw
17 end
```

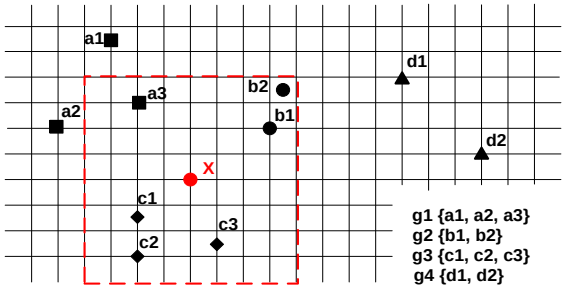


Figure 2.4.: Processing the point x using L_∞ with $\epsilon = 4$.

Handling Overlapped Points

The final step of SGB-All in Procedure 1 processes the groups in the Set $OverlapGroups$. $OverlapGroups$ consists of groups, where each group has some data points (but not all of them) that satisfy the similarity predicate with the new input point p_i. This step is required by the ELIMINATE and FORM-NEW-GROUP semantics. Procedure $ProcessOverlap$ handles the ELIMINATE semantics as follows. It iterates over $OverlapGroups$ and deletes overlapped data points. Consider the example illustrated in Figure 2.4. Set $OverlapGroups$ consists of $\{g_1\}$ with overlapped Data-Point a_3. Finally, $ProcessOverlap$ handles the FORM-NEW-GROUP semantics by inserting the overlapped data points into a temporary set termed S' and deletes these points from their original groups.

The time complexity for SGB-All according the algorithmic framework in Procedure 1 is dominated by the time complexity of $FindCloseGroups$. The time complexity of $ProcessGrouping$ and $ProcessOverlap$ (Lines 3-6) is linear in the size of $CandidateGroups$ and $OverlapGroups$. Consequently, given an input set of size n, Procedure *Naive FindCloseGroups* incurs $\binom{n}{2}$ distance computations that makes the upper-bound time complexity of SGB-All quadratic i.e., $O(n^2)$. Section 2.5.3 introduces a filter-refine paradigm to optimize over Procedure *Naive FindCloseGroups*.

2.5.3 The Bounds-Checking Approach

In this section, we introduce a group Bounds-Checking approach to optimize over Procedure *Naive FindCloseGroups*. Consider the data points of Group g illustrated in Figure 2.5a. Procedure *Naive FindCloseGroups* performs six distance computations to determine whether a new data point x can join Group g. To reduce the number of comparisons, we introduce a bounding rectangle for each Group g in conjunction with the similarity threshold ϵ so that all data points that are bounded by the rectangle satisfy the distance-to-all similarity predicate. For example, Data Element x in Figure 2.5b is located inside g's bounding rectangle. Therefore, g is a candidate group for x.

Definition 5 *Given a set of multi-dimensional points and a similarity predicate $\xi_{\delta_\infty,\epsilon}$, the* ***$\epsilon$-All Bounding Rectangle*** *$R_{\epsilon-All}$ is a bounding rectangle such that for any two points x_i and y_i bounded by $R_{\epsilon-All}$, the simiarity predicate $\xi_{\delta_\infty,\epsilon}(x_i, y_i)$ is true.*

Consider Figure 2.5c, where the bounding rectangle $R_{\epsilon-All}$ is constructed for a group that consists of a single Point a_1, where $\epsilon = 2$ and the sides of the rectangle are 2ϵ by 2ϵ centered at a_1. After inserting the second Point a_2 into g, as in Figure 2.5d, $R_{\epsilon-All}$ is shrunk to include the area where the similarity predicate is true for both Points a_1 and a_2. The invariant that $R_{\epsilon-All}$ maintains varies depending on the distance metric used. For the L_∞ distance metric, $R_{\epsilon-All}$ is updated such that if a Point, say x_i, is inside $R_{\epsilon-All}$, then x_i is guaranteed to be within Distance ϵ from all the points that form Group g. For the Euclidean distance, the invariant that $R_{\epsilon-All}$ maintains is that if a point, says x_i, is outside $R_{\epsilon-All}$, then x_i cannot belong to Group g. In this case, if x_i is inside $R_{\epsilon-All}$, it is likely that x_i is within distance ϵ from all the points inside $R_{\epsilon-All}$. Hence, for the Euclidean distance, $R_{\epsilon-All}$ is a conservative representation of the group g and serves as a filter step to save needless comparisons for points that end up being outside of the group. We illustrate in Figures 2.5c- 2.5e how to maintain these invariants when a new point joins the group. We use the case of L_∞ for illustration. When a new point x_i is inside the bounding rectange $R_{\epsilon-All}$ of Group g, then x_i is within Distance ϵ from all the points in the group, and hence will join Group g. Once x_i joins Group g, the bounds of Rectangle $R_{\epsilon-All}$ are updated to retain the truth of $R_{\epsilon-All}$'s invariant. The sides of $R_{\epsilon-All}$ will need to shrink and will be updated as illustrated in Figures 2.5d-2.5e.

Notice that deciding membership of a point into the group requires a constant number of comparisons regardless of the number of points inside Group g. Furthermore, the maintenance of the bounding rectangle of the group takes constant time for every inserted point into g. Also, notice that $R_{\epsilon-All}$ stops shrinking if its dimensions reach $\epsilon \times \epsilon$, which is a lower-bound on the size of $R_{\epsilon-All}$. Figure 2.5e gives the updated $R_{\epsilon-All}$ after Point a_3 is inserted into the group.

Procedure 4 gives the pseudocode for *Bounds-Checking FindCloseGroups*. The procedure uses the ϵ-All bounding rectangle to reduce the number of distance computa-

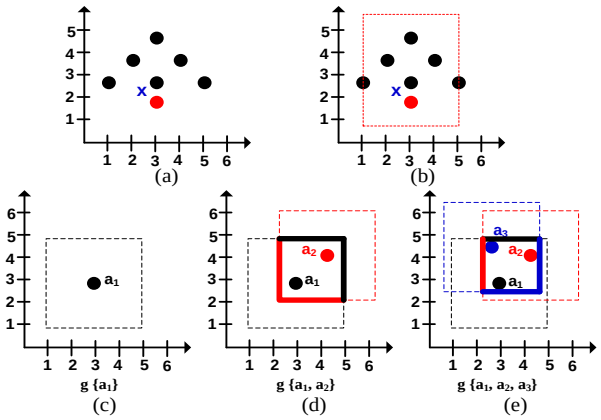


Figure 2.5.: The ϵ-All bounding rectangle approach.

tions needed to realize $FindCloseGroups$ using the L_∞ distance metric. Procedure $PointInRectangleTest$ (Line 4) uses the ϵ-All rectangle to determine in constant time whether g_j is a candidate group for the input point. Procedure $OverlapRectangleTest$ (Line 6) tests whether the ϵ-All rectangle of p_i overlaps Group g_j's bounding rectangle. In case of an overlap, all data points in g_j are inspected to verify whether the overlap is nonempty. The correctness of the ϵ-All bounding rectangle for the L_∞ distance metric follows from the fact that the rectangles are closed under intersection, i.e., the intersection of two rectangles is also a rectangle.

A major bottleneck of the bounding rectangles approach is in the need to linearly scan all existing bounding rectangles that represent the groups to identify sets $CandidateGroups$ and $OverlapGroups$, which is costly. To speedup Procedure *Bounds-Checking FindCloseGroups*, we use a spatial access method (e.g., an R-tree [26]), to index the $R_{\epsilon-All}$ bounding rectangles of the existing groups.

Procedure 5 gives the pseudocode for *Index Bounds-Checking FindCloseGroups*. The procedure performs a window query on the index $Groups\_IX$ (Line 4) to retrieve the set $GSet$ of all groups that intersect the bounding rectangle R_{p_i} for the newly inserted point p_i . Next, it iterates over $GSet$ (Lines 4-11) and executes $PointInRectangleTest$ to determine whether the inspected group belongs to either one of the two sets $CandidateGroups$ or

Algorithm 4: Bounds-Checking FindCloseGroups

Input: p_i: data point, ϵ: similarity threshold , δ: distance function, CLS: ON-OVERLAP clause, G: set of existing groups

Output: $CandidateGroups$, $OverlapGroups$

```
1  CandidateGroups ← NULL
2  OverlapGroups ← NULL
3  for each group g_j in G do
4      if PointInRectangleTest(p_i, g_j) is True then
5          insert g_j into CandidateGroups
6      else if CLS is not JOIN-ANY and OverlapRectangleTest(p_i, g_j) is True then
7          for each p_k in g_j do
8              if (Distance(p_i, p_k, δ)⩽ ϵ) then
9                  insert g_j into OverlapGroups
10                 break
11             end
12         end
13     end
14 end
```

$OverlapGroups$. Finally, the elements of $OverlapGroups$ are inspected to retrieve the subset of elements that satisfy the similarity predicate.

Refer to Figure 2.6 for illustration. An R-tree index, termed $Groups\_IX$, is used to index the bounding rectangles of the groups discovered so far. In this case, $Groups\_IX$ contains bounding rectangles for Groups g_1-g_4. Given the newly arriving Point x, a window query of the ϵ-All rectangle for x is performed on $Groups\_IX$ that returns all the intersecting rectangles; in this case, g_1, g_2, and g_3. The outcome of the query is used to construct the sets $CandidateGroups$ and $OverlapGroups$.

Algorithm 5: Index Bounds-Checking FindCloseGroups

Input: p_i: data point, ϵ: similarity threshold , δ: distance function, CLS: ON-OVERLAP clause, G: set of existing groups, $Groups\_IX$: index on G's bounding rectangles

Output: $CandidateGroups$, $OverlapGroups$

```
1  CandidateGroups ← NULL
2  OverlapGroups ← NULL
3  R_{p_i} ← CreateBoundingRectangle(p_i, ε)
4  GSet ← WindowQuery(p_i, R_{p_i}, Groups_IX)
5  for each group g_j in GSet do
6      if PointInRectangleTest(p_i, g_j) is True then
7          insert g_j into CandidateGroups
8      else if CLS is not JOIN-ANY then
9          for each p_k in g_j do
10             if (Distance(p_i, p_k, δ)⩽ ε) then
11                 insert g_j into OverlapGroups
12                 break
13             end
14         end
15     end
16 end
```

2.5.4 Handling False Positives L_2

In this section, we study the effect of using L_2 as a similarity distance function on the SGB-All operator. Refer to Figure 2.7a for illustration. In contrast to the L_∞ distance, the set of points that are exactly ϵ away from a_1 in the L_2 metric space form a circle. Inserting a_2 (Figure 2.7b) is correct using the L_∞ distance since a_2 is inside the ϵ-All rectangle of a_1's group. However, under the L_2 distance, a_2 is more than ϵ away from a_1 since a_2 lies outside a_1's ϵ-circle. As a result, all points that are inside a_1's ϵ-All group rectangle but

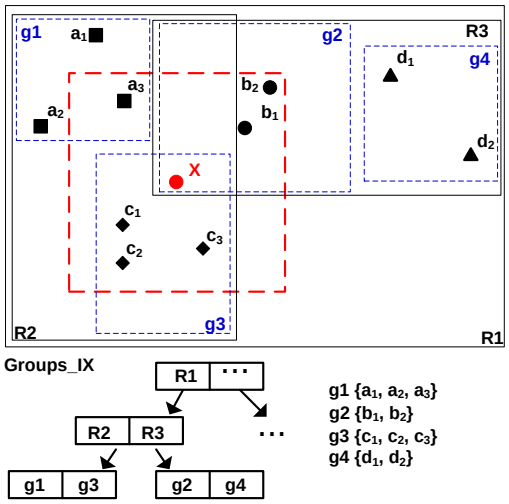


Figure 2.6.: SGB-All: Performing a window query on $Groups\_IX$ using $\epsilon = 4$ and L_∞

are outside the ϵ-circle (i.e., the grey-shaded area in Figure 2.7b) falsely pass the bounding rectangle test.

Procedure *Naive FindCloseGroups* in (Procedure 2) inspects all input data points. Therefore, the problem of false-positive points does not occur. On the other hand, the Bounds-Checking approach introduced in Procedures 4 and 5 uses the ϵ-All rectangle technique to identify the sets *CandidateGroups* and *OverlapGroups* and hence must address the issue of false-positive points for the L_2 distance metric.

We introduce a **Convex Hull Test** to refine the data points that pass the Bounds-Checking filter step. Given a group of points, a convex hull [27] is the smallest convex set of points within a group. In Figure 2.7c, the points a_1-a_5 form the convex hull set for Group g. Based on the SGB-All semantics, the diameter of the conevex hull (i.e., the two farthest points) satisfies the similarity predicate.

The *Convex Hull Test*, illustrated in Procedure 6, verifies whether a point is a false-positive. This additional test can be inserted immediately after (Line 4) in Procedure 4 or immediately after (Line 6) in Procedure 5. Consequently, any new point that lies inside a group's convex hull (e.g., Point y in Figure 2.7c) satisfies the similarity predicate. In addition, in order to verify points that are outside the convex hull (e.g., Point x in Figure

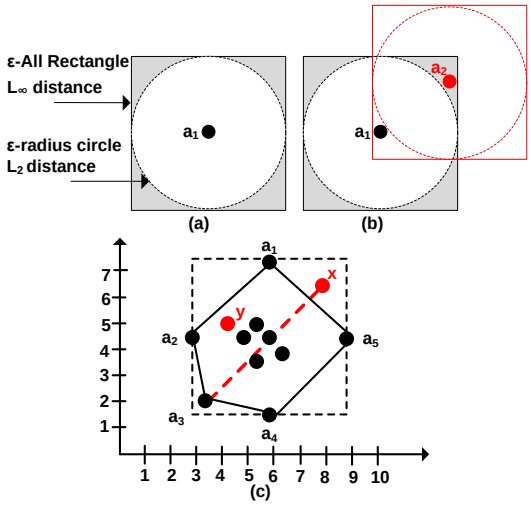


Figure 2.7.: (a) The ϵ-radius circle, (b) The problem of false positive for L_2, (c) The ϵ-convex hull

Algorithm 6: Convex Hull Test

Input: p_i: data point, g: existing group

Output: True if p_i is not false positive, False otherwise

```
1  ConvexHullSet ← getConvexHull(g)
2  if p_i inside convex hull then
3      return True
4  else
5      farthestPoint ← getMaxDistElem(ConvexHullSet, p_i)
6      if distance(farthestDistPoint, p_i) <= ε then
7          return True
8      end
9  end
10 return False
```

2.7c), it is enough to evaluate the similarity predicate between p_i and the convex hull. The correctness of the convex hull test follows from the fact that the convex hull set contains the farthest point from p_i, say p_f. Therefore, it is sufficient to evaluate the similarity predicate

Algorithm 7: Similarity Group-By ANY Framework

Input: P: set of data points, ϵ: similarity threshold, δ: distance function, $Points\_IX$: spatial index

Output: Set of groups G

```
1 for each data element p_i in P do
2     CandidateGroups ← FindCandidateGroups(p_i,Points_IX, ε, δ)
3     ProcessGroupingANY(p_i, CandidateGroups)
4 end
```

between p_i and p_f (e.g., x and Point a_3 in Figure 2.7c). Section 2.7 discusses the complexity of the convex hull approach.

2.6 Algorithms for SGB-Any

In this section, we present an algorithmic framework to realize similarity-based grouping using the distance-to-any semantics. The generic SGB-Any framework in Procedure 7 proceeds as follows. For each data point, say p_i, Procedure $FindCandidateGroups$ (Line 2) uses the distance-to-any similarity predicate to identify the set $CandidateGroups$ that consists of all the existing groups that p_i can join. In contrast to SGB-All, in the distance-to-any semantics, a point, say p_i, can join a candidate group, say g, when p_i is within a predefined similarity threshold from at least one another point in g. Procedure $ProcessGroupingANY$ (Line 3) inserts p_i into a new or an existing group.

2.6.1 Finding Candidate Groups

A Naive *FindCandidateGroups* approach similar to Procedure 2 can identify the set $CandidateGroups$. However, this solution incurs many distance computations, and brings the upper-bound time complexity of the SGB-Any framework to $O(n^2)$.

The filter-refine paradigm using an ϵ-group bounds-checking approach while applying a distance-to-any predicate (i.e., similar to Procedures 4-6) suffers from two main challenges.

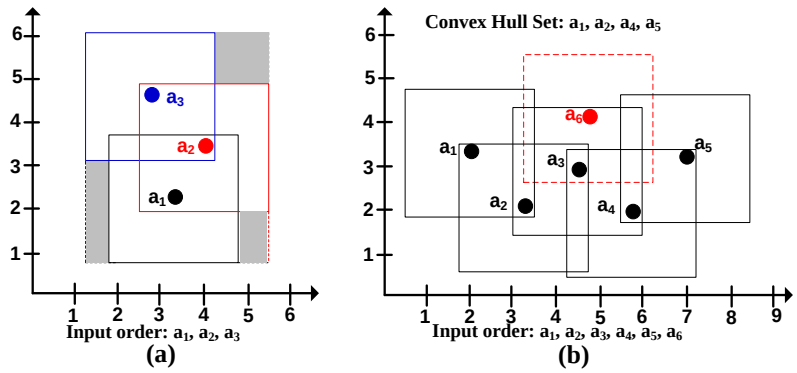


Figure 2.8.: (a) The ϵ-Any bounding rectangle, (b) The false negative problem

Consider Figure 2.8a that illustrates a group of points that satisfy the distance-to-any predicate. In Figure 2.8a, the Point a_1 is within ϵ from a_2 that in turn is within ϵ from a_3. By drawing squares of size $\epsilon \times \epsilon$ around the input point and forming a bounding rectangle that encloses all these squares results in a consecutive chain-like region and the area of false-positive progressively increases in size as we add new data points. Furthermore, the convex hull approach to test for false-positive points cannot be applied in SGB-Any as it suffers from false-negatives caused by the fact that the length of the diameter of the convex hull can actually be more than ϵ in the case of SGB-Any. Details are omitted here for brevity. Consider Figure 2.8b, After processing Point a_5, the convex hull set consists of $\{a_1, a_2, a_4, a_5\}$. Point a_6 is more than Distance ϵ from all of the points in the convex hull set. However, a_6 is within ϵ from Point a_3. Therefore, the convex hull test suffers from the problem of false-negatives caused by the fact that the length of the convex hull set diameter is more than ϵ in SGB-Any semantics.

Consequently, *FindCandidateGroups* in Procedure 8 uses an R-tree index, termed $Points\_IX$. $Points\_IX$ maintains the previously processed data points to efficiently find $CandidateGroups$. Refer to Figure 2.9 for illustration. For an incoming point, say Point x, an ϵ-rectangle (Line 2 of Procedure 8) is created to perform a window query on $Points\_IX$ to retrieve $PointsSet$ (Line 3). $PointsSet$ corresponds to the points that are within $epsilon$ from x, e.g., $\{a_3, c_1, c_2, c_3, b_1, b_2\}$. Based on the semantics of SGB-Any, $CandidateGroups$ contains the groups that cover the points in $PointsSet$. For instance,

Algorithm 8: FindCandidateGroups

Input: p_i: data point, $Points\_IX$: spatial index, δ: distance function, ϵ: similarity threshold

Output: $CandidateGroups$

1 $CandidateGroups \leftarrow NULL$

2 $R_{pi} \leftarrow$ CreateBoundingRectangle(p_i, ϵ)

3 $PointsSet \leftarrow WindowQuery(p_i, R_{pi}, Points\_IX)$

4 **if** δ *is* L_2 **then**

5 $\quad PointsSet \leftarrow VerifyPoints(Points\_IX, \delta, \epsilon)$

6 **end**

7 $CandidateGroups \leftarrow GetGroups(PointsSet)$

8 insert p_i into $Points\_IX$

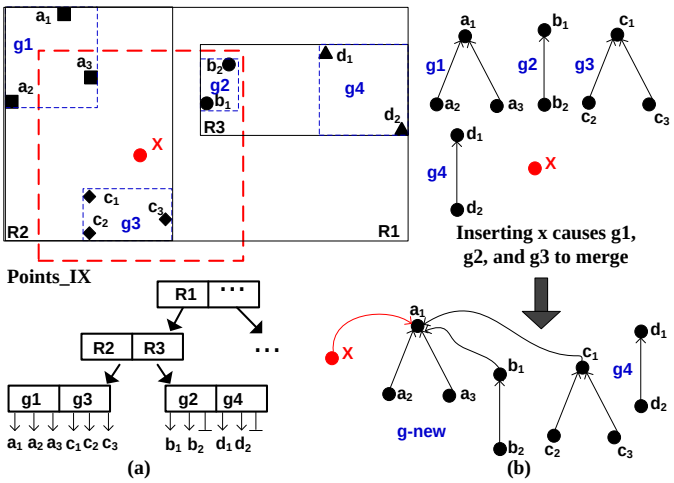


Figure 2.9.: (a) SGB-Any: Performing a window query (b) The disjoint data structure: Union-Find

point a_3 belongs to g_1, points $\{c_1$–$c_3\}$ belong to g_2, and points $\{b_1$–$b_2\}$ belong to group g_3. Hence, $CandidateGroups = \{g_1, g_2, g_3\}$. Procedure $GetGroups$ (Line 7) employs a Union-Find data structure [28] to keep track of existing, newly created, and merged groups (see Figure 2.9b) to efficiently construct $CandidateGroups$ given $PointsSet$.

Algorithm 9: ProcessGroupingANY

Input: p_i: data point, $CandidateGroups$

Output: updates $CandidateGroups$

```
1 if CandidateGroups is Empty then
2     create a new group g_new
3     ProcessInsert(p_i, g_new)
4 else if sizeof(CandidateGroups) == 1 then
5     insert into existing group g_out
6     ProcessInsert(p_i, g_out)
7 else
8     MergeGroupsInsert(CandidateGroups, p_i)
9 end
```

2.6.2 Processing New Points

Procedure 9 gives the pseudocode for *ProcessGroupingANY*. The procedure (Lines 1-6) identifies the cases when $CandidateGroups$ is empty, or when it consists of one group. In these cases, p_i is inserted into a newly created group or into an existing group. Next, it handles the case that occurs when p_i is close to more than one group. In the SGB-Any semantics, all candidate groups that p_i can join are merged into one group. Therefore, Procedure $MergeGroupsInsert$ (Line 8) handles merging candidate groups and then inserts p_i into the merged groups. Referring to Figure 2.9b, Point x overlaps groups $g_1, g_2,$ and g_3. Based on the semantics of SGB-Any, the overlapped groups g_1, g_2, and g_3 are merged into one encompassing bigger group, termed $G–new$. In this case, the root pointers of g_1, g_2 and x in the Union-Find data strucure are redirected to Point a_1.

2.7 Complexity Analysis

We analyze the runtime of SGB-All and SGB-Any. Let n, k, $|G|$, $|G_c|$, $|G_v|$ be the data cardinality, the expected number of points per group, the number of existing groups, the

size $CandidateGroups$, and the size of $OverlapGroups$, respectively, where $k \leq n$ and $|G| \leq n$ as each point can belong to only one group.

2.7.1 SGB-All

The runtime for SGB-All is output-sensitive and is influenced by several factors e.g., the ON-OVERLAP options, and the runtimes of $FindCloseGroups$ and $ProcessOverlap$. These factors vary with ϵ and with the data distribution. For instance, the number of Groups $|G|$ can vary from 1 to n depending on the value of ϵ. For example, when ϵ is very small, $|G| = n$. Next, we analyze the runtime complexity for *Bounds-Checking* and, the *on-the-fly index for Bounds-Checking* using the various ON-OVERLAP options.

SGB-All Join-Any. Refer to Procedure 4 *Bounds-Checking*. It finds the groups $CandidateGroups$ by linearly testing all existing groups (Lines 4-6) to determine if point p_i can join Group g_j. Each test takes constant time. Thus, the runtime of ON-OVERLAP JOIN-ANY is bounded by the number of groups, i.e., $O(n\ |G|)$.

Refer to Procedure 5. $Groups\_IX$ is an *on-the-fly* R-tree that indexes the bounding rectangles of all existing groups. Given a new data point, say p_i, a window query of size 2ϵ on $Groups\_IX$ finds the groups $CandidateGroups$ that p_i can join. Thus, the runtime for Procedure 5 (Line 4) is $O(log\ |G|)$ and the overall runtime of ON-OVERLAP JOIN-ANY is $O(n\ log\ |G|)$. When $|G| = n$ (the number of inputs tuples), the worst-case runtime of the *on-the-fly Index for Bounds-Checking* ON-OVERLAP JOIN-ANY is no better than $O(n\ log\ n)$. In contrast, when $|G|$ is constant, e.g., 1, the best-case runtime is $O(n)$. Finally, the average-case runtime of the *on-the-fly index for Bounds-Checking* is $O(n\ log\ |G|)$.

SGB-All Eliminate. The semantics of ON-OVERLAP ELIMINATE incurs additional ($k\ |G_v|$) time while inspecting Set $OverlapGroups$ to retrieve the subset that satisfies the similarity predicate (Lines 8-10) in Procedure 4 and (Lines 10-12) in Procedure 5). In addition, $ProcessEliminate$ (Line 13) in Procedure 3 incurs additional cost of $|G_c|$ to update the bounds of the candidates groups after removing the overlapped points. Thus, the

runtime of *Bounds-Checking* ON-OVERLAP ELIMINATE is $O(n\ (|G| + |G_c| + |G_v|\ k))$ while the runtime of *on-the-fly Index for Bounds-Checking* ON-OVERLAP ELIMINATE is $O(n\ (log\ |G| + |G_c| + |G_v|\ k))$. Naturally, $k = n/|G|$, so the runtime of *on-the-fly Index for Bounds-Checking* ON-OVERLAP ELIMINATE is $O(n\ (log\ |G| + |G_c| + n\ |G_v|/|G|))$. In the worst-case, $|G| = n$, $|G_c| = |G|$ and $|G_v|/|G| = constant$, and the corresponding runtime of *on-the-fly Index for Bounds-Checking* ON-OVERLAP ELIMINATE is $O(n^2)$. In contrast, the best-case runtime is $O(n)$ when the sizes $|G| = |G_v| = |G_c| = 1$. The average-case runtime is $O(n\ log\ |G|)$ when the sizes of $OverlapGroups$ $|G_v| \ll n$ and $CandidateGroups$ $|G_c| \ll n$.

SGB-All FORM-NEW-GROUP. Procedures $ProcessNewGroup$ and $Process\text{-}OverlapNewGroup$ insert the overlapped points into a temporary set S'. Upon finding all points in S', SGB-All recursively performs a new round of Form-NEW-GROUP while grouping the contents of S' until S' is empty. Let m be the recursion counter that is initially 0, and S'_m be the set S' at recursion stage m. Then, S'_0 is the input dataset where the size of S'_0 i.e., $|S'_0| = n$. The time cost for each round is t_m= O($|S'_m|\ O(FindClose\text{-}GroupsALL) + O(ProcessOverlap)$) that is t_m=$O(|S'_m|\ (|G^m| + |G^m_c| + |G^m_v|\ k^m)$, where $|G^m|$, $|G^m_c|$ and $|G^m_v|$ are the number of existing groups, $CandidateGroups$, and $OverlapGroups$ at each round m, respectively. Thus, the overall runtime of SGB-All FORM-NEW-GROUP is the sum of t_m from recursion depth 0 to DP, where t_m is the cost at Recursion Depth m. Then, the complexity of *Bounds-Checking* is $\sum_{m=0}^{DP} t_m$ = $\sum_{m=0}^{d} O(|S'_m|\ (|G^m| + |G^m_c| + |G^m_v|\ k^m))$. Similarly, the time complexity of the *on-the-fly index* for Bounds-Checking is $\sum_{m=0}^{d} O(S'_m\ \ (log\ |G^m| + |G^m_c| + |G^m_v|\ k_m))$. The best-case behavior of *Index Bounds-Checking* for FORM-NEW-GROUP occurs when set $OverlapGroups$ is empty and the size of $CandidateGroups$ is constant. Then, the best-case runtime is $O(n)$. In contrast, if the recursion depth is almost n, the worst-case runtime is $O(n^3)$. On average, the recursion counter $m = constant \ll n$ and $|S'_m| \ll n$, and the complexity is $O(m\ n\ log(|G|))$.

The Convex Hull Test in Section 2.5.4 forms a convex hull for each group g_j to filter out the false-positive points. The expected size of the convex hull for one group g_j is h,

where $h = log\ k$ [29], where k is the expected number points in g_j. Refer to Procedure 6. It takes $O(log\ h)$ to test if a point is inside the convex hull (Line 2). Moreover, given a point, say p_i, located outside the convex hull, it takes $O(log\ h)$ to obtain the farthest point from p_i (Line 5). Thus, for a group of points, g_j, the time to test if p_i can join g_j is $O(log\ h + log\ h)$; that is $O(log\ log\ k)$. ***ConvexHullTest*** is performed for each group that passes the $PointInRectangle$ test with $O(log\ k)$ cost (using L_∞). Thus, the computation cost to extend Procedures 4 and 5 with ***ConvexHullTest*** is $O(n\ |G|\ log\ k)$ for ***Bounds-Checking*** and $O(n\ log\ |G|\ log\ k)$ for the ***on-the-fly Index*** for Bounds-Checking. Finally, the average-case runtime of the ***on-the-fly Index*** for Bounds-Checking when using L_2 is $O(n\ log\ |G|\ log\ k)$. Notice that the actual running time is faster than the average-case because the convex hull test is executed only if a new point has passed the Group g_j's rectangle test.

2.7.2 SGB-Any

Refer to Procedure 8. For each new input point p_i, the window query returns the processed points that are within ϵ from p_i. Given a set of n points, the complexity of the window query is $O(n\ log\ n)$. Moreover, Procedures $getGroups$ and $MergeGroupsInsert$ use Union-Find to keep track of new, existing, and merged groups. The amortized runtime of Union-Find for n points is $O(m'\alpha(n))$ [28], where m' is the total operations to build new groups, $m' = |G|$, $\alpha(n)$ is a very slowly growing function, and $\alpha(n) \leq 4$. Therefore, the average case of Union-Find running time is $O(n)$, where $m' \leq n$. Hence, the average-case runtime of SGB-Any usinpg an on-the-fly index is $O(n\ log\ n) + O(n)$, that is $O(n\ log\ n)$. Also, using L_2 requires an additional step $(verifyPoints)$ to filter out the points that do not satisfy the similarity predicate in ***OverlapGroups*** (Line 7) with a cost k' per point, where k' is the expected number of points within a window query. Consequently, the runtime cost of SGB-Any using L_2 is $O(n\ log\ n + n\ k')$. k' is influenced by ϵ. Thus, the worst-case runtime when using L_2 is n^2, when $k' \approx n$. If k' is constant, the average-case runtime is

$O(n\ log\ n)$. The average-case runtime of the *on-the-fly Index* for SGB-Any is $O(n\ log\ n)$ for both L_∞ and L_2.

Table 2.1 summarizes the average-case running time of SGB-All using the proposed optimizations for the L_∞ distance metric. The *All-Pairs* algorithm corresponds to naive *FindCloseGroups* in Procedure 1. Similarly, *Bounds-Checking* and *On-the-fly Indexing* corresponds to the *Bounds-Checking* and *Index Bounds-Checking* optimizations, where $|G|$ is the number of output Groups and m is the recursion depth for the ON-OVERLAP FORM-NEW. In addition, the average-case running time of SGB-Any when using the index is $O(n\ log\ n)$. The worst-case and best-case running times, and detailed analysis are given in the Appendix.

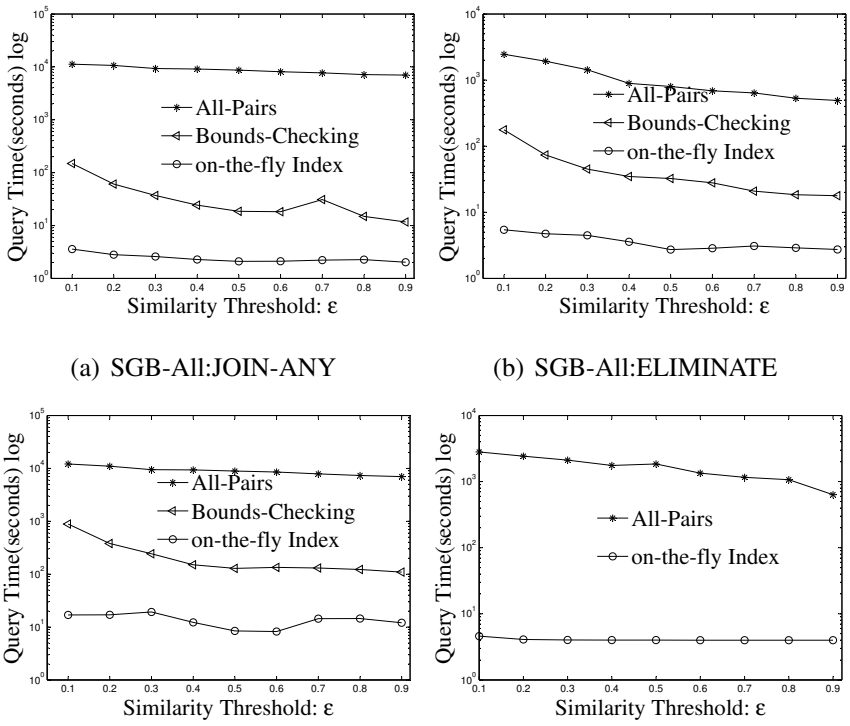
Table 2.1.: SGB-All complexity for the L_∞ distance

	JOIN-ANY	ELIMINATE	FORM-NEW-GROUP
All-Pairs	$O(n^2)$	$O(n^2)$	$O(n^3)$
Bounds-Checking	$O(n\ \|G\|)$	$O(n\ \|G\|))$	$O(m\ n\ \|G\|)$
on-the-fly Index	$O(n\ log\ \|G\|)$	$O(n\ log\ \|G\|)$	$O(m\ n\ log\ \|G\|)$

2.8 Realization and Evaluation

2.8.1 Implementation

We realize the proposed SGB operators inside PostgreSQL. In the *parser*, the grammar rules, and actions related to the "SELECT" statement syntax are updated with similarity keywords (e.g., DISTANCE-TO-ALL and DISTANCE-TO-ANY) to support the SGB query syntax. The parse and query trees are augmented with parameters that contain the similarity semantics (e.g., the threshold value and the overlap action). The *Planner and Optimizer* routines use the extended query-tree to create a similarity-aware plan-tree. In this extension, the optimizer is manipulated to choose a hash-based SGB plan.



(a) SGB-All:JOIN-ANY (b) SGB-All:ELIMINATE

(c) SGB-All:FORM-NEW-GROUP (d) SGB-ANY

Figure 2.10.: The effect of similarity threshold eps on SGB-All and SGB-ANY

Table 2.2.: Performance evaluation queries on TPC-H

Business Question: Retrieve large volume customers	
GB1	Same as the TPCH-Q18
Business Question: Retrieve customers with similar buying power, account balance	
SGB1 or SGB2	SELECT max(ab), min(tb),max(tb), average(ab), array_agg(R1.c_custkey) FROM (SELECT c_custkey, c_acctbal as ab FROM Customer WHERE c_acctbal >100) as R1 (SELECT o_custkey, sum(o_totalprice) as tp FROM Orders, Lineitem WHERE o_orderkey in (SELECT l_orderkey FROM lineitem GROUP BY Rl_orderkey having sum(l_quantity) >3000) and o_orderkey =l_orderkey and o_totalprice > 30000) as R2 WHERE R1.c_custkey=R2.o_custkey GROUP BY ab,tp **DISTANCE-ALL** WITHIN ϵ USING lone/ltwo on_overlap join-any/form-new/eliminate **or** GROUP BY ab,tp **DISTANCE-ANY** WITHIN ϵ USING lone/ltwo
Business Question: Report profit on a given line of parts (by supplier nation and year)	
GB2	Same as the TPCH-Q9
Business Question: Report profit and shipment time of parts share similar profit and shipment date	
SGB3 or SGB4	SELECT count(),sum(tprof), sum(stime) FROM (SELECT ps_partkey as partkey, sum(l_extendedprice * (1 - l_discount) - ps_supplycost *l_quantity) as tprof, sum(l_receiptdate-l_shipdate) as stime FROM lineitem, partsupp,supplier WHERE ps_partkey = l_partkey and s_suppkey=ps_suppkey GROUP BY ps_partkey) as profit GROUP BY tprof, stime **DISTANCE-ALL** WITHIN ϵ USING lone/ltwo on_overlap join-any/form-new/eliminate **or** GROUP BY tprof, stime **DISTANCE-ANY** WITHIN ϵ USING lone/ltwo
Business Question: Determines top supplier who contributed the most to the overall revenue for parts)	
GB3	Same as the TPCH-Q15
Business Question: Report supplier who contributed the similar profit and account balance	
SGB5 or SGB6	SELECT array_agg(s_suppkey), sum(r.trevenue), sum(s_acctbal) FROM (SELECT l_suppkey as suppkey, sum(l_extendedprice * (1 - l_discount)) as trevenue , sum(s_acctbal) As acctbal FROM Lineitem WHERE l_shipdate > date '[1995-01-01]' and l_shipdate < date '[1996-01-01]'+ interval '10' month GROUP BY l_suppkey)as r GROUP BY r.trevenue, s_acctbal **DISTANCE-ALL** WITHIN ϵ USING lone/ltwo on_overlap join-any/form-new/eliminate **or** GROUP BY r.trevenue, s_acctbal **DISTANCE-ANY** WITHIN ϵ USING lone/ltwo

The executor modifies the hash-based aggregate group-by routine. Typically, an aggregate operation is carried out by the incremental evaluation of the aggregate function on the processed data. However, the semantics of ON-OVERLAP ELIMINATE and ON-OVERLAP FORM-NEW-GROUP can realize final groupings only after processing the

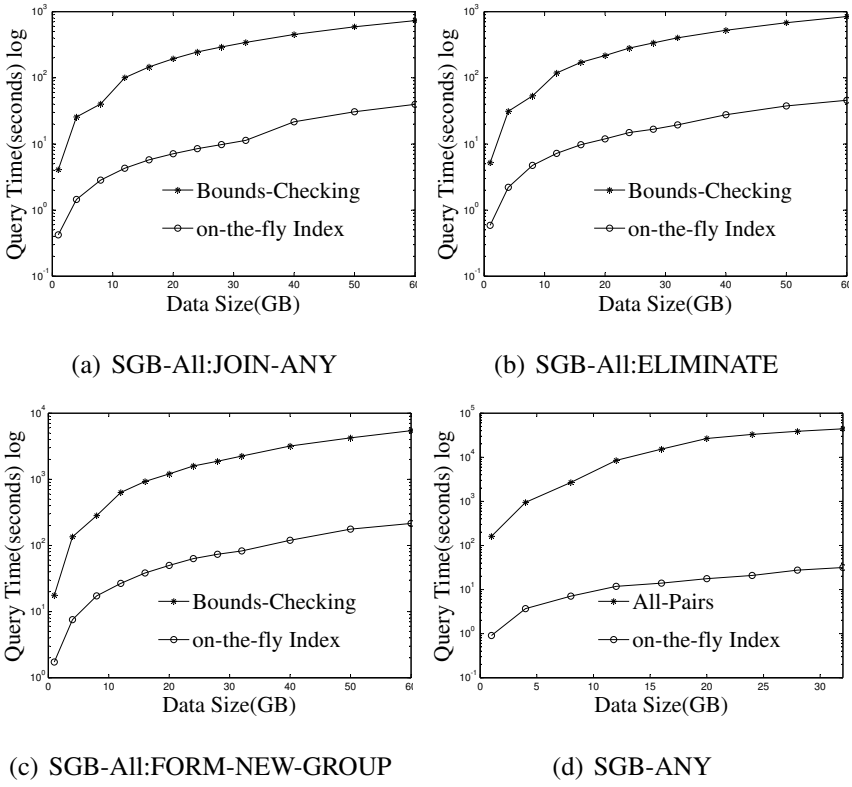
complete dataset. Therefore, the aggregate hash table keeps track of the existing groups in the following way. First, the aggregate hash table entry (AggHashEntry) is extended with a TupleStore data structure that serves as a temporary storage for the previously processed data points. Next, referring to the Bounds-Checking FindCloseGroups presented in Procedure 4, each group's bounding rectangle is mapped into an entry inside the hash directory. Bounds-Checking FindCloseGroups linearly iterates over the hash table directory to build the sets $CandidateGroups$ and $OverlapGroups$. The Index Bounds-Checking in Procedure 5 employs a spatial index to efficiently look up all existing groups a data point can join. Consequently, we extend the executor with an in-memory R-tree that efficiently indexes the existing groups' bounding rectangles.

In the implementation of FindCloseGroupsAny in Procedure 8, a spatial index is created to maintain the set of points that have been processed and assigned to groups. Moreover, we extend the executor with the Union-Find data structure Disjoint-set forest to support the operations $GetGroups$ and $MergeGroupsInsert$.

2.8.2 Datasets

The goal of the experimental study is to validate the effectiveness of the proposed SGB-All and SGB-Any operators using the optimization methods discussed in Sections 2.5 and 2.6. The datasets used in the experiments are based on the TPC-H benchmark [30], and real-world social checking data Brightite and Gowalla [31]. Table 2.2 gives the queries used for performance evaluation experiments on TPC-H data. The multi-dimensional attribute is the combination of different tables. For example, SGB queries, i.e., SGB1/SGB2, are combination of Customer and Order Table, and the number of tuples in the Customer and Order tables is $150K * SF$ and $1500K * SF$, respectively, where the scale factor SF ranges from 1 to 60. For Brightite and Gowalla data, SGB queries follow Examples 1 and 3 to cluster users into groups by check-in information i.e., latitude and longitude.

The experiments are performed on an Intel(R) Xeon (R) E5320 1.86 GHz 4-core processor with 8G memory running Linux, and using the default configuration parameters in



(a) SGB-All:JOIN-ANY (b) SGB-All:ELIMINATE

(c) SGB-All:FORM-NEW-GROUP (d) SGB-ANY

Figure 2.11.: The effect of increasing data size on the SGB-All variants and SGB-ANY

PostgreSQL. At first, we focus on the time taken by SGB and hence disregard the data preprocessing times, (e.g., the inner join and filter predicates in Query 18). Furthermore, to understand the overhead of new SGB query, we calculate SGB response time with complexity query(e.g., the SGB Query 3 to 6). In the chapter, we only give the execution time of the L_2 distance metric because the performance when using the L_∞ distance metric exhibits a similar behavior.

2.8.3 Effect of Similarity Threshold Eps

The effect of the similarity threshold ϵ on the query runtime is given in Figure 2.10 for SGB-Any and all three overlap variants of SGB-All; JOIN-ANY, ELIMINATE and

FORM-NEW-GROUP. The experiment's data size is 0.5 million records. ϵ varies from 0.1 to 0.9.

Consider an unskewed dataset, Performing SGB-All using a smaller value of ϵ (e.g., 0.1 or 0.2) forms too many output groups because the similarity predicate evaluates to true on small groups of the data. On the other hand, increasing the value of ϵ forms large groups that decreases the expected number of output groups. Thus, we observe in Figure 2.10(a), 2.10(b), 2.10(c) that the runtime of SGB-All using the various semantics decreases as the value of ϵ approaches 0.9 with the exception of ϵ of value 0.7. The slight increase in runtime in the JOIN-ANY and FORM-NEW-GROUP semantics is attributed to the distribution of the experimental of data.

The runtime and speedup in Figure 2.10(a), 2.10(b), 2.10(c) validate the advantage of the optimizations for *Bounds-Checking* and *on-the-fly Index* over *All-Pairs*. The *on-the-fly Index* approach shows two orders of magnitude speedup over *All-Pairs*, and *Bounds-Checking* approach wins one order magnitude faster than that of *All-Pairs*. The reason is that *All-Pairs* realizes similarity grouping by inspecting all pairs of data points in the input, and its runtime is bounded by the input size. In contrast, *Bounds-Checking* defines group bounds in conjunction with the similarity threshold to avoid excessive runtime while grouping. Therefore, the runtime of *Bounds-Checking* is bounded by the number of output groups. Lastly, indexing output groups using *on-the-fly Index* alleviates the effect of the number of output groups on the overall runtime and makes it steady across the various ON-OVERLAP options.

The effect of the similarity threshold ϵ on the query runtime for the SGB-Any query is given in Figure 2.10(d). The experiment illustrates that the runtime for *All-Pairs* SGB-Any decreases as the value of ϵ increases. Furthermore, the runtime of the *on-the-fly Index* method slightly changes. As a result, the speedup between the *All Pairs* and the *on-the-fly Index* methods slightly decreases. The runtime result validates that the performance of the *on-the-fly Index* method is stable as we vary the value of ϵ. The reason is that the Union-Find data structure efficiently finds and merges the candidate groups. Figure 2.10(d)

verifies that, for all values of ϵ, the runtime performance of the *on-the-fly Index* method for SGB-Any is two orders of mangitude faster than the *All-Pairs* SGB-Any.

2.8.4 Speedup

Figure 2.11(a), 2.11(b) and 2.11(c) give the performance and speedup of the *Bounds-Checking* and *on-the-fly Index* methods for large datasets with scale factor up to 60. The similarity threshold ϵ is fixed to 0.2. We do not show the results for the naive approach *All-Pairs* because its runtime increases quadratically as the data size increases. From Figure 2.11(a), 2.11(b) and 2.11(c), we observe that the runtime of the *Bounds-Checking* method increases as the number and size of groups increases. The *on-the-fly Index Bounds-Checking* method finds the sets $CandidateGroups$ and $OverlapGroups$ efficiently using the R-tree index, and the runtime of *on-the-fly Index Bounds-Checking* method increases steadily and is consistently lower than the *Bounds-Checking* methods. We observe that the speedup of the *on-the-fly Index Bounds-Checking* method is an order of magnitude better than that of *Bounds-Checking*.

Figure 2.11(d) gives the effect of varying the data size on the runtime of SGB-Any when ϵ is fixed to 0.2. The TPC-H scale factor (SF) ranges from 1 to 32. We observe that, as the data size increases, the runtime of the *All-Pairs* method increases quadratically, while the runtime of the *on-the-fly Index* method has a linear speedup. Moreover, the speedup results in the figure demonstrate that the *on-the-fly Index* method is approximately three orders of magnitude faster than *All-Pairs SGB-Any* as the data size increases.

2.8.5 Comparison with Clustering Algorithm

We compared our algorithm with outside of database approach clustering, K-means [21], DBSCAN [23], BIRCH [22]. For DBSCAN, we use the efficiently implementation from [32], which provide the spatial index to enhance the range query. The similarity threshold ϵ of DBSCAN and SGB is fixed to 0.2. Figure 2.12 shows the SGB method significantly outperformed DBSCAN, BIRCH and K-means even by 1 to 3 order of magnitude on the

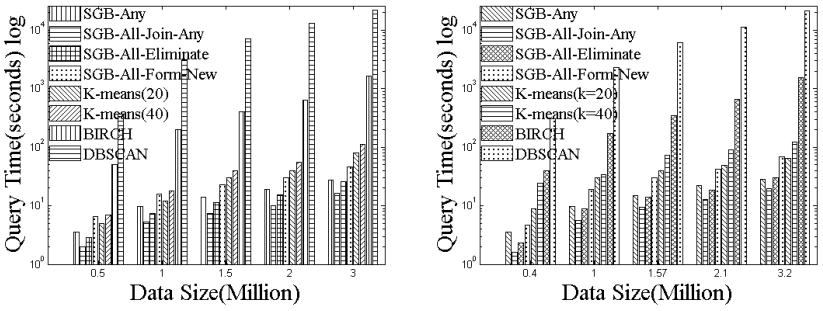
real-world data. The main reason is that cluster algorithm usually scan data more than once, this involved more computation time. On the other hand, we are building groups while scan tuples, and use group bound and spatial index to reduce the time to query data more than once.

2.8.6 Overhead of SGB

Figure 2.13 illustrates the effect of the various data sizes on the runtime of similarity-based groupings and traditional Group-By queries while varying the scale factor from 1G to 20G. The similarity threshold ϵ is fixed to 0.2. The semantics of the ON-OVERLAP clause plays a key role on the runtime of SGB-All. For instance, the JOIN-ANY variant achieves the best runtime among the SGB-All variants since it places overlapped elements into arbitrarily chosen groups. To the contrary, the FORM-NEW-GROUPS semantics incurs additional runtime cost while placing overlapped elements into new groups. The ELIMINATE semantics drops all overlapped elements causing the size of the output groups to shrink. Furthermore, we observe that the performance of the traditional Group-by operator is comparable to the SGB-All and SGB-Any variants when using the *on-the-fly Index*. The *on-the-fly Index* SGB-All ON-OVERLAP JOIN-ANY shows better performance than that of traditional Group-By, and the *on-the-fly Index* SGB-All ON-OVERLAP ELIMINATE bring additional 15 percentage overhead when compared with the traditional Group-By because the postprocessing step to eliminate the overlap Group sets. The *on-the-fly Index* SGB-All ON-OVERLAP FORM-NEW shows 40 percent overhead than the traditional Group-By. Finally, the *on-the-fly Index* SGB-Any brings an additional 20 percent overhead than when compared with the traditional Group-By.

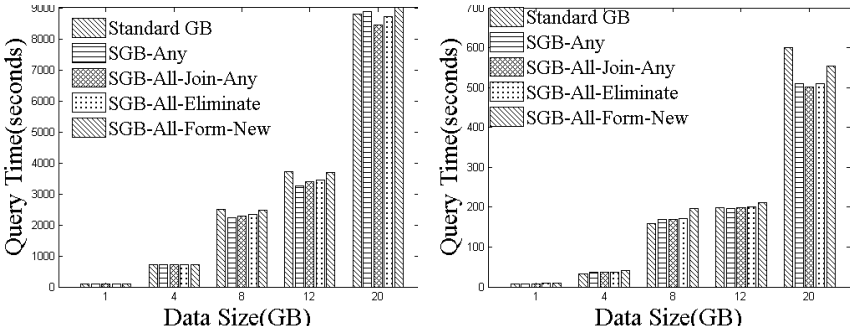
2.9 Related Work

Previous work on similarity-aware query processing addressed the theoretical foundation and query optimization issues for similarity-aware query operators [9]. [33, 34] introduce similarity algebra that extends relational algebra operations, e.g., joins and set oper-



(a) Brightkite (b) Gowalla

Figure 2.12.: Comparison with clustering methods



(a) GBY2 vs SGB3 and SGB4 (b) GBY3 vs SGB5 and SGB6

Figure 2.13.: The effect of the data size on SGB vs. SQL GBY

ations, with similarity semantics. Similarity queries and their optimization include algorithms for similarity range search and K-Nearest Neighbor (KNN) [35], similarity join, and similarity aggregates. Most of work focus on semantic and transformation rules for query optimization purpose independently from actual algorithms to realize similarity-aware operators. In contrast, our focus is on the latter.

Clustering forms groups of similar data for the purpose of learning hidden knowledge. Clustering methods and algorithms have been extensively studied in the literature, e.g., see [10], and the corresponding algorithm are widely extended for data preprocessing, e.g., see [36–38]. The main clustering methods are partitioning, hierarchical, and density-based. *K-means* [21] is a widely used partitioning algorithm that uses several iterations to refine the output clusters. Hierarchical methods build clusters either divisively (i.e., top-down) such as in *BIRCH* [23], or agglomeratively (i.e., bottom-up) such as in *CURE* [39]. Density-based methods, e.g., *DBSCAN* [22], cluster data based on local criteria, e.g., density reachability among data elements. The key differences between our proposed SGB operators and clustering are: (1) the proposed SGB operators are relational operator that are integrated in a relational query evaluation pipeline with various grouping semantics. Hence, they avoid the impedance mismatch experienced by standalone clustering and data mining packages that mandate extracting the data to be clustered out of the DBMS. (2) In contrast to standalone clustering algorithms, the SGB operators can be interleaved with other relational operators. (3) Standard relational query optimization techniques that apply to the standard relational group-by are also applicable to the SGB operators as illustrated in [9]. This is not feasible with standalone clustering algorithms. Also, improved performance can be gained by using database access methods that process multi-dimensional data.

An early work on similarity-based grouping appears in [40]. It addresses the inconsistencies and redundancy encountered while integrating information systems with dirty data. However, this work realizes similarity grouping through pairwise comparisons which incur excessive computations in the absence of a proper index. Furthermore, the introduced extensions are not integrated as first class database operators. The work in [41] focuses on overcoming the limitations of the distinct-value group-by operator and introduces the

SQL construct “Cluster By” that uses conventional clustering algorithms, e.g., *DBSCAN*, to realize similarity grouping. Cluster By addresses the impedance mismatch due to the data being outside the DBMS to perform clustering. Our SGB operators are more generic as they use a set of predicates and clauses to refine the grouping semantics, e.g., the distance relationships among the data elements that constitute the group and how inter-group overlaps are dealt with.

Several DBMSs have been extended to support similarity operations. SIREN [42] is a similarity retrieval engine that allows executing similarity queries over a relational DBMS. POSTGRESQL-IE [43] is an image handling extension of PostgreSQL to support content-based image retrieval capabilities, e.g., supporting the image data type and responding to image similarity queries. While these extensions incorporate various notions of similarity into query processing, they focus on the similarity search operation. SimDB [9] is a PostgreSQL extension that supports similarity-based queries and their optimizations. Several similarity operations, e.g., join and group-by, are implemented in as first-class database operators. However, the similarity operators in SimDB focus on one-dimensional data and do not handle multi-dimensional attributes.

2.10 Summary

In this chapter, we address the problem of similarity-based grouping over multi-dimensional data. We define new similarity grouping operators with a variety of practical and useful semantics to handle overlap. We provide an extensible algorithmic framework to efficiently implement these operators inside a relational database management system under a variety of semantic flavors. The performance of SGB-All performs up to three orders of magnitude better than the naive *All-Pairs* grouping method. Moreover, the performance of the optimized SGB-Any performs more than three orders of magnitude better than the naive approach. Finally, the performance of the proposed SGB operators is comparable to that of standard relational Group-by.

3 EFFICIENT PROCESSING OF HAMMING-DISTANCE-BASED SIMILARITY-SEARCH QUERIES OVER MAPREDUCE

3.1 Introduction

Hamming-distance search over big data plays an important role in a large variety of applications. For example, widely used search engines, such as Google, Baidu, and Bing, use Hamming-distance search in their image content-based search engines that usually index billions of images (e.g., refer to [11]). Typically, each image is modeled by a high-dimensional vector of extracted features, e.g., color histograms, texture features, and edge orientation. Then, based on the learned similarity hash function, e.g., as in [11–13], each image is converted into a binary code. Given a query image that gets modeled with the same high-dimensional vector of features, the search engine maps it into a binary code and performs a Hamming-distance search to find images whose binary codes have a Hamming distance smaller than a given threshold $\hat{h}$ from the query image. Hamming search is also widely used to detect duplicate web pages in applications, e.g., web mirroring, plagiarism, and spam detection [6]. A similarity hash function [44] is applied to map a high-dimensional vector into a binary code, then a Hamming-distance range search finds web documents that are similar to the query document.

Typically, computing the Hamming distance between two binary codes is performed by an Exclusive-Or operation (XOR, for short) that is followed by a count operation to sum up the number of ones in the XOR result. The number of ones corresponds to the number of differing bits between the two binary codes. Thus, a linear scan over the binary codes of the underlying dataset takes place to perform the XORing, the counting, and the ranking to retrieve the objects within a certain range of t_q (i.e., the ones within the predefined Hamming distance threshold $\hat{h}$). Due to the linear scan, this approach is slow. When joining two tables via a Hamming distance predicate, the linear scan approach induces a

quadratic cost to evaluate the join. An efficient indexing of the binary codes is called for to perform the Hamming range query and avoid a complete scan over the underlying dataset, while remaining low on memory usage.

The Hamming distance problem [45, 46] is first studied for small distance thresholds, i.e., $\hat{h} = 1$. An algorithm proposed by Manku et al. [6] uses multiple hash tables to enhance query speed. However, duplicating the hash entries multiple times for the entire datasets is expensive and performance tends to degrade as a linear scan over tuples within a bucket is required. HEngine [47] extends Manku's algorithm to improve the query's speed with less memory. However, HEngine is sensitive to the Hamming distance threshold $\hat{h}$, and it needs to generate one-bit differing binary code with each query, then carry out several binary searches over sorted hash tables. Recently, MapReduce as a reliable distributed computing model has been adopted for handling a variety of similarity queries, e.g., [6, 48–51]. Existing techniques for Hamming-distance queries cannot be easily extended for MapReduce. The reason is that most of the existing techniques use centralized multiple hash-table indexes. Because MapReduce needs to write intermediate data on disk when shuffling data between the mappers and the reducers, rearranging multiple indexes and multiple versions of the same data can be quite inefficient.

In this chapter, we focus on two variants of the Hamming distance query problem, namely Hamming-distance-based select and Hamming-distance-based join (for short, Hamming-select and Hamming-join, respectively). We propose a new index, termed the HA-Index, that is designed to reduce redundant and duplicate distance computations during the Hamming-distance search. The HA-Index assumes that the underlying datasets are preprocessed; data is mapped from the high-dimensional space into one-dimensional binary codes that are fixed-length strings of 0's and 1's. Then, the binary codes are sorted using the Gray ordering [24]. Sorting the binary codes in this way helps group together multiple binary codes that share a common substring or non-contiguous yet similar sequences of bits. By computing the distances between the query binary code and similar substrings, many redundant distance computations can be avoided.

Table 3.1.: Symbols and their definitions

Symbol	**Definition**
$\mathbb{R}^d$	d-dimensional vector space
n, $\|R\|$	Number of tuples in dataset R
m, $\|S\|$	Number of tuples in dataset S
t_q	Query tuple
k	The required number of nearest neighbors
$\|\|t_i, t_j\|\|_h$	Hamming distance between tuples t_i and t_j
H	Similarity Hash function
U_i	Binary code for tuple t_i
$L = \|U\|$	Length of the binary code U
l_i	The ith bit in the binary code
$\hat{h}$	Hamming distance threshold
$\hat{h}$-select(t_q, S)	Hamming distance select for tuple t_q and datasets S
$\hat{h}$-join(R, S)	Hamming distance join between datasets R and S
N	Number of data partitions

The contributions of this chapter are as follows.

- Based on properties of binary codes, we introduce two approaches to improve the performance of Hamming-select and Hamming-join. The first approach uses a simple Radix-tree index from the literature. The second approach is based on the HA-Index with both a static and a dynamic version. We also introduce the maintenance operations, i.e., build, insert, update, and search operations, for the dynamic HA-Index.

- For Hamming-joins over large and skewed data, we propose an efficient data partitioning technique for balancing data computations among servers, and introduce a distributed version of the HA-Index to reduce data shuffling inside MapReduce.

- We conduct an extensive experimental study using real datasets and demonstrate that the HA-Index (i) enhances the performance of Hamming-select and Hamming-join by two orders of magnitude over state-of-the-art techniques, and (ii) saves memory usage by more than one order of magnitude. We also evaluate how the proposed index improves approximate algorithms for kNN-select and kNN-join operations.

The rest of this chapter proceeds as follows. Section 3.5 discusses related work. Section 3.2 presents the problem definition. Section 3.3 introduces the centralized-server approach for approximate Hamming-select and Hamming-join. Section 3.4 introduces the distributed version of the HA-Index using MapReduce and explains how Hamming-select and Hamming-join can be performed in MapReduce. Section 3.6 presents and discusses the experimental results. Finally, Section 3.7 contains concluding remarks.

3.2 Preliminaries

3.2.1 Hamming-distance-based Similarity Operations

We assume that data tuples represent points in a d-dimensional metric space, say $\mathbb{R}^d$. Given two data tuples, say t_i and t_j, let $||t_i, t_j||$ be the distance between t_i and t_j in $\mathbb{R}^d$. The Hamming distance between t_i and t_j, denoted by $||t_i, t_j||_h$, helps in retrieving the tuples in a dataset that are within some threshold from an input tuple, either t_i or t_j in this case. Table 3.1 summarizes the symbols used in this chapter.

Definition 3.2.1 *Hamming-distance-based Similarity Select [6] (referred to as Hamming-select, for short): Given a query tuple, say t_q, and a dataset, say S, with its corresponding collection of binary codes, denoted by U_S, and an integer, say $\hat{h}$, that represents the similarity threshold for the Hamming distance, Hamming-select identifies a subset from S, denoted by $\hat{h}$-select(t_q, S) for short, where $\forall o \in \hat{h}$-select$(t_q, S), ||o, t_q||_h \leq \hat{h}$.*

Similarly, we define the Hamming-distance-based similarity join as follows.

Definition 3.2.2 *Hamming-distance-based Similarity Join (referred to as Hamming-join, for short): Given two collections of binary codes, say U_R and U_S, that correspond to two datasets, say R and S, respectively, and an integer, say $\hat{h}$, that represents the similarity threshold for the Hamming distance, Hamming-join identifies the set $\hat{h}$-join(R, S) of tuple pairs such that $(t_i, t_j) \in \hat{h}$-join(R, S) iff $t_i \in R$ and $t_j \in S$ and $||t_i, t_j||_h \leq \hat{h}$.*[1]

Example 5 *Consider the set of binary codes given in Table 3.2.1 and a Hamming distance threshold $\hat{h} = 3$. The query tuple t_q has a binary code "101100010". The output of the Hamming-distance-based similarity select is $\{t_0, t_3, t_4, t_6\}$. Using the same Hamming distance threshold $\hat{h}$, the output of the Hamming-distance-based similarity join for the datasets in Tables 3.2.1 and 3.2.1 is $\{(r_0, t_0), (r_0, t_3), (r_0, t_4), (r_0, t_6)\}$,$\{(r_1, t_0), (r_1, t_3), (r_1, t_4), (r_1, t_6)\}$, $\{(r_2, t_3)\}$.*

From the example above, one can produce the output set by simply scanning the table one tuple at a time, performing Hamming distance calculation via the XOR operation, and reporting the tuple as an output if the computed Hamming distance is smaller than or equal to $\hat{h}$. If $|S| = n$, then the cost of computing Hamming-select consists of $O(n)$ tuple reads and $O(n)$ Hamming-distance computations. Similarly, the cost of computing Hamming-join between the two datasets R and S, where $|R| = m$ and $|S| = n$ respectively, with a nested-loop join algorithm, consists of $O(mn)$ tuple reads and $O(mn)$ Hamming-distance computations. The focus of this chapter is to develop a Hamming-distance-based tree index to reduce the above costs.

[1]Different from the kNN-join, $\hat{h}$-join for datasets R and S is symmetric, i.e., $\hat{h}$-join(R, S)= $\hat{h}$-join(S, R).

Table 3.2.: Table S

tuple	**binary** U
t_0	001 001 010
t_1	001 011 101
t_2	011 001 100
t_3	101 001 010
t_4	101 110 110
t_5	101 011 101
t_6	101 101 010
t_7	111 001 100

Table 3.3.: Table R

tuple	**binary** U
r_0	101 100 010
r_1	101 010 010
r_2	110 000 010

3.3 Hamming-select Algorithms

In this section, we first introduce the basic concept and principles of binary hash codes, and illustrate the Radix-Tree-based approach. We then introduce two variants of our proposed HA-Index, namely the static and dynamic HA-indexes along with their associated algorithms.

3.3.1 Properties of Binary Codes

Definition 3.3.1 *A binary code $\hat{U}$ is said to be a fixed-length substring ($FLSS$) of another binary code U if $|U| = |\hat{U}|$ and there exist i and j, $1 \leq i$, $i + j \leq |U|$ such that $\forall i, i \leq v \leq$*

$i + j$, and $U[v]$ = $\hat{U}[v]$. Thus, only the bits between i and $i + j$ are the same and all the remaining can be any combination of 0*s and* 1*s.*

For example, consider Tuple t_0 in Table 3.2.1. Let $\cdot$ denote a 0 or a 1. Based on the above definition, $\hat{U}$="$\cdots\cdot$**0101** $\cdot$ " is one $FLSS$ of t_0's binary code "0011**0101**0". Alternatively, $\hat{V}$ ="$101\cdots\cdots$" is not an $FLSS$ of t_0's binary code.

Definition 3.3.2 *A binary code, $\hat{U}$, is the fixed-length SubSequence ($FLSSeq$, for short) of a binary code U if there exists a strictly increasing sequence of indices of U such that $\forall j \in \{1, 2, \ldots, k'\}$, we have $U[j] = \hat{U}[j]$ and $|U| = |\hat{U}|$.*

For example, $\hat{U}$="$\cdots$**0**$\cdot$**1**$\cdot$**1**$\cdot$" is one possible $FLSSeq$ of t_0's binary code "001**0**01**0**1**0**" in Table 3.2.1. Thus, $\hat{U}$ belongs to Set $FLSSeq$ of Tuple t_0. To compute the Hamming distance between an $FLSSeq$ and a query binary code, we only count the bit difference in the corresponding effective bit positions. For instance, if one $FLSSeq$ is $\hat{U}$="$\cdots$**0**$\cdot$**1**$\cdot$**1**$\cdot$" and the query binary code is U="001**0**01**0**1**0**", the Hamming distance $||\hat{U}, U||_h$=2.

Proposition 3.3.1 ***Hamming Downward Closure Property*** *A binary code $U \in \hat{h}$-select(t_q, S) iff each $FLSS$ of U, say U_{FLSS}, (each $FLSSeq$ of U, say U_{FLSSeq}, respectively) meets the condition $||t_q, U_{FLSS}||_h \leq \hat{h}$ ($||t_q, U_{FLSSeq}||_h \leq \hat{h}$, respectively).*

We omit the proof for simplicity. Instead, we illustrate the above proposition using the following example.

Example 6 *Refer to the Hamming-distance query in Example 5 and Table 3.2.1 and Table 3.2.1. Suppose that the Hamming-distance threshold $\hat{h} = 2$. Consider the following example cases:*

- *Case 1: Given a query binary code t_q = "110010010", since one FLSS, U_{FLSS} =* "$001\cdots\cdots$"*, is the binary code of an $FLSS$ for both t_0 and t_1 and $||U_{FLSS}, t_q||_h \geq 3$, then neither t_0 nor t_1 can belong to $\hat{h}$-select(t_q, S).*

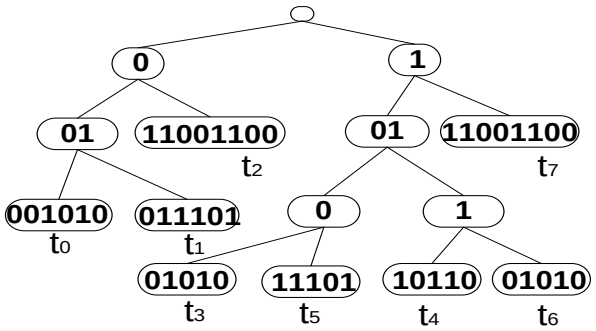


Figure 3.1.: Radix tree

- *Case 2: Given a query binary code t_q = "110110010", the binary code* "$\cdot$11001100" *is an FLSS (U_{FLSS}) for both t_2 and t_7, $||U_{FLSS}, t_q||_h \geq 3$, thus, neither t_2 nor t_7 can belong to $\hat{h}$-select(t_q, S).*

- *Case 3: Given a query binary code t_q = "110100010", the binary code* "1010$\cdot$1$\cdots$" *is an $FLSSeq$ for both t_3 and t_5, $||U_{FLSSeq}, t_q||_h \geq 3$, therefore, neither t_3 nor t_5 can belong to $\hat{h}$-select(t_q, S).*

3.3.2 Radix-Tree-Based Approach

The idea behind using a Radix-Tree index (also termed the PATRICIA trie) [52] is to merge the XOR operations for various binary codes if they happen to share $FLSS$s, e.g., similar to Case 1 of the example above. One XOR operation on a common $FLSS$ can be used to verify all participant tuples in this $FLSS$. Thus, we can build a prefix tree out of the binary codes. Based on the above closure property (Proposition 3.3.1), we can compute the Hamming distance with prefixes of the Radix-Tree from the root to find qualifying binary codes in a top-down fashion.

Example 7 *Figure 3.1 gives the corresponding Radix-Tree for the binary codes in Table 3.2.1. From the Radix-Tree, Tuples t_0 and t_1 in Table 3.2.1 share the same FLSS U_{FLSS} =* "001$\cdot\cdot\cdot\cdot\cdot$". *Given the query binary code t_q =* "110010110" *and a Hamming-*

distance query threshold $\hat{h} = 2$, both Tuples t_0 and t_1 can be discarded without computing the whole Hamming distance for all binary positions, because the Hamming distance from U_{FLSS} with the first three bits of t_q is bigger than the predefined threshold $\hat{h}$. Thus, processing the Hamming-distance-based select can stop early at the upper level of the Radix-Tree.

Notice that although useful in the above example, the Radix-Tree-based approach has several disadvantages, mainly due to its prefix-sensitiveness. For example, Tuples t_2 and t_7 in Figure 3.1 are split into two branches in the Radix-Tree, although only the first bit in the two tuples is different while all their remaining bits are the same. Thus, the search path would go to different branches of the tree and redundant computations in these two branches cannot be avoided. In the worst case, if the binary codes in the Radix-Tree do not share common prefixes, then searching from the root will bring the computation cost as bad as $O(2^L)$, because it would go through every branch of the Radix-Tree. As a result, we propose the HA-Index to address the prefix-sensitivity of the Radix-Tree-based approach.

3.3.3 Static HA-Index

The idea behind the Static HA-Index is to share the common substrings, i.e., the maximal $FLSS$s, in contrast to sharing the common prefixes for the binary codes of the underlying dataset. Thus, redundant Hamming distance computations can be avoided. Recall Case 2 of Example 6, the $FLSS$ for t_2 and t_7 is "$\cdot\, 01101010$". For the Radix-Tree-based approach in Figure 3.1, searching for the qualifying tuples would proceed to different paths, which introduces redundant computations. Thus, if we are able to realize an index that shares the common $FLSS$s, we would be able to avoid redundant and unnecessary Hamming-distance computations.

Static bit segmentation: We segment the binary codes into fixed-length contiguous substrings (called fixed-length segments). For instance, assuming that each segment is of Size 3, the binary code for tuple t_2 is divided into three segments, "011", "001" and "100". The path along these segments can be traced via an undirected path. For example, the path that corresponds to tuple t_2 is illustrated in Figure 3.2 where it connects Nodes N_2 to N_{11}

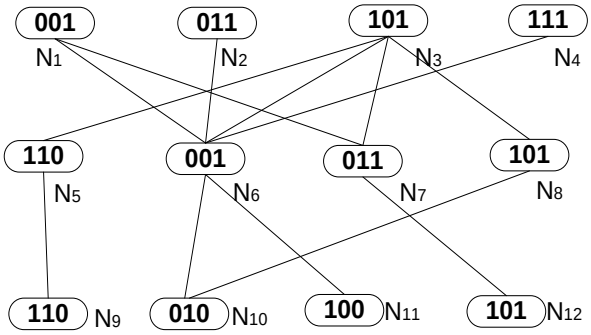


Figure 3.2.: Static HA-Index

via Intermediate Node N_6. Meanwhile, the path of Tuple t_7 includes Nodes N_4, N_6 and N_{11}. Thus, Tuples t_2 and t_7 can share the same vertex nodes N_6 and N_{11}. While traversing the index, the Hamming-distance computation for Nodes N_6 and N_{11} will be performed only once. In the next section, we demonstrate how the Static HA-Index can be used to evaluate both the Hamming-select and Hamming-join operations.

The static HA-Index has several limitations though. Both the height and the length of the paths in the Static HA-index are sensitive to the segment size. Because the segment sizes are fixed, it is possible to miss common bit substrings that do not align to segment boundaries. Also, both the Radix-Tree and the static HA-Index optimize for the $FLSS$s of the binary codes. An index that would support $FLSSeq$s, in contrast to just the $FLSS$s (recall that the $FLSS$s are subsets of the $FLSSeq$s), would allow for more shared distance computations and hence additional savings. Consider Case 3 of Example 6. Both the Radix-Tree and Static-HA-Index approaches fail to capture the common $FLSSeq$ between t_3 and t_5. In the next section, we introduce the Dynamic HA-Index to address these limitations.

3.3.4 Dynamic HA-Index

Definition 3.3.3 *Gray Order: is an ordering of the binary codes such that consecutive binary codes differ only by one bit, i.e., the Hamming distance between two consecutive binary codes that are sorted according to the Gray order is equal one [24].*

Proposition 3.3.2 ***Gray Order and Clustering****: When the binary codes are ordered based on the Gray order, data tuples are naturally clustered [53],i.e., the Hamming distance between consecutive ordered binary codes is small as the consecutively ordered binary codes share common $FLSSeq$s.*

For instance, the data tuples in Table 3.2.1 can be ordered based on the Gray order of their corresponding binary codes in descending order, and the resulting sorted order is $\{t_0, t_1, t_2, t_7, t_4, t_6, t_3, t_5\}$. Observe that the sorted binary codes provide two important

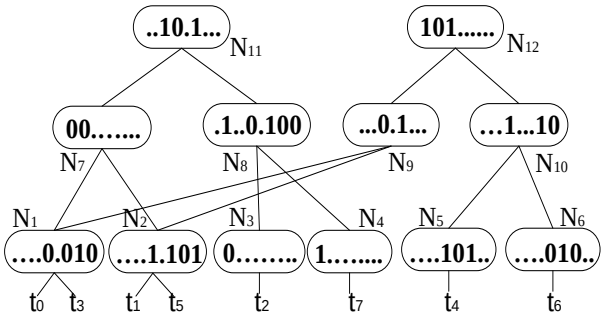


Figure 3.3.: Dynamic HA-Index

properties, namely the downward closure and the clustering properties, that facilitate efficient Hamming-distance-based query processing. Thus, our aim is to realize an index structure that preserves and leverages these properties. The Dynamic HA-Index will strategically divide the binary codes into segments (i.e., sequences of data points that are close in their binary values according to the Gray order). As such, the clustering property is preserved to ensure that nodes with similar $FLSSeq$s are close to each other in the index. For example, Tuples t_2 and t_7 are ordered next to each other, and these properties can overcome the prefix-sensitivity of the Radix-Tree-based approach.

In the Dynamic HA-Index, the leaf nodes store data tuples while the non-leaf nodes store the $FLSSeq$s of the children nodes. Refer to Figure 3.3 for an illustration. Internal node N_1 represents the $FLSSeq =$ "$\cdots 0 \cdot 010$" of Tuples t_0 and t_3. Internal node N_2 represents the $FLSSeq =$ "$\cdots 1 \cdot 101$" that is common to both Tuples t_1 and t_5. Furthermore, Internal Node N_7 represents the $FLSSeq$ for Nodes N_1 and N_2. Notice, all the descendants of an HA-Index node can be safely discarded from further Hamming-distance computations if the node's corresponding $FLSSeq$ does not qualify the Hamming-distance threshold, thereby reducing computation overheads.

3.3.5 Dynamic HA-Index Manipulation

The primary objective of all the Dynamic HA-Index manipulation algorithms, including build, delete, and insert, is to maintain the $FLSSeq$ properties of the index while keeping the size of the index reasonably small.

Bulkloading builds the Dynamic HA-Index in a bottom-up fashion. It has two steps. The first step sorts all the data tuples according to the Gray order of their nondecreasing binary codes. The second step scans these tuples sequentially using a sliding window with w slots to form index nodes. Algorithm 10 illustrates the pseudo-code to build the Dynamic HA-Index. A queue is initialized to store the temporary nodes from the window (Line 2). From the tuples within a window, Function `extractFLSSeq` extracts the maximal $FLSSeq$s from the tuples' binary codes to form new parent nodes (Line 5), and denotes

Algorithm 10: H-Build

Input: T: Set of data points, w: Window, md: Depth of HA-index, s: Sliding window size

Output: *HA*:HA-Index for dataset T

```
1  Sort T based on the non-decreasing Gray order of the tuples' binary codes;
2  q: Queue;
3  for each data element t_i of T inside Window w do
4      var n, n̂: Node;
5      n, n̂ ← extractFLSSeq(t_i, ··· , t_{i+w}); // n, the parent node of n̂
6      if n̂ is new then
7          insert n̂ into the current level of the HA-Index.
8      end
9      else
10         update n̂'s frequency
11     end
12     if n is not empty then
13         q.enqueue(n);
14     end
15     else
16         put Tuple t_i inside Window w into the top level of the HA-Index;
17     end
18     w← w+s; //sliding the window
19 end
20 var d:0, begin:0, end:q.size;
21 while q is not empty and d ≤ md do
22     // Process similar to Lines 4-18
23     // Use two pointers for q to record the HA-Index depth d
24 end
```

the new binary code of the child node. Then, the new temporal node is inserted into the queue (Line 7). For instance, Tuples t_0 and t_1 share the same $FLSSeq =$ "$0010 \cdot 1 \cdots$".

Thus, this $FLSSeq$'s corresponding new node is formed and is inserted into the queue. To save memory storage, Function `extractFLSSeq` captures the binary code of t_0 as "$\cdots\cdot 0 \cdot 010$". Therefore, the non-leaf nodes with the same $FLSSeq$ are consolidated into one node. Hence, Tuple t_3 would be denoted with the same binary code as that of t_0, and would share the same binary codes. Notice that we record the frequency of each node (Line 6-11). For example, Node N_1 represents the binary code for t_0 and t_3. Thus, the frequency for N_1 is 2. If tuples inside the window do not share any $FLSSeq$ among each other, these tuples are linked to the top level of the HA-Index (Line 16). The window continues to slide until all the data points are scanned in the first round. Lines 21-24 merge the internal nodes as Lines 4-18 and we can use two pointers `begin` and `end` for the queue to indicate the depth. The building process continues until the desired depth is reached.

In addition, more than one leaf node can be linked to the same internal node, e.g., Tuples t_1 and t_5 are linked to Internal Node N_1 in Figure 3.3. Thus, we build a hash table for the bottom node, e.g., N_1, where the key is the leaf node's binary codes, and value is the tuple's ID. Naturally, if users only want to learn the qualifying binary codes, then there is no need to keep the leaf nodes of The HA-Index. An HA-Index without leaf nodes could save the overhead of building hash tables, and can be used in MapReduce Hamming-join as in Section 3.4.

Deletion removes a tuple with its corresponding binary code from a Dynamic HA-Index. Algorithm 11 gives the corresponding process. First, a leaf node that contains the tuple to be deleted is located by depth-first search using the tuple's binary code as the search key. One stack is used to denote the unexplored paths. Function `bitmatch` tests whether one binary code is the $FLSS$ or $FLSSeq$ of the deleted tuple (Lines 3 and 14). Then, the tuple is removed from the HA-Index. After deletion, the frequency of the corresponding node needs to be decremented (Lines 5 and 16). If one node contains 0 or less entries, it is removed.

Inserting a new data tuple into a Dynamic HA-Index is similar to the deletion process. Insertion uses a depth-first search to locate the corresponding leaf node, then the search process looks for the leaf node that shares the maximal $FLSSeq$ with the newly inserted

data tuple. If no such leaf node is found, we put the newly inserted data tuple into a temporary buffer. When the buffer reaches a predefined maximum size, a process similar to H-Build is invoked to append these newly inserted tuples into the existing HA-Index. We omit these details here for brevity as they are similar to Algorithms H-Delete and H-Build.

3.3.6 HA-Index Query Processing

With the dataset organized in an HA-Index, H-Search traverses the index to visit the relevant index nodes in a breadth-first order with a queue to keep track of the unexplored qualifying paths that match the query's binary code. Algorithm 12 gives the pseudocode for H-Search. Initially, H-Search fetches the index nodes/data points from the top level of the HA-Index (Lines 2-6). If the Hamming distance between the query tuple and the pattern of the corresponding node is smaller than the threshold $\hat{h}$, then the node is inserted into the queue. For the non-top level nodes, in each round, the binary code of a node is examined against the query binary code by invoking a Hamming-distance computation. If its corresponding Hamming distance is smaller than the threshold (Line 12), the node is further explored (Lines 13-17). When a leaf node of the HA-Index is reached, the qualified data tuples are collected and are inserted into ret (Line 23-25). The algorithm terminates when all the entries from the qualifying nodes are examined.

To illustrate the H-Search Algorithm, consider the tuples in Table 3.2.1. Figure 3.3 gives the corresponding HA-Index. The execution trace is given in Table 3.4, where query binary code t_q = "010001011" is searching the dataset in Table 3.2.1. Suppose that the query binary code is t_q = "010001011" and the Hamming-distance threshold is 3. Initially, the Hamming distance between t_q and the top-level entries, i.e., $||N_{11}, t_q||_h = 1$ and $||N_{12}, t_q||_h = 3$, where both are no bigger than 3. Thus, Nodes N_{11} and N_{12} are pushed into the queue, and ret is still empty. Next, the children nodes of N_{11}, i.e., Nodes N_7 and N_8 are visited. The Hamming distances $||N_7, t_q||_h = 1$ and $||N_8, t_q||_h = 4$ are computed. As a result, the corresponding qualifying binary codes for Nodes N_{11} and N_7 are combined,

Algorithm 11: H-Delete

Input: t_q: Deleted query tuple,*HA*: HA-Index for queried dataset

```
1  s: Stack;
2  for each top level node n_i in HA do
3      if bitmatch(t_q, n_i) then
4          s.push(n_i);
5          n_i.frequency ← n_i.frequency-1 ;
6          remove n_i from HA if n_i.frequency is 0 ;
7      end
8  end
9  while s is not empty do
10     var n: Node;
11     s.pop(n);
12     if n is a non-leaf node then
13         for all child nodes c of n do
14             if bitmatch(t_q, n_i) then
15                 s.push(n_i);
16                 n_i.frequency ← n_i.frequency-1 ;
17                 remove n_i from HA if n_i.frequency is 0 ;
18             end
19         end
20     end
21     else
22         break;
23     end
24 end
```

which results in the pattern "$0010 \cdot 1 \cdots$". Thus, $[N_7, N_{11}]$ is put into the queue. But Node N_8 is discarded from the qualifying candidates because the path $N_{11} \rightarrow N_8$ has a combined Hamming distance $||N_{11}, t_q||_h + ||N_8, t_q||_h > 3$. Then, N_{12} is explored and its

children nodes(e.g.,N_9 and N_{10}) are visited. According to the Hamming-distance closure properties, $[N_9, N_{12}]$ is inserted into the queue as well, while N_{10} is discarded. The H-Search process continues until the queue is empty as shown in Table 3.4. Finally, Tuple t_0 is reported as one output tuple qualifying the query. Notice that each node maintains a *visited* flag to indicate whether the node has already been visited or not. This helps avoid redundant Hamming-distance computations. For example, Nodes N_1 and N_2 are already visited. Therefore, we do not need to compute the Hamming distance for both nodes again, and hence avoid unnecessary distance computation overhead. In addition, Algorithm H-Search for the dynamic HA-Index can be applied to the static HA-Index, and thus is not repeated in the chapter.

3.3.7 Analysis

Example 8 *Assume that we have eight tuples* $t_0 =$ "000", $t_1 =$ "001", $t_2 =$ "010", $\cdots$, *and* $t_7 =$ "111", *where all binary codes are distinct. At most 3 bits are needed to represent all the tuples, i.e., the length L of the hash values is 3. According to the H-Build process with Window Size of 2, the output HA-Index is illustrated in Figure 3.4.*

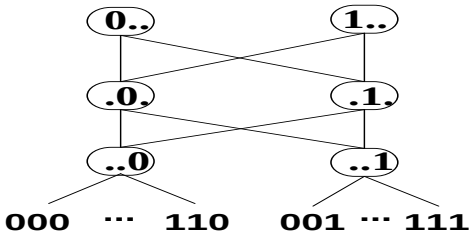


Figure 3.4.: Full binary codes and the corresponding HA-Index

Observe that the number of internal nodes of this HA-index is 6, and the number of edges is 8. Based on the breadth-first-search strategy of the H-Search algorithm, the worst search cost is bounded by the number of internal nodes and the number of edges, denoted by $|V|$ and $|E|$, respectively. Refer to Figure 3.4 for illustration. The search cost is at worst 14. Suppose that the number of distinct binary codes is n_d, and $n_d = 2^L$. An HA-Index for

this example is illustrated in Figure 3.4. The reason is that the $FLSSeq$ for the binary codes in the same window is maximized with Length $L-1$, and this $FLSSeq$ also shares the maximum similar patterns with its neighboring $FLSSeq$. Therefore, for the dataset with $n_d = 2^L$ data points and the built HA-Index as in Figure 3.4, the number of internal nodes $|V| = 2L$ or $|V| = 2\log_2 n_d$, and number of edges $|E| = 4(L-1)$ or $|E| = 4(\log_2 n_d - 1)$. This can be proven via induction (Details are omitted for brevity). Thus, the worst case for H-Search on this HA-Index is $|V| + |E| = 2\log_2 n_d + 4(\log_2 n_d - 1)$, i.e., is $O(\log_2 n_d)$. This indicates that H-Search can achieve the best performance under this scenario. We will discuss more general cases later.

Window size We discuss the relationship between window size, say as w, and binary string length L. Inspired by the previous extreme example, it is desirable that the n tuples can span the space of binary strings of L bits. L can be chosen such that $L = \lceil \log_2 n \rceil$, i.e., $2^{L-1} < n \leq 2^L$. Thus, if n is closer to 2^L, then the corresponding HA-Index is closer to the extreme case in our motivating example above. On the other hand, the smallest value for n is $2^{L-1} + 1$, and this is the worst case, i.e., the sparsest distribution of tuples on the space of binary strings of Length L. For the simplicity of discussion, we assume that the hashed binary strings are uniformly distributed.

Under the above assumption, the maximum Hamming distance L_m for a window of size of w satisfies $\lceil \log_2 w \rceil \leq L_m \leq L$. If $L_m = L$, then the binary strings in the same window cannot be merged together since no shared bit position exists. Therefore, a careful choice should be made on the window size w. The extreme case when $w = n$ is apparently a bad choice since no sharing pattern can be extracted from the window. A similar argument applies for $w = 1$. For smaller values of w, many internal nodes are generated and this results in indexes with larger heights. A suggested value for the window size w is $w = 2^{\lceil L/2 \rceil}$ when $n \approx 2^L$. Suppose that $w = 2^{\lceil L/2 \rceil}$, then the maximum Hamming distance L_m in each window satisfies $\lceil L/2 \rceil \leq L_m \leq L$.

Number of nodes in an HA-Index If $n \approx 2^L$, suppose that there are only few windows with $L_m = L$ and we denote the number of these binary codes within that window as δ_1. Since the leaves share about half of the bits in their binary codes, this results in a number of

$2^{\lceil L/2 \rceil} + \delta_1$ of internals nodes 1-level higher above the leaves, where $\delta_1 \ll 2^{\lceil L/2 \rceil}$. With the HA-index progressively growing, a higher level with $2^{\lceil L/4 \rceil} + \delta_2$ internal nodes can be built where $\delta_2 \ll \delta_1$. In the same way, the HA-index grows to the highest level with $2^{\lceil L/2^h \rceil} + \delta_h$ uppermost internal nodes, where h is the height of the index. Thus, the total number of nodes $|V|$ in the HA-index can be estimated by:

$$\begin{aligned}
|V| &= 2^{\lceil L/2 \rceil} + 2^{\lceil L/4 \rceil} + \cdots + 2^{\lceil L/2^h \rceil} + \sum_{i=1}^{h} \delta_i \\
&= 2^{\lceil \log_2 n^{1/2} \rceil} + 2^{\lceil \log_2 n^{1/4} \rceil} + \cdots + \sum_{i=1}^{h} \delta_i \\
&< 2 \times 2^{\lceil \log_2 n^{1/2} \rceil} + \sum_{i=1}^{h} \delta_i \\
&< 2 \times 2^{\lceil \log_2 n^{1/2} \rceil} \\
&= O(\sqrt{n}).
\end{aligned}$$

We can safely ignore the delta part since the summation is negligible compared to the dominant term.

If $n \approx 2^{L-1}$, then the window size w needs to shrink to a proper length. Based on the assumption of uniform distribution of the binary strings and Gray ordering, a proper window size can be set to $w = 2^{\lceil L/4 \rceil}$. The maximum Hamming distance L_m within a window satisfies $\lceil L/4 \rceil \leq L_m \leq L$. A similar analysis suggests that the number of internal nodes $|V'|$ satisfies:

$$\begin{aligned}
|V'| &= 2^{\lceil L/4 \rceil} + 2^{\lceil L/4^2 \rceil} + \cdots + 2^{\lceil L/4^h \rceil} + \sum_{i=1}^{h} \delta'_i \\
&= 2^{\lceil \log_2 n^{1/4} \rceil} + 2^{\lceil \log_2 n^{1/4^2} \rceil} + \cdots + 2^{\lceil \log_2 n^{1/4^h} \rceil} + \sum_{i=1}^{h} \delta'_i \\
&= O(\sqrt[4]{n}).
\end{aligned}$$

Number of Edges in an HA-Index For the number of edges in an HA-index, there are two extreme cases. Suppose that $n \approx 2^L$ and we have already discussed that the two levels above the leaves contain $2^{\lceil L/2 \rceil}$ and $2^{\lceil L/4 \rceil}$ internal nodes, respectively. The worst case is

that each of the $2^{\lceil L/2 \rceil}$ nodes connects to each of the $2^{\lceil L/4 \rceil}$ nodes. This induces about $2^{3L/4}$ edges. Similarly, the edge number can be estimated,

$$|E| = 2^{3L/4} + 2^{3L/8} + \cdots + 2^{3L/2^{h+1}} < 2 \times 2^{3L/4} = O(\sqrt[4]{n^3}).$$

On the other hand, the best estimate is that there are no cross edges between the children and different parents. For this case, a lower bound of the number of edges is $O(\sqrt{n})$, which is similar to the number of vertices.

Query Cost and Storage Space of the HA-Index The cost of H-Search is bounded by the number of nodes and edges, i.e., $|V| + |E|$. Therefore, the worst cost for H-Search is traversing all the edges and nodes in the HA-index. This indicates that H-Search can be bounded in the range $[O(\sqrt{n}), O(\sqrt[4]{n^3})]$. Meanwhile, besides the storage of the leaf nodes, the space usage of the HA-index also depends on the sum of the number of nodes and edges, i.e., $[O(\sqrt{n}), O(\sqrt[4]{n^3})]$. Compared to the state-of-the-art approaches [6, 47], the HA-Index does not need to maintain several copies of the dataset. Thus, it can be kept in memory for fast query processing. Furthermore, the internal nodes of the HA-Index store enough binary information for the whole dataset, and hence introduce low overhead to broadcast an HA-Index to each server.

3.4 Parallel Algorithm for Hamming-Join

To process Hamming-join on two datasets, say R and S, one straightforward approach is to build an HA-Index for R, then execute H-Search on the built index for each tuple of S. However, to build an HA-Index for R, sorting R would be slower as R gets larger. Secondly, executing H-Search between each tuple of S and the HA-Index for R would make the query time bounded by the number tuples in S. In this section, we address these limitations of the centralized environment and introduce Hamming-join on the MapReduce platform [54].

To support Hamming-join over MapReduce, we focus on two important issues. First, load balancing is important because the slowest mapper or reducer determines the job running time. Secondly, data shuffle from the mappers to reducers usually results in large disk

I/O and network communication costs that heavily influences the run-time performance. Therefore, we not only need to reduce the data shuffle cost, but also make sure data partitions in each mapper or reducer are well balanced.

3.4.1 Overview of MapReduce-based Hamming-Join

In this section, we introduce our implementation of the Hamming-join operation in MapReduce. As Figure 3.5 illustrates, the proposed algorithm includes three phases as explained below.

- **Preprocessing phase** Retrieve a sample from Datasets R and S. Then, use the sampled data to learn the hash function H. To handle data skew, build a data histogram for the sampled data and learn the data partitioning rule for the entire MapReduce job.

- **Global HA-Index building phase** Assume that the size of R is smaller than that of S. Partition R based on the pivot values from the data preprocessing step, then build the HA-Index for each partition using MapReduce by calling the H-Build function. Then, merge each local HA-Index to realize a global HA-Index for R.

- **Hamming-join phase** To join HA-Index of R with tuples in S, two possible options are applicable based on the size of R. More details are given later.

To learn the hash function, we utilize a random sample obtained from both R and S using reservoir sampling [55]. With the learned hash function H, high-dimensional data tuples in R and S are mapped into their corresponding binary codes. As discussed in the previous section, hash binary codes are ordered using the Gray order to preserve the clustering property. Hence, the data in each partition is more likely to share common $FLSSeq$ patterns. Then, we build the data histogram for the binary codes of the sampled data, and get a set of pivot values, denoted by *Pv*, for each Partition Pt_m. This guarantees that each partition receives approximately the same amount of data, where data in the various partitions is ordered according to the Gray order. More formally, given a set of data partitions

Pt, and a set *Pv* of corresponding binary code values that form the partitioning pivots, Tuple $t_i \in Pt_m$, if the Gray order for t_i's binary code, say $\hat{U}_i$, belongs to the pivot range, i.e., $\hat{U}_i \in [Pv_m, Pv_{m+1})$, where Pv_m and Pv_{m+1} are the pivot values for Partition Pt_m.

Thus, let $|Pt_m|$ be the number of tuples belonging to Partition Pt_m, Pivot set *Pv* partition dataset R, s.t $R = \bigcup_{m=1}^{N} Pt_m$, and $|Pt_m| \simeq |Pt_{m+1}|$. Therefore, we can build the HA-Index and Hamming-join in each server as illustrated below.

3.4.2 Global HA-Index Building

Given the set of pivot values *Pv* selected in the preprocessing step, a MapReduce job partitions the data and builds an HA-Index locally in each partition. Specifically, before launching the map function, the selected pivots *Pv* and the learned hash function H are loaded into memory in each mapper via distributed cache in MapReduce. A mapper sequentially reads each input data tuple, say t_i, from the mapper's corresponding partition. The hash function maps the high-dimensional input data tuples into their corresponding binary codes, i.e., U. Then, a binary search is performed for the closest pivots in *Pv*. For the closest partition region, Partition ID is assigned. Finally, the mapper(s) produce(s) as output each object t_i along with its Partition ID, original dataset tuple identifier (R or S), and its binary code value U.

In the data shuffling phase, the key-value pairs emitted by all map functions are grouped by each distinct Partition ID, and a reduce function is called within each node. Each reduce function computes the local HA-Index via the H-Build function of Section 4, and produces the local HA-Index as output. In addition, a postprocessing step to merge the various local HA-Indexes into one global HA-Index. Mainly, non-leaf nodes with the same $FLSSeq$ from the different local HA-Indexes are merged into one node, and the corresponding edges between the index nodes are relinked. Because the HA-Index is relatively small, the processing overhead is acceptable. After the first MapReduce job finishes, the global HA-Index for dataset R is built. This index is used by H-Search in the next phase.

3.4.3 Hamming-Join

The second MapReduce job performs the Hamming-join in two possible ways.

Option(A): When Dataset R is small, i.e., storage of the leaf nodes of the HA-Index does not dominate the space of the HA-Index, the HA-Index maintains the leaf nodes as in Figure 3.3. Next, the Map function partitions Dataset S into N parts, i.e., $S = \bigcup_{i=1}^{N} S_i$. Then, it duplicates the global HA-Index for Dataset R and broadcasts to each server. The Map function computes the Hamming-join for Partition S_i and the replicated HA-Index of R. Specifically, before launching the MapReduce Job, the master node broadcasts the pivots *Pv*, the hash function H, and the global HA-Index of R to various servers. The main task of the mapper in the second MapReduce Job is to map high-dimensional data into binary codes, then partition dataset S into N partitions. Next, each reducer performs the Hamming-join between a pair of HA-Index and $\hat{S}_i$, and output the Hamming-join results.

Option(B): If Dataset R is big, e.g.,the number of tuples $|R|$ is more than millions, the storage of leaf nodes of the HA-Index dominates the space usage of the HA-Index. Therefore, the HA-Index of Dataset R does not maintain leaf nodes, and is duplicated to each server. By this way, the H-Search Algorithm 12 only returns the qualifying binary codes for Hamming-select, and a post-precessing step is carried out to find the tuple IDs for the qualifying binary codes. Take query tuple t_6 in Table 3.2.1 as an example. The H-Search algorithm computes binary codes from Table 3.2.1, i.e., "101100010" and "101010010", which have a Hamming distance of 3 from t_6. In order to find the tuple IDs for those qualifying binaries, one post-processing step is invoked. Naturally, if Dataset R fits into memory, then the qualifying binaries are joined with R's hash table in memory. On the other hand, if Dataset R is too large to fit in memory, MapReduce hash-join [56] for Dataset R and the qualifying binaries is applied.

3.4.4 Shuffle Cost Analysis

The performance of MapReduce Hamming-join depends on the running time of Hamming-select as well as on the data shuffling cost. Let $|R| = m$ and $|S| = n$, re-

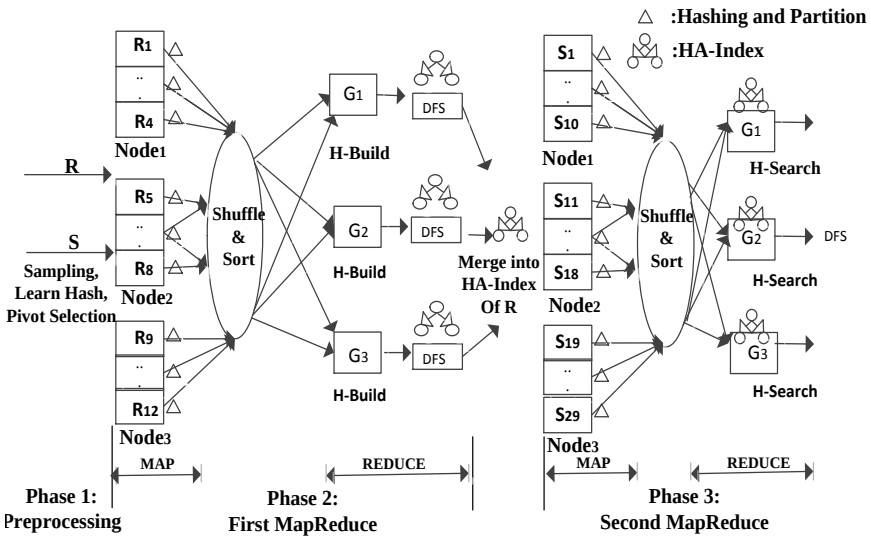


Figure 3.5.: An overview of Hamming-Join processing in MapReduce.

spectively, d be the data dimension, and N be the number of partitions. In the previous work [6], Dataset R is duplicated and broadcast to each server, and the data shuffling cost is approximate to $O(mNd + nd)$. In this work, instead of duplicating the whole dataset R, only the HA-Index, is broadcast to each server. Hence, the data shuffling cost is reduced to $O(|HA|N + n)$, where $|HA|$ is the size of the HA-index. As introduced in Section 4, the space storage of HA is bound to $[O(\sqrt{m}), O(\sqrt[4]{m^3})]$. Therefore, the shuffling cost is bounded in $[O(\sqrt{m}N + n), O(\sqrt[m]{n^3}N + n)]$.

3.5 Related Work

Using the Hamming distance as a similarity metric has been studied in the theory community, e.g., [45, 46]. When the Hamming-distance query threshold is small, i.e., $\hat{h} = 1$, Yao et.al [46] propose an algorithm with $O(m \log\log(n))$ query time and $O(nm \log(m))$ space. Yao's algorithm recursively cuts the query binary code and each binary code in the

dataset in half, and then finds exact matches in the dataset for the left or the right half of the query binary code. [57] demonstrates that similarity search in chemical information via the Tanimoto Similarity metric can be transformed into a Hamming-distance query. In a most recent work [58], each binary code is mapped to a set and an inverted index is built on the set.

Hamming-distance queries are attracting more attention for processing large volumes of data. A relatively recent work [6] uses multiple hash tables, and hence more space, to reduce the linear computation of the Hamming distance during query time. The idea behind this approach is that if two binary codes are within a Hamming distance $\hat{h}$, then at least one of the $\hat{h}$+1 segments are exact matches for two binary codes. This algorithm needs to replicate the database multiple times, and it sorts each copy based on parts of the segments. The Hamming-distance computation is still performed in a linear fashion over tuples of the same bucket in a certain hash table. Thus, it fails to scale as data size increases. For performing a Hamming-join of two datasets, say R and S, [6] extends the sequential approach to MapReduce by broadcasting Table R into each server, then applying a sequential algorithm between R and S. This approach is subject to a very heavy shuffling cost and servers cannot work in a load-balanced way when data is skewed. HEngine [47] adopts a similar idea to that in [6], but uses approximate matching instead by generating multiple one-bit difference binary codes. The HEngine uses less memory but achieves limited performance speedup. HmSearch [57] is an exact matching approach that index over signature of the binary codes. The size of the index increases dramatically, because HmSearch need to generated large amount of unique signatures. If used in the context of MapReduce, the shuffling cost between the mappers and the reducers is expected to be expensive. Our proposed HA-Index extracts and groups similar binary codes from among the various tuples to reduce the cost of shuffling and hence is applicable to MapReduce as we illustrate later in this chapter. Through data sampling, we partition that data in a way that uniformly distributes the dataset among the reducer servers and hence enables better load balancing. Experimental comparison with [6, 47] shows that our proposed HA-

Index is two orders of magnitude faster and uses ten times less memory as illustrated in the experimental section of this chapter.

Two related and popular operations to Hamming distance queries are the k-nearest-neighbor select (kNN-select) and k-nearest-neighbor join (kNN-join) [9, 59]. Given a dataset, say S, and a query focal point, say t_q, kNN-select finds in S the k-nearest-neighbors to t_q. Given two datasets, say R and S, R kNN-join S finds the k-nearest-neighbors in S for each tuple in R. In high-dimensional spaces and because of the curse of dimensionality [60], data-independent hash-based approximate kNN (e.g., locality sensitive hashing (LSH) [61]) has attracted attention as it can speed-up query execution while having acceptable error margins. Recently, data-dependent hashing has been proposed to learn the hash function, say $H()$, given the underlying dataset, e.g., as in [12]. There has been a plethora of work in learning good and representative hash functions, e.g., [11–13]. Given the learned similarity hash function $H()$, a tuple, say t_i, is mapped into its binary code, say U_i, i.e., $H(t_i) = U_i$. Afterwards, all the binary codes of the dataset R are scanned to find data tuples that are different from the query's binary code U_i by at most $\hat{h}$ bit-positions. If the answer set size is more than k, then only the k-closest answers are retained. However, if the size of the result set is less than k, then a larger distance threshold is estimated and the near neighbor query is repeated. The process is stopped when k or more answers are reported. Notice that the core of the method for approximate kNN search is a Hamming-distance query with a threshold $\hat{h}$. In our experiments, we use the state-of-the-art approach [12] to learn the hash function, and show how our proposed approach can speed up approximate kNN-select and kNN-join.

3.6 Performance Evaluation

We implement all the algorithms in Java. The experiments for Hamming-select are performed on an Intel(R) Xeon (R) E5320 1.86 GHz 4-core processor with 8G memory running Linux. The experiments on MapReduce are performed on a cluster of 16 nodes of

Intel(R) Xeon (R) E5320 1.86 GHz 4-core machines with 8GB of main memory running Linux. We use Hadoop 0.22 and apply the default cluster environment setting. We evaluate the performance of the proposed techniques using the following three high-dimensional real datasets: (1) NUS-WIDE[2] is a web image dataset containing 269,648 images. We use 225-D block-wise color moments as the image features, thus obtaining a 225-dimension data. (2) Flickr[3] is a an image hosting website. We crawled 1 million images and extracted 512 features via the GIST Descriptor [62] (the data dimension is 512). (3) DBPedia[4] data aims to extract structured content from Wikipedia. We extract 1 million documents, and then apply standard NLP techniques to pre-process the documents, e.g., to remove stop words. We use the Latent Dirichlet Allocation (LDA) [7] model to extract topics, and we keep 250 topics for each document.

To evaluate the performance on larger data sizes, we synthetically generate more data while maintaining the same distribution as the original data distribution, e.g., as in [48,49]. Suppose that the original dataset D has k dimensions. First, we get the frequencies of values in each dimension, and then sort the data in ascending order of their frequencies. Therefore, k copies of the dataset D are generated, one copy per dimension, e.g., D_j one copy of the dataset that is sorted based on the j-th dimension. Then, for each tuple, say t, in Dataset D, $t \in D$, we create a new tuple, say $\hat{t}$, according to the position of each component of t in the corresponding sorted copy D_j. For example, $t = (t_1, \ldots, t_j, \ldots, t_d)$ and t'_j is the first value larger than t_j in copy D_j, then $\hat{t} = (t'_1, \ldots, t'_j, \ldots, t'_d)$. If t_j is the largest element in Copy D_j, then $\hat{t}_j = t_j$. We use"$\times s$" to denote the increase in dataset size, where $s \in [5, 25]$ is the increase or scale factor. We consider the following approaches to evaluate Hamming-select:

(1) **Nested-Loops** is the naive approach to linearly XOR and count the binary data to perform the Hamming-distance computation. (2) **MultiHashTable** [6] is the state-of-the-art to search binary codes for similarity hashing that uses multiple-hash tables to reduce the linear search cost. While a large number of hash tables can achieve better performance, we

[2]http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm

[3]htttp://www.flickr.com

[4]http://wiki.dbpedia.org/About

limit ourselves to just 4 and 10 hash tables to avoid memory overflow. For short, we refer to these two possibilities, as MH-4 and MH-10. (3) **HEngine^s** [47] is the most recent work to improve the MultiHashTable approach in query time and memory usage. (4) **Radix-Tree** is the approach introduced in Section 4.2. (5) **Static HA-Index (SHA-Index) and Dynamic HA-Index (DHA-Index)** are the approaches introduced in Sections 4.3 and 4.4, respectively. SHA-Index(32) or DHA-Index(32) means that the length of the binary code is 32 bits.

We further evaluate the following approaches for kNN-select, and show how the approximate kNN-select can benefit from the enhancement of HA-Index searching over binary codes: (1) **Locality-Sensitive Hashing(E2LSH)** [61] is the state-of-the-art implementation for the data-independent LSH. We use 20 hash tables for E2LSH. (2) **LSB-TREE** [63] uses the Z-order curve to map high-dimensional data into one-dimensional Z-values, and index the Z-values using a B-tree. In our experiments, we build the LSB-Tree with 25 trees to compare the performance.

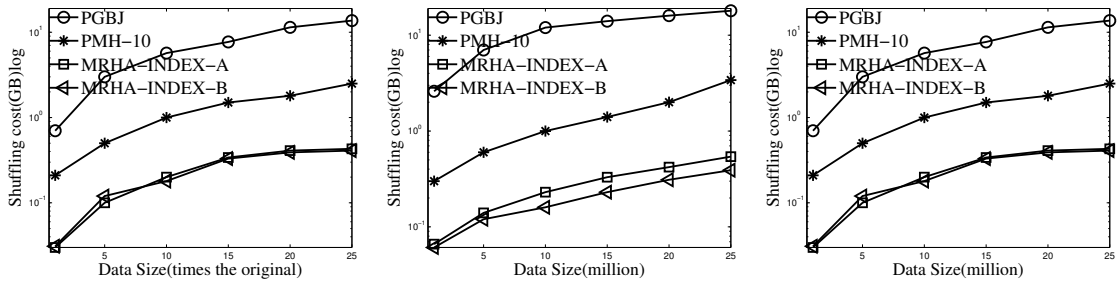
Also, we evaluate the following approaches to test the Self-Hamming-join, and verify how our approach of Map-Reduce Hamming-join can speedup the state-of-art algorithm for exact Self-kNN-join: (1) **Parallel-exact-KNN-join** (short as PGBJ) [49] is the state-of-the-art approach for performing exact kNN-join over multi-dimensional data in MapReduce, and it is 10 times speedup over the Z-order curve based approach [50]. We get the implementation generously provided by the authors [49]. (2) **Parallel Hamming-join via MultiHashTable** (PMH, for short) that handles approximate batch queries for web page duplicate identification [6]. PMH-10 means that 10 hash tables are used. (3) **Parallel Hamming-join via Dynamic HA-Index** (MRHA-Index, for short) is the approach introduced in Section 5. Specifically, in terms of the Hamming-join phase, if Option A is used, we term it MRHA-Index-A, and if Option B is used, we term it MRHA-Index-B.

The performance measures for each algorithm include the query time, the index update time, the index building time, memory usage, and the data shuffling cost. All performance measures are averaged over eight runs. Some running times are not plotted because they would use more than five hours. Unless mentioned, the default value of k is 50, and the

Hamming-distance threshold $\hat{h}$ is 3. We choose the state-of-the-art Spectral Hashing [12] as the hash function in our experiments, but our approach is not limited to this hash function.

3.6.1 Results for Hamming-select

Effectiveness of the HA-Index



(a) NUS-WIDE (b) Flickr (c) DBPedia

Figure 3.6.: Shuffling cost of Hamming-Join and kNN-Join

Table 3.5, 3.6 and 3.7 summarizes the query time, index update time, and memory space usage by the various approaches. The dynamic-HA-Index is the most efficient in terms of query time and space usage, the binary code length is 32 bits. Notice for DHA-Index, **28/11** means 28MB and 11MB space usage for internal and leaf nodes were kept or only internal nodes, respectively. Specifically, index update corresponds to the operation to delete one tuple first, then insert the same tuple back into the index. From TableTable 3.5, 3.6 and 3.7 , we have the following observations: 1) The Radix-Tree and HA-index-based approaches outperform the naive nested-loop and state-of-the-art methods [6, 47] on query time for the three datasets, mainly because the new proposed approach avoids many redundant Hamming-distance computations, and avoids scanning all the underlying data when they are hashed into the same bucket; 2) The HA-Index-based approach, i.e., the Static and Dynamic HA-Indexes, outperforms the Radix-Tree approach. The speedup is around 10 times because the Radix-Tree behaves as a prefix tree when many of the binary codes do not share long common prefixes, and hence cannot avoid the redun-

dant Hamming distance computations; 3) The Static HA-Index shows better index-update time than that of the Dynamic HA-Index because the static segmentation enables us to track different binary segmentations directly, thus, we can search the paths of binary codes more efficiently; 4) The Radix-Tree and the HA-Index-based approaches save more memory than the state-of-the-art methods [6,47] because the HA-Index-based approaches do not need to duplicate tuples and can share common $FLSS$s and $FLSSeq$s for different binary codes. This can reduce memory usage further; 5) For the Dynamic-HA-Index, if only the internal nodes of the HA-Index are kept, the memory usage can be reduced further. For instance, the memory usage for the Flickr and DBpedia datasets is reduced from 251MB and 225MB to 63MB and 47MB, respectively.

Effect of Hamming-Distance Threshold

We evaluate whether the running time of proposed approach is sensitive to the query threshold $\hat{h}$. Figure 3.9 gives the data query time when varying the Hamming-distance threshold. Notice that the query time of both the HA-Index-based approaches increases relatively slowly as the threshold increases. The reason is that the searching process in the HA-Index usually terminates early in the upper-level nodes, and this can improve the query speed. On the other hand, the searching path length of the Radix-Tree is not under control, and it tends to reach each leaf node when the Radix-Tree shares very little and changes to a prefix-tree-like format. However, state-of-the-art methods [6, 47] are sensitive to the Hamming-distance threshold because both approaches have to scan intermediate data to filter out non-qualifying tuples. Hence, the bigger $\hat{h}$ is, the more intermediate results that need to be scanned. This directly degrades the performance.

Effect of HA-Index Parameters

We study the effects of the window length and the index depth of the dynamic HA-Index w.r.t. the index building and query processing times. The window length is normalized by the number of tuples in the dataset. Figure 3.7(a) illustrates that the building time for the HA-index drops as the depth decreases. The reason is that index construction stops early while the depth is small. Meanwhile, the HA-Index building time grows as the window size increases because the time to extract the same subpatterns for binaries of one window

depends on the number of tuples inside the window. Meanwhile, the query processing time demonstrates stable growth as the window size and index depth increase. Observe that the window size increases four times and the query processing time only grows by less than 10%. Thus, the HA-Index is not sensitive to these parameters.

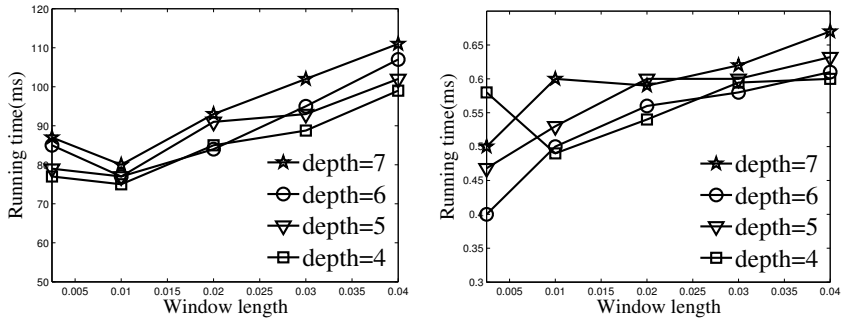
Comparison of Approaches for kNN-Select

As introduced in Section 2, Hamming-select is a core operation for evaluating approximate kNN-select. In this section, we demonstrate the performance gains when using the HA-Index to speedup approximate kNN-select. Table 3.8 illustrates the runtime for data querying and index construction for LSH, LSB-Tree, and the HA-Index-based approaches, note that the dataset size is set to 300k tuples. Observe that the HA-Index-based approach outperforms the state-of-the-art methods on all tasks when the binary code length is relatively large (i.e., 32 or 64 bits). Compared to the LSH approach, both HA-index-based approaches achieve two orders of magnitude speedup. The reason is that the LSH approach assumes uniformity in the distribution of the underlying data while real datasets are not uniform. In addition, the LSB-Tree can improve the query time compared to the LSH approach. However, the time to build the LSB-Tree index is expensive (more than 24 hours). In addition, the query and index building times for the HA-Index-based approach increases relatively smoothly as the binary code length increases. This demonstrates that the HA-Index approach is robust with the binary code length. Finally, the LSB-Tree consumes extensive disk space to store the index, LSB-Tree uses more than 20GB to store the index for the Flickr data, while the HA-Index-based approach only takes less than 300MB. This significantly reduces disk I/O time for the HA-Index-based approach.

3.6.2 Results of Hamming-Join in MapReduce

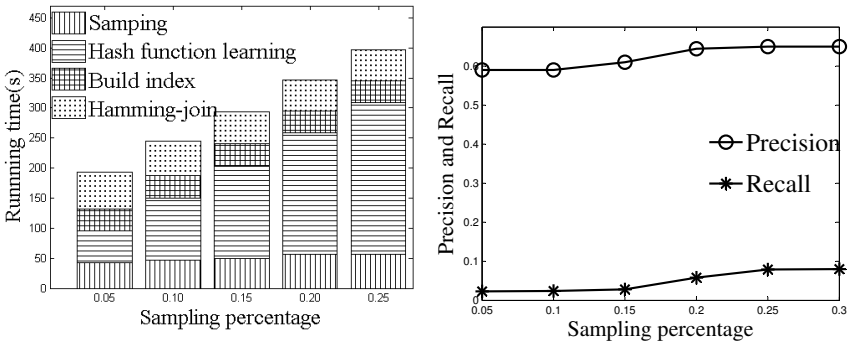
Shuffling Cost

We measure the effect of data size on the shuffling cost for PGBJ, PMH and the MRHA-Index. Figure 3.6 gives the data shuffle costs when the data size varies. The shuffle cost is plotted in logarithmic scale. The smaller the shuffle costs, the better the performance is. We



(a) NUS-WIDE (b) Flickr

Figure 3.7.: DHA-Index building time and query processing when varying the window size.



(a) NUS-WIDE (b) Flickr

Figure 3.8.: Effect of sampling on query processing time and precision/recall

observe that the shuffle costs for approximate kNN-join approach, i.e., PMH and MRHA-INDEX, are 10 times smaller when compared to the PGBJ approach. The reason is that the hashing technique maps the high-dimensional data into binary codes, and hence the data shuffling cost does not depend on the dimensions of the data. Notice that the data shuffling cost for PGBJ increases linearly with the data size. This is two orders of magnitude worse when compared to the data shuffling cost for the MRHA-INDEX approach. Duplicating and distributing the HA-Index into different nodes can improve the data shuffle cost 10 times less than that of the PMH approach. On the other hand, the larger shuffle cost would

stop the PGBJ approach from achieving a linear speedup and its corresponding execution time shows quadratic increase. The corresponding running times are given below. Finally, for the Hamming-join step in the HA-Index-based approach, Option B saves more data shuffling cost than Option A because the former does not need to duplicate the whole dataset into each server, and hence the space usage of the HA-Index remains relatively small.

Scalability and Speedup

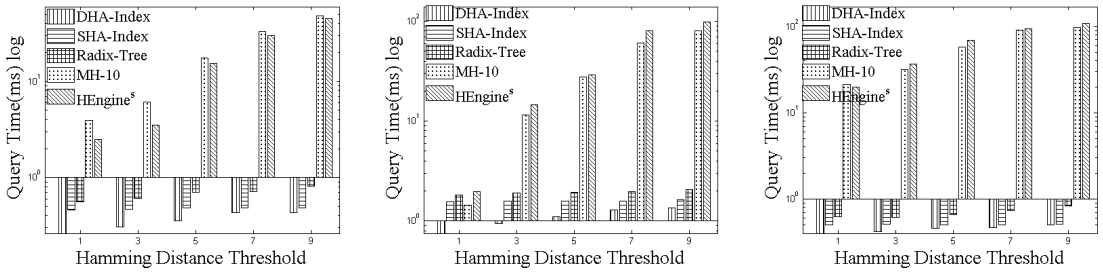
We investigate the scalability of the three approaches in Figure 3.10. The figure presents the results by varying the data size from 1 to 25 times of the original dataset sizes. From the figure, the overall execution time of PGBJ shows quadratic increase when the data size increases. For example, PGBJ's running time is almost 13 hours when the data is DBPedia$\times 15$, which is excessively slow. The approximate kNN-join via similarity hashing always outperforms the PGBJ approach. Comparing with the state-of-the-art PMH-10 approach, the running time of the HA-Index outperforms PMH-10 by 5 times.

Effect of Data Sampling

Figure 3.8(a) gives the query execution time for the various processing phases of Hamming-join. From the Figure, more sampling of the data reflects the global data distribution more clearly, and this helps the sampling data pivot to partition different regions more evenly, and hence, improves the parallel HA-Index building and Hamming-join query time. The hash function learning usually takes more time, but for real-world applications, we only need to learn the hash function again when a certain amount of the new data is updated, which can save the time. Figure 3.8(b) illustrates how data sampling affects the query quality. Observe that the precision and recall can moderately improve as the sampling data size increases. However, the recall value is low.Improving the recall and precision can be found in [64].

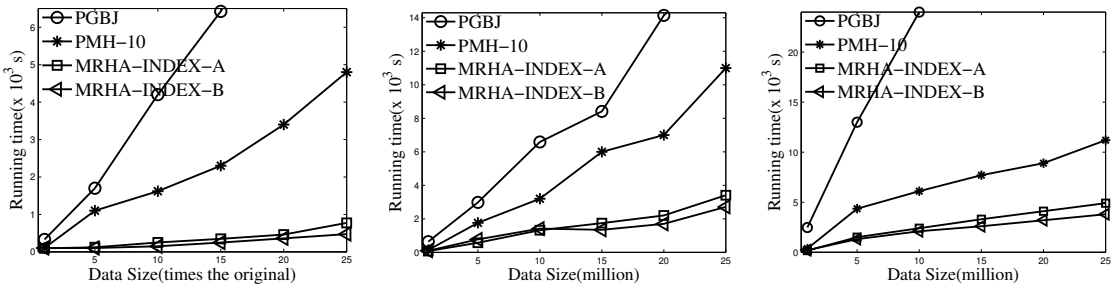
3.7 Summary

In this chapter, we study the problem of efficiently performing the Hamming-select and Hamming-join operations. The proposed HA-Index approach executes the Hamming-distance-based similarity operations while avoiding unnecessary Hamming-distance computations. Extensive experiments using real datasets demonstrate that the proposed approaches outperforms the state-of-the-art techniques by two orders of magnitude. We explore how to extend the HA-Index approach to support Hamming-join in MapReduce. The new proposed approach can reduce the data shuffling cost and save computation time. In the future, it would be interesting to explore Hamming-distance similarity operation for relational operation i.e., intersection [65].



(a) NUS-WIDE (b) Flickr (c) DBPedia

Figure 3.9.: Effect of Hamming-distance threshold on Hamming select



(a) NUS-WIDE (b) Flickr (c) DBPedia

Figure 3.10.: Speedup and scalability: Running time of MapReduce Hamming-Join and kNN-Join.

Algorithm 12: H-Search

Input: t_q: Query tuple, $\hat{h}$: Hamming distance query threshold, *HA*: HA-Index for queried dataset

Output: *ret*: Qualified tuple in *HA* within Hamming distance $\hat{h}$ from tuple t_q

```
1  q: Queue.
2  for each top level node n_i in HA do
3      if hdist(t_q, n_i) ≤ ĥ then
4          n_i.h ← hdis(t_q, c);
5          q.enqueue(n_i);
6      end
7  end
8  while q is not empty do
9      var n:Node;
10     q.dequeue(n);
11     if n is a non-leaf node then
12         for all children node c of n do
13             if (hdis(t_q, c)+n.h)≤ ĥ then
14                 var m:Node;
15                 m.b ← combine(c.b, n.b); //combine binary code of c and n
16                 m.h ← hdis(t_q, c)+n.h; //update Hamming distance
17                 m.children ← c.children ;
18                 q.enqueue(m);
19             end
20         end
21     end
22     else
23         var binary ← getBinary(n);
24         var tuple ← gettuple(binary);
25         ret.insert(tuple);
26     end
27 end
28 output ret;
```

Table 3.4.: Sample execution trace for H-Search

Queue	**Qualified tuples** ret
N_{11}, N_{12}	$\emptyset$
N_{12}, $[N_7, N_{11}]$	$\emptyset$
$[N_7, N_{11}]$, $[N_9, N_{12}]$	$\emptyset$
$[N_9, N_{12}]$	t_0
$\emptyset$	t_0

Table 3.5.: Overall comparative study for Hamming-select: NUS-WIDE.

method	query time(ms)	update time(ms)	space usage(MB)
Nested-Loops	16.42	15.22	/
MH-4	6.22	0.21	475
MH-10	4.91	0.25	531
HEngine^s	3.53	0.45	210
Radix Tree	1.61	0.19	39
SHA-Index	0.87	**0.16**	29
DHA-Index	**0.68**	0.18	**28/11**

Table 3.6.: Overall comparative study for Hamming-select: Flickr.

method	query time(ms)	update time(ms)	space usage(MB)
Nested-Loops	42.97	41.19	/
MH-4	16.09	0.60	712
MH-10	14.03	0.83	1204
HEngine^s	14.75	1.14	820
Radix Tree	3.98	0.64	365
SHA-Index	1.75	**0.52**	254
DHA-Index	**0.74**	0.58	**251/63**

Table 3.7.: Overall comparative study for Hamming-select: DBPedia.

method	query time(ms)	update time(ms)	space usage(MB)
Nested-Loops	59.16	53.53	/
MH-4	40.28	0.45	819
MH-10	34.46	0.64	1364
HEngine^s	36.91	1.91	763
Radix Tree	17.64	0.44	352
SHA-Index	3.54	**0.43**	239
DHA-Index	**1.07**	0.51	**225/47**

Table 3.8.: Comparison with the state-of-the-art kNN-select approaches

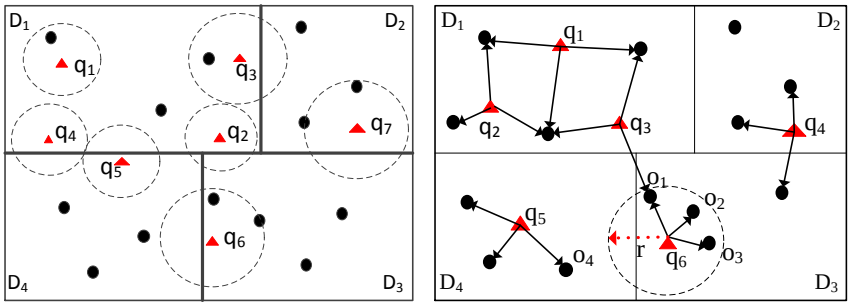
Dataset	Algorithm	Query time(ms)	Index build time
NUS-WIDE	LSH	2400	680(s)
	LSB-Tree(25)	47	37(Hr)
	SHA-Index(32)	2.74	68(s)
	SHA-Index(64)	4.78	97(s)
	DHA-Index(32)	1.64	87(s)
	DHA-Index(64)	2.43	103(s)
Flickr	LSH	340	1080(s)
	LSB-Tree(25)	63	50(Hr)
	SHA-Index(32)	2.21	176(s)
	SHA-Index(64)	3.54	189(s)
	DHA-Index(32)	2.17	210(s)
	DHA-Index(64)	2.88	244(s)
DBpedia	LSH	266	340(s)
	LSB-Tree(25)	59	44(Hr)
	SHA-Index(32)	2.94	150(s)
	SHA-Index(64)	4.88	290(s)
	DHA-Index(32)	2.18	230(s)
	DHA-Index(64)	3.85	310(s)

4 IN-MEMORY DISTRIBUTED SIMILARITY QUERY PROCESSING AND OPTIMIZATION FOR SPATIAL DATA

4.1 Introduction

Spatial computing is becoming increasingly important with the proliferation of mobile devices. Meanwhile, the growing scale and importance of location data have driven the development of numerous specialized spatial data processing systems, e.g., SpatialHadoop [14], Hadoop-GIS [66] and MD-Hbase [15]. By taking advantage of the power and cost-effectiveness of MapReduce, these systems typically outperform spatial extensions on top of relational database systems by orders of magnitude [66]. MapReduce-based systems allow users to run spatial queries using predefined high-level spatial operators without worrying about fault tolerance or computation distribution. However, these systems have the following two main limitations: (1) They do not leverage the power of distributed memory, and (2) They are unable to reuse intermediate data [67]. Nonetheless, data reuse is very common in spatial data processing. For example, spatial datasets, e.g., Open Street Map (OSM, for short, $>$60G) and Point of Interest (POI, for short, $>$20G) [14], are usually large. It is unnecessary to read these datasets continuously from disk (e.g., using HDFS [68]) to respond to user queries. Moreover, intermediate query results have to be written back to HDFS, thus directly impeding the performance of further data analysis steps.

One way to address the above challenges is to develop an efficient execution engine for large-scale spatial data computation based on an in-memory computation framework (in this case, Spark [67]). Spark is a computation framework that allows users to work on distributed in-memory data without worrying about data distribution or fault-tolerance. Recently, various Spark-based systems have been proposed for spatial data analysis, e.g., SpatialSpark [69], GeoSpark [70], Magellan [71], Simba [72] and LocationSpark [3].



(a) Spatial-Range-Join (b) kNN Join

Figure 4.1.: Illustration of Spatial-Range-Join and kNN-Join operators

Although addressing several challenges in spatial query processing, none of the existing systems is able to overcome the computation skew introduced by spatial queries. "Spatial query skew" is observed in distributed environments during spatial query processing when certain data partitions get overloaded by spatial queries. Traditionally, distributed spatial computing systems (e.g., [14, 66, 69]) first learn the spatial data distribution by sampling of the input data. Afterwards, spatial data gets evenly partitioned into equi-sized partitions. For example, in Figure 4.1, the data points with dark dots are evenly distributed into four partitions. Given the partitioned data, consider the spatial range and kNN joins that serve as primitive operations to combine two datasets, say D and Q, with respect to a spatial relationship. Refer to Figure 4.1(a) for illustration. For each point $q \in Q$, a spatial range join returns data points in D that are inside the radius of the circle centered at q. In contrast, a kNN join (refer to Figure 4.1(b) for illustration) returns the k nearest-neighbors from the dataset D for each query point $q \in Q$. Both spatial operators are expensive and may incur computation skew in certain workers, thus greatly degrading the overall performance of query processing.

For illustration, consider a large spatial dataset, with millions of points of interests (POIs), that is preprocessed and is partitioned into different computation nodes based on the spatial distribution of the data, e.g., one data partition represents data from San Francisco, CA, and another one for Chicago, IL, etc. Assume that we have incoming queries

from people looking for different POIs, e.g., restaurants, train or bus stations, and grocery stores, around their locations. These spatial range queries are consolidated into batches to be joined via an index to the POI data (e.g., using indexed nested-loops join). After partitioning the incoming spatial queries based on their locations, we observe the following issues: During rush hours in San Francisco from 4PM to 6PM (PST), San Francisco's corresponding data partition may encounter more queries than the data partition in Chicago, since Chicago is already evening. Without an appropriate optimization technique, the data partition for San Francisco will take much longer time to process its corresponding queries while the workers responsible for the other partitions are lightly loaded. As another example, in Figure 4.1, the data points (the dark dots) correspond to Uber car's GPS records where multiple users (the triangles) are looking for the Uber carpool service around. Partition D_1 that corresponds to an airport, experiences more queries than other partitions because people may prefer using Uber at this location. Being aware of the spatial query skew provides a new opportunity to optimize queries in distributed spatial data environments. The skew partitions have to be assigned more computation power to reduce the overall processing time.

Furthermore, communication cost, generally a key factor of the overall performance, may become a bottleneck. When a spatial query usually touches more than one data partition, it may be the case that some of these partitions do not contribute to the final query result. For example, in Figure 4.1(a), queries q_2, q_3, q_4, and q_5 overlap more than one data partition (D_1, D_2, and D_4), but these partitions do not contain data points that satisfy the queries. Thus, scheduling queries (e.g., q_4 and q_5) to the overlapping data partition D_4 incurs unnecessary communication cost. More importantly, for the spatial range join or kNN join operators over two large datasets, the cost of network communication may become prohibitive without proper optimization.

In this chapter, we introduce LOCATIONSPARK, an efficient in-memory distributed spatial query processing system. In particular, it has a query scheduler with an automatic skew analyzer and a plan optimizer to mitigate query skew. The query scheduler uses a cost model to analyze the skew for spatial operators, and a plan generation algorithm

to construct a load-balanced query execution plan. After plan generation, local computation nodes select the proper algorithms to improve their local performance, based on the available spatial indexes and the registered queries on each worker. Finally, to reduce the communication cost when dispatching queries to their overlapping data partitions, LOCATIONSPARK develops a new spatial Bloom filter (called sFilter) that can speed up query processing by avoiding needless communication with data partitions that do not contribute to the query answer. We implement LOCATIONSPARK as a library in Spark that provides spatial query processing and optimization APIs based on the Spark's standard dataflow operators. LOCATIONSPARK requires no modifications to Spark, revealing a general method to combine spatial data processing within distributed dataflow frameworks.

The rest of this chapter proceeds as follows. Section 4.2 presents the problem definition, and an overview of LOCATIONSPARK. Section 4.3 introduces the cost model and the cost-based query plan scheduler and optimizer and their corresponding algorithms. Section 4.4 presents the empirical study for local execution plans in local computation node. Section 4.5 introduces the spatial Bloom filter, and explains how it can speedup spatial query processing in a distributed setup. The experimental results are presented in Section 4.6. Section 4.7 introduces the related work. Finally, Section 4.8 concludes this chapter.

4.2 Preliminaries

4.2.1 Data Model and Operators

LOCATIONSPARK stores spatial data as key-value pairs. A tuple, say o_i, contains a spatial geometric key k_i and a related value v_i. The spatial data type for key k_i can be a two-dimensional point, e.g., latitude-longitude, a line-segment, a poly-line, a rectangle, or a polygon. The value type v_i is specified by the user, e.g., a text data type if the data tuple is a tweet. In this chapter, we assume that queries are issued progressively by users, and are processed by the system in batches (i.e., similar to the DStream model [67]).

LOCATIONSPARK supports various types of spatial query predicates including spatial range search, k-NN search, spatial range join, and kNN join. In this chapter, we focus our

discussion on the spatial range join and the kNN join operators. These two operators are prohibitively expensive especially when processing big spatial data.

Definition 4.2.1 ***Spatial Range Search*** *- $range(q, D)$: Given a spatial range area q (e.g., circle or rectangle) and a dataset D, $range(q, D)$ finds all tuples from D that overlap the spatial range defined by q.*

Definition 4.2.2 ***Spatial-Range-Join*** *- $Q \bowtie sj\ D$: Given two dataset Q and D, $Q \bowtie sjD$, combines each object $q \in Q$ with its range search results from D, $Q \bowtie sjD = \{(q, o)|q \in Q, o \in range(q, D)\}$.*

Definition 4.2.3 ***kNN Search*** *- kNN(q,D): Given a query tuple q, a dataset D, and an integer k, kNN(q,D), returns the output set $\{o|o \in D$ and $\forall s \in D$ and $s \neq o, ||o, q|| \leq ||s, q||\}$, where the number of output objects from D, $|kNN(q, D)|$, is k.*

Definition 4.2.4 ***kNN-Join*** *- $Q \bowtie knn\ D$: Given a parameter k, kNN join of Q and D computes each object $q \in Q$ with its k nearest neighbors from D. $Q \bowtie knn\ D = \{(q, o)|\forall q \in Q, \forall o \in kNN(q, D)\}$.*

4.2.2 Overview of Distributed Similarity Query Processing

To facilitate spatial query processing, we build a distributed spatial index for in-memory spatial data. Given a spatial dataset D, we obtain samples from D and construct a spatial index (e.g., an R-tree) with N leaves over the samples. We refer to this index on the sample data by the *global spatial index*. Next, each worker partitions the dataset D into N partitions according to the built global spatial index via data shuffling. The global spatial index guarantees that each data partition approximately has the same amount of data. Then, each worker takes $1/N$th of the data, and builds a local spatial index for its local data partition, say D_i. Finally, the indexed data (termed the LocationRDD) is cached into memory. Figure 4.2 gives the architecture of LOCATIONSPARK and the physical representation of the partitioned spatial data based on the procedure outlined above, where the master node stores the global spatial index that indexes the data partitions, while each worker has a local

spatial index over the local spatial data within the partition. Notice that the global spatial index partitions the data into local LocationRDDs as in Figure 4.2, and this index can be copied into various workers to help partition the data in parallel. The type of local index, e.g., a Grid, an R-tree, or an IR-tree, for a data partition can be determined based on the specifics of the application scenarios.

For spatial range join, two strategies are possible; either replicate the outer table and send it to the inner table data or replicate the inner table data and send it to the different processing nodes where the outer table tuples are. In shared execution, the outer table is typically a collection of range query tuples and the inner table is the queried dataset. If this is the case, then it makes sense to send the outer table of queries to the inner data tables as the outer table of queries will be much smaller in size compared to the inner data tables. In this paper, we adopt the first approach because it is impracticable to replicate and forward copies of the large inner data table.

Thus, each tuple $q \in Q$ is replicated and is forwarded to the partitions that spatially overlap q. These overlapping partitions are identified using the global index. Then, a post-processing step merges the local results to produce the final output. For example, outer table tuple q_2 in Figure 4.1(a) is replicated and is forwarded to data partitions D_1, D_3, and D_4. Then, we execute a spatial range search on each data partition locally. Next, we merge the local results to form the overall output of tuple q. As illustrated in Figure 4.2, the outer table that corresponds to a shared execution plan's collection of queries (termed queryRDD) are first partitioned into qRDD based on the overlap between the queries in qRDD and the corresponding data partitions. Then, local search takes place over the local data partitions of LocationRDD.

The kNN join operator is implemented similarly in a simple two-round process. First, each outer focal points $q_i \in Q$ is transferred to the worker that holds the data partition that q_i spatially belongs to. Then, the kNN join is executed locally in each data partition, producing the kNN candidates for each focal point q_i. Afterwards, the maximum distance from q_i to its kNN candidates, say radius r_i, is computed. If the radius r_i overlaps multiple data partitions, point q_i is replicated into these overlapping partitions, and another set of

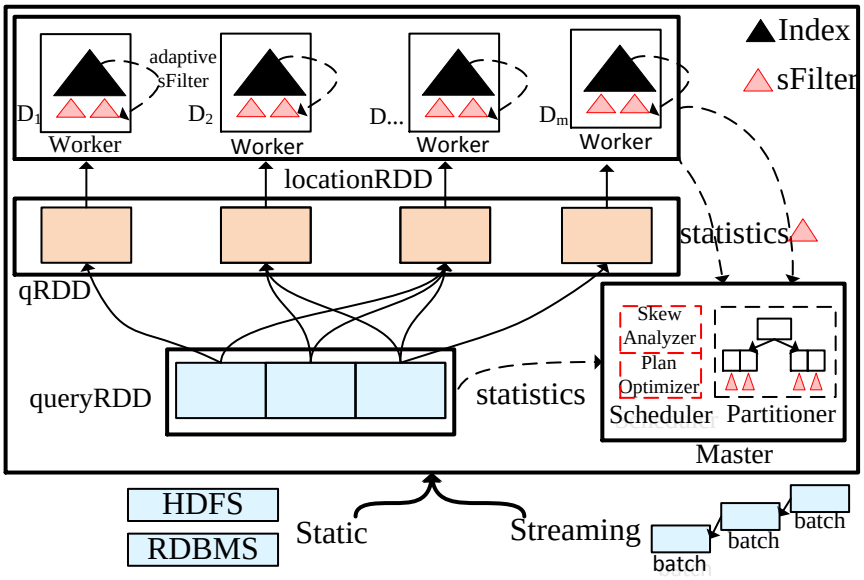


Figure 4.2.: Architecture of LOCATIONSPARK

kNN candidates is computed in each of these partitions. Finally, we merge the kNN candidates from the various partitions to get the exact result. For example, in Figure 4.1(b), assume that we want to evaluate a 3NN query for Point q_6. The first step is to find the 3NN candidates for q_6 in data Partition D_3. Next, we find that the radius r for the 3NN candidates from Partition D_3 overlaps Partition D_4. Thus, we need to compute the 3NN of q_6 in Partition D_4 as well. Notice that the radius r can enhance the 3NN search in Partition D_4 because only the data points within Radius r are among the 3NN of q_6. Finally, the 3NN of q_6 are o_1, o_2 and o_3.

4.2.3 Challenges

The outer and inner tables (or, in shared execution terminology, the queries and the data) are spatially collocated in distributed spatial computing. In the following discussion, we refer to the outer table as being the queries table, e.g., containing the ranges of range operations, or the focal points of kNN operations. We further assume that the outer (or queries) table is the smaller of the two, in contrast to the inner table that we refer to by the

data table (in the case of shared execution of multiple queries together). The distribution of the incoming spatial queries (in the outer tables) changes dynamically over time, with bursts in certain spatial regions. Thus, evenly distributing the input data D to the various workers may result in load imbalance at times. LOCATIONSPARK's skew analyzer identifies the skewed data partitions based on a cost model and then repartitions and redistributes the data accordingly. The plan optimizer selects the optimal repartitioning strategies for both the outer and inner tables, and consequently generates an overall optimized execution plan.

Communication cost is a major factor that affects system performance. LOCATIONSPARK adopts a spatial Bloom filter to reduce network communication cost. The spatial Bloom filter's role is to prune the data partitions that overlap the spatial ranges from the outer tables but do not contribute to the final operation's results. This spatial Bloom filter is memory-based and is space- and time-efficient. The spatial Bloom filter adapts its structure as the data and query distributions change.

4.3 Query Plan Scheduler

This section addresses how to dynamically handle query (outer table) skew. First, we present the cost functions for query processing and analyze the bottlenecks. Then, we show how to repartition the skewed data partitions to speedup processing. This is formulated as an optimization problem that we show is NP-complete. Thus, we introduce an approximation algorithm to solve the skew data repartitioning problem. Although presented for spatial range joins, the proposed technique applies to kNN join as well.

4.3.1 Cost Model

The input dataset D (i.e., inner table of spatial range join) is distributed into N data partitions, and each data partition D_i is indexed and cached in memory. For the query dataset Q (i.e., outer table of spatial range join), each query $q_i \in Q$ is shuffled to the data partitions that spatially overlap with it. The shuffling cost is denoted by $\epsilon(Q, N)$. The execution time of local queries at Partition D_i is $\gamma(D_i) = \gamma(|D_i|, |Q_i|)$, where $|D_i|$ and

$|Q_i|$ are the number of data points and queries at Partition D_i. The execution times of local queries depend on the queries and built indexes, and the estimation of $\gamma(D_i)$ is presented later. After the local results are computed, the postprocessing step merges these local results to produce the final output. The corresponding cost is denoted by $\rho(Q)$. Overall, the runtime cost for the Spatial-Range-Join operation is:

$$C(D, Q) = \epsilon(Q, N) + \max_{i \in [1,N]} \gamma(D_i) + \rho(Q), \tag{4.1}$$

where N is the number of data partitions. In reality, the cost of query shuffling is far less than the other costs as the number of queries is much smaller than the number of data items. Thus, the runtime cost can be estimated as follows:

$$C(D, Q) = \max_{i \in [1,N]} \gamma(D_i) + \rho(Q) \tag{4.2}$$

In Equation 4.2, data partitions are categorized into two types: skewed ($\hat{D}$) and non-skewed $\bar{D}$. The execution time of the local queries in the skewed partitions is the bottleneck. The runtime costs for skewed and non-skewed data partitions are $\max_{i \in [1,\hat{N}]} \gamma(\hat{D}_i)$ and $\max_{j \in [1,\bar{N}]} \gamma(\bar{D}_j)$, respectively, where $\hat{N}$ (and $\bar{N}$) is the number of skewed (and non-skewed) data partitions, and $N = \hat{N} + \bar{N}$. Thus, Equation 4.2 can be rewritten as follows:

$$C(D, Q) = \max\{ \max_{i \in [1,\hat{N}]} \gamma(\hat{D}_i), \max_{j \in [1,\bar{N}]} \gamma(\bar{D}_j)\} + \rho(Q) \tag{4.3}$$

4.3.2 Execution Plan Generation

The goal of the query optimizer is to minimize the query processing time subject to the following constraints: (1) the limited number of available resources (i.e., the number of partitions) in a cluster, and (2) the overhead of network bandwidth and disk I/O. Given the partitioned and indexed spatial data, the cost estimator for query processing based on sampling that we introduce below, and the available number of data partitions, the optimizer returns an execution plan that minimizes query processing time. First, the optimizer needs to determine if any partitions are skewed. Then, it repartitions them subject to the introduced cluster and networking constraints. Then, the optimizer evaluates the plan given

the new repartitioned data to determine whether it minimizes query execution time or not (Refer to Figure 4.3, where the red lines identify local operations, and black lines show the data partitioning. D_s and D_{ns} are the skew and non-skew data partitions, respectively. Queries Q are partitioned into skew Q_s and non skew Q_{ns} in Stage 1. Stages 2 and 3 execute spatial queries independently. Finally, Stage 4 merges the results.).

Estimating the runtime cost for executing the local queries and the cost of merging the final results is not straightforward. The local query processing time $\gamma(D_i)$ is influenced by various factors including the types of spatial indexes used, the number of data points in D_i, the number of queries directed to D_i, related spatial regions and the available memory. Similar to [73], we assume that the related cost functions are monotonic, and they can be approximated using samples from the dataset and the queries. Thus, the local query execution time is formulated as follows: $\gamma(D_i) = \gamma(\tilde{D}_i, \tilde{Q}_i, \alpha, B)$ where $\tilde{D}_i$ is a sample of the original dataset, $\tilde{Q}_i$ is the sample of the queries, B is the area of the underlying spatial region, and α is the sample ratio to scale up the estimate to the entire dataset. After computing a representative sample of the data points and queries via sampling technique, e.g., using reservoir sampling [55], the cost function $\gamma(D_i)$ estimates the query processing runtime in data partition D_i. More details on computing $\gamma(D_i)$, $\rho(Q_i)$, and the sample size can be learned from previous work [73].

Given the estimated runtime cost over skewed and non-skewed partitions, the optimizer splits one skewed data Partition $\hat{D}_i$ into m' data sub-partitions. Assume that $\hat{Q}_i$ is the set of queries originally assigned to Partition $\hat{D}_i$. Let the overheads due to data shuffling, re-indexing, and merging be $\beta(\hat{D}_i)$, $\kappa(\hat{D}_i)$ and $\rho(\hat{Q}_i)$, respectively. Thus, after splitting a skewed Partition $\hat{D}_i$, the new runtime is:

$$\widehat{\gamma(\hat{D}_i)} = \beta(\hat{D}_i) + \max_{s \in [1,m']} \{\gamma(D_s) + \kappa(D_s)\}\} + \rho(\hat{Q}_i), \tag{4.4}$$

Hence, we can split one skewed Partition $\hat{D}_i$ into multiple partitions only if $\widehat{\gamma(\hat{D}_i)} < \gamma(\hat{D}_i)$. As a result, the new query execution time, say $\widehat{C(D,Q)}$, is:

$$\widehat{C(D,Q)} = \max\{\max_{i \in [1,\hat{N}]} \{\widehat{\gamma(\hat{D}_i)}\}, \max_{j \in [1,\bar{N}]} \{\gamma(\bar{D}_j)\}\} + \rho(\bar{Q}) \tag{4.5}$$

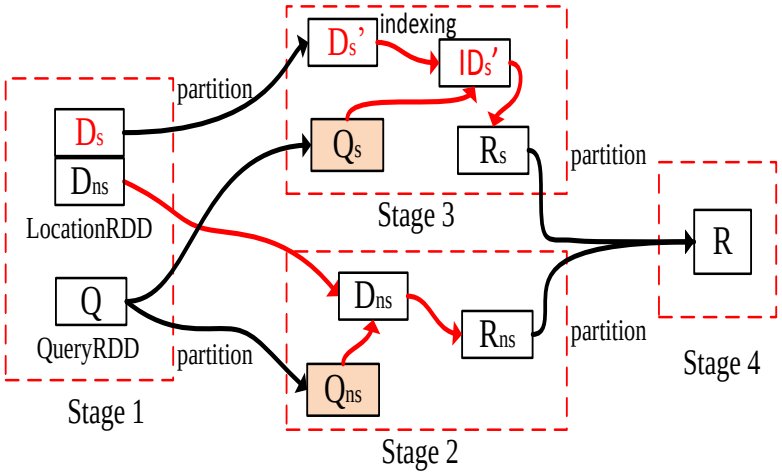


Figure 4.3.: Execution plan for Spatial-Range-Join

Thus, we can formulate the query plan generation based on the data repartitioning problem as follows:

Definition 4.3.1 *Let D be the set of spatially indexed data partitions, Q be the set of spatial queries, M be the total number of data partitions, and their corresponding cost estimation functions, i.e., query processing $\gamma(D_i)$, data repartitioning $\beta(D_i)$, and data indexing cost estimates $\kappa(Q_i)$. The query optimization problem is to choose a skewed Partition $\hat{D}$ from D, repartition each $\hat{D}_i \in \hat{D}$ into multiple partitions, and assign spatial queries to the new data partitions. The new data partition set, say D', contains partitions $D'_1, D'_2, \ldots, D'_k$. s.t. (1) the $\widehat{C(D,Q)} < C(D,Q)$ and (2) $|D'| \leq M$.*

Unfortunately, this problem is NP-complete. In the next section, we present a greedy algorithm for this problem.

Theorem 4.3.1 *Optimized query plan generation with data repartitioning for distributed indexed spatial data is NP-complete.*

The proof is given in [4].

4.3.3 A Greedy Algorithm

The general idea is to search for skew partitions based on their local query execution time. Then, we split the selected data partitions only if the runtime can be improved. If the runtime cannot be improved, or if all the available data partitions are consumed, the algorithm terminates. While this greedy algorithm cannot guarantee optimal query performance, in the experimental section, it shows one order of magnitude improvement over the plan executing over the original partitions. Algorithm 13 gives the pseudocode for the greedy partitioning procedure.

Algorithm 13 includes two functions, namely *numberOfPartitions* and *repartition*. Function *numberOfPartitions* computes the number of partitions m' for splitting one skew partition. Naively, we could split a skew partition into two partitions each time. But this is not necessarily efficient. Given data partitions $D = \{D_1, D_2, \ldots, D_N\}$, let Partition D_1 be the one with the largest local query execution time $\gamma(D_1)$. From Equation 4.2, the execution time is approximated by $\gamma(D_1) + \rho(Q)$. To improve the execution time, Partition D_1 is split into m' partitions, and the query execution time for Partition D_1 is updated to $\widehat{\gamma(D_1)}$. For all other partitions $D_i \in D$ ($i \neq 1$), the runtime is the $\max\{\gamma(D_i)\} + \rho(Q') = \Delta$, where $i = [2, \ldots, N]$ and Q' are the queries related to all data partitions except D_1. Thus, the runtime is $\max\{\Delta, \widehat{\gamma(D_1)}\}$, and is improved if

$$\max\{\Delta, \widehat{\gamma(D_1)}\} < \gamma(D_1) + \rho(Q) \tag{4.6}$$

As a result, we need to compute the minimum value of m' to satisfy Equation 4.6, since Δ, $\gamma(D_1)$, and $\rho(Q)$ are known.

Function *repartition* splits the skewed data partitions and reassigns the spatial queries to the new data partitions using two strategies. The first strategy is to repartition based on the data distribution. Because each data partition D_i is already indexed by a spatial index, the data distribution can be learned directly by recording the number of data points in each branch of the index. Then, we repartition data points in D_i into multiple parts based on the recorded numbers while guaranteeing that each newly generated sub-partition contains an equal amount of data. In the second strategy, we repartition a skewed Partition D_i based on

the distribution of the spatial queries. First, we collect a sample Q_s from the queries Q_i that are originally assigned to partition D_i. Then, we compute how Q_s is distributed in Partition D_i by recording the frequencies of the queries as they belong to branches of the index over the data. Thus, we can repartition the indexed data based on the query frequencies. Although the data sizes may not be equal, the execution workload will be balanced. In our experiments, we choose this second approach to overcome query skew. To illustrate how the proposed query-plan optimization algorithm works, consider the following example.

Running Example. Given data partitions $D = \{D_1, D_2, D_3, D_4, D_5\}$, where the number of data points in each partition is 50, the number of queries in each partition D_i, $1 \leq i \leq 5$ is 30, 20, 10, 10, and 10, respectively, and the available data partitions M is 5. The local query processing cost is $\gamma(D_i) = |D_i| \times |Q_i| \times p_e$, where $p_e = 0.2$ is a constant. The cost of merging the results is $\rho(Q) = |Q| \times \lambda \times p_m$, where $p_m = 0.05$, and $\lambda = 10$ is the approximate number of retrieved data points per query. Data repartitioning indexing costs are $\beta(D_i, m') = |D_i| \times m' \times p_r$, and $\gamma(D_s) = |D_s| \times p_x$, respectively, where $p_r = 0.01$ and $p_x = 0.02$.

Without any optimization, according to Equation 4.2, the estimated runtime cost for this input dataset is 340. LOCATIONSPARK optimizes the query as follows. At first, it chooses data Partition D_1 as the skew partition to be repartitioned because D_1 has the highest local runtime (300), while the second largest cost is D_2's (200). Using Equation 4.6, the number of partitions m' to split D_1 into is 2. Thus, the optimizer splits D_1 into two the partitions D'_1 and D'_2 based on the query distribution within Partition D_1. The number of data points in D'_1 and D'_2 is 22 and 28, respectively, and the number of queries are 12 and 18, respectively. Therefore, the new runtime is reduced to $\approx 200 + 25$ because D_1's runtime is reduced to ≈ 100 based on Equation 4.4. Therefore, the two new data partitions D'_1 and D'_2 are introduced in place of D_1. Next, Partition D_2 is chosen to be split into two partitions, and the optimized runtime is $\approx 100 + 15$. Finally, the procedure exits because there is only one available partition left.

Algorithm 13: Greedy Partitioning Algorithm

Input: D: Indexed spatial data partitions,

Stat: Collected statistics, e.g., the number of data points and queries in data partition D_i,

M: number of available data partitions.

Output: $Plan$: Optimized data and query partition plan, C: estimated query cost

h: Maximum Heap;
inserts D_i into $heap$ // data partitions are ordered by cost $\gamma(D_i)$ that is computed using Stat
$Cost_o \leftarrow \gamma(h.top) + \rho(Q)$ // old execution plan runtime cost
$Plan$
while $M > 0$ **do**
    Var $D_x \leftarrow h$.pop(); //get the partition with maximum runtime cost
    Var $m' \leftarrow$ *numberOfPartitions*(h,D_x, M)
    Var (D_s, PL_s) $\leftarrow$ *repartition*(D_x, m') //split D_x into m' partitions
    $Cost_x \leftarrow \beta(D_x) + \max_{s\in[1,m']}\{\kappa(D_s) + \gamma(D_s)\}\} + \rho(Q_x)$ //updated runtime cost over selected skew partition
    if $Cost_x$¡ $Cost_o$ **then**
        save Partitions D_s into h
        save Partition plan PL_s into $Plan$
        $Cost_o \leftarrow Cost_x$
        $M \leftarrow M$-m'
    end
    else
        break;
    end
end

4.4 Local Execution

Once the query plan is generated, each computation node chooses a specific local execution plan based on the queries assigned to it and the indexes it has. We implement various centralized algorithms for spatial range join and kNN join operators within each

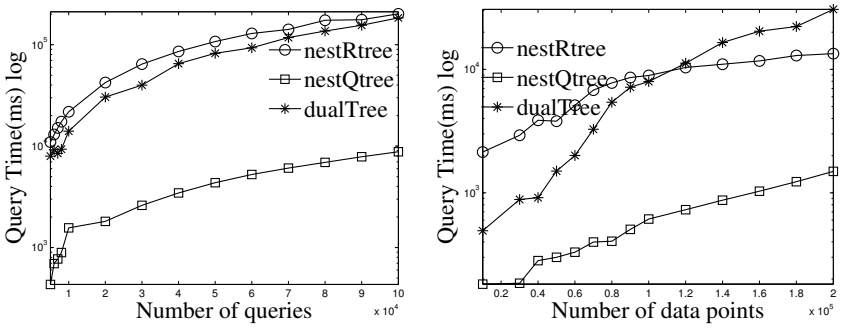
worker and study their performance. The algorithms are implemented in Spark. We use the execution time as the performance measure.

4.4.1 Spatial-Range-Join

Two algorithms for spatial range join [74] are implemented. The first is indexed nested-loops join, where we probe the spatial index repeatedly for each outer tuple (or range query in the case of shared execution). The tested algorithms are nestRtree, nestGrid and nestQtree, where they use an R-tree, a Grid, and a Quadtree as index for the inner table, respectively.

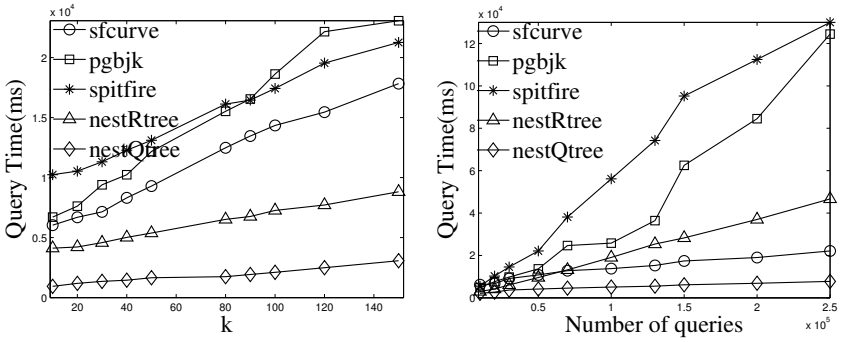
The second algorithm for spatial range join is based on the dual-tree traversal [75]. It builds two spatial indexes (e.g., an R-tree) over both the input queries (i.e., the outer table) and the data (i.e, the inner table), and performs a depth-first search over the dual trees simultaneously.

Figure 4.4(a) gives the performance of nestRtree, nestQtree, and dual-tree, where the number of data points in each workers is set to 0.3 million. We do not show the results for nestGrid, since it always shows the worst performance. The dual-tree approach provides a 1.8x speedup over the nestRtree. This conforms with other published results [74]. nestQtree achieves an order of magnitude improvement over the dual-tree approach. The reason is that the minimum bounding rectangles (MBRs) of the spatial queries overlap with multiple MBRs in the data index, and this reduces the pruning power of the underlying R-tree. The same trend is observed when increasing the number of indexed data points (see Figure 4.4(b)). The dual-tree approach outperforms nestRtree, when the number of data points is smaller than 120k. However, dual-tree slows down afterwards. In this experiment, we only show the results for indexing over two dimensional data points. However, Quadtree performs worst when the indexed data are polygons [76]. Overall, for multidimensional points, the local planner chooses nestQtree as the default approach. For complex geometric types, the local planner uses the dual-tree approach based on an R-tree implementation.



(a) Effect of outer table size (b) Effect of inner table size

Figure 4.4.: Evaluation of local Spatial-Range-Join algorithms



(a) Effect of k (b) Effect of inner table size

Figure 4.5.: Evaluation of local kNN-Join algorithms

4.4.2 kNN Join

Similar to Spatial-Range-Join, indexed nested-loops can be applied to kNN join, and it computes the set of kNN objects for each query point in the outer table where an index is built on the inner table (the data table). The other kinds of kNN join algorithms are block-based. They partition the queries and the data points into different blocks, and find the kNN candidates for queries in the same block. Then, a post-processing refine step computes kNN for each query point in the same block. Gorder [77] divides query and data points into different rectangles based on G-ordered data ordering, and utilizes two distance bounds to reduce visiting unnecessary blocks. For example, the min-distance bound is the minimum distance between the rectangles of the query points and the data points. The max-distance bound is the maximum distance from the queries to their kNN sets. If the max-distance is smaller than the min-distance bound, the related data block is pruned. PGBJ [78] has a similar idea that extends to parallel kNN join using MapReduce. Recently, Spitfire [79] is a parallel kNN self-join algorithm for in-memory data. It replicates the possible kNN candidates into its neighboring data blocks. Both PGBJ and Spitfire are designed for parallel kNN join, but they are not directly applicable to indexed data. The reason is that PGBJ partitions queries and data points based on the selected pivots while Spitfire is specifically optimized for kNN self-join.

LOCATIONSPARK enhances the performance of the local kNN join procedure. For the Gorder [77], instead of using principal component analysis (PCA) in Gorder, which is expensive, we apply the Hilbert curve to partition the query points. We term the modified Gorder method *sfcurve*. We modify PGBJ as follows. First, we compute the pivots of the query points based on a clustering algorithm (e.g., k-means) over sample data, and then partition the query points are into different blocks based on the computed pivots. Next, we compute the MBR of each block. Because the data points are already indexed (e.g., using an R-tree), the min-distance from the MBRs of the query points and the index data is computed, and the max-distance bound is calculated based on the kNN results from the

pivots. This approach is termed *pgbjk*. In terms of spitfire, we use a spatial index to speedup finding the kNN candidates.

Figure 4.5(a) gives the performance of the specialized kNN join approaches within alocal computation node when varying k from 10 to 150. The nestQtree approach always performs the best, followed by nestRtree, sfcurve, pgbjk, and spitfire. Notice that block-based approaches induce extensive amounts of kNN candidates for query points in the same block, and it directly degrades the performance of the kNN refine step. More importantly, the min-distance bound between the MBR of the query points and the MBR of the data points is much smaller than the max-distance boundary. Then, most of the data blocks cannot be pruned, and result in redundant computations. On the other hand, the nested-loops-based join algorithms prune some data blocks, because the min-distance bound from the query point to the data points in the same block is bigger than the max-distance boundary of this query point. Figure 4.5(b) gives the performance of the kNN join algorithms by varying the number of query points. Initially, for small numbers of query points (less than 70k), nestRtree outperforms sfcurve, then nestRtree degrades linearly with more query points. Overall, we adopt nestQtree as the local kNN join algorithm.

4.5 Spatial Bloom Filter

In this section, we introduce a new spatial Bloom filter termed *sFilter*. The sFilter can help decide for an outer tuple, say q, of a spatial range join, if there exist tuples in the inner table that actually join with q. This helps reduce the communication overhead. For example, consider an outer tuple q of a spatial range join where q has a range that overlaps multiple data partitions of the inner table. Typically, all the overlapping partitions need to be examined by communicating q's range to them, and searching the data within each partition to test for overlap with q's range. This incurs high communication and search costs. Using the sFilter, given q's range that overlaps multiple partitions of the inner table, the sFilter can decide which overlapping partitions contain data that overlaps q's range without actually communicating with and searching the data in the partitions. Only the

partitions that contain data that overlap with q's range are the ones that will be contacted and searched.

In the remaining of this section, we introduce the data structures for the sFilter and its corresponding range search algorithm. Next, we describe how sFilter reduces unnecessary communication costs as the scheduler assigns a query to local computation nodes. Finally, we present an approach to update the sFilter adaptively when the distribution of data and queries change over time.

4.5.1 Overview of sFilter

Figure 4.6 gives an example of an sFilter. The sFilter is motivated by in-memory indexes [80, 81]. Conceptually, an sFilter is a quadtree variant that has internal and leaf nodes [76]. Internal nodes are for index navigation, and leaf nodes, each has a marker to indicate whether or not there are data items in the node's corresponding region. We encode the sFilter into two binary codes and execute queries over this encoding.

Binary Encoding of the sFilter

The sFilter is encoded into two long sequences of bits. The first bit-sequence corresponds to internal nodes while the second bit-sequence corresponds to leaf nodes. Notice that in these two binary sequences, no pointers are needed. Each internal node of the sFilter takes four bits, where each bit represents one of the internal node's children. These children are encoded in clock time order. The bit value of an internal node determines the type of its child nodes, i.e., whether the child is internal (a 1 value) or leaf (a 0 value). For example, the root node A in Figure 4.6 has binary code 1011 (in clock time order) that means it has three of its children being internal nodes, and one is a leaf node (the second node). The four-bit encodings of all the internal nodes are concatenated together to form the internal-node bit-sequence of the sFilter. The ordering of the internal nodes in this sequence is based on a breadth-first search traversal of the quadtree. In contrast, a leaf node only takes one bit, and its bit value indicates whether or not data points exist inside the

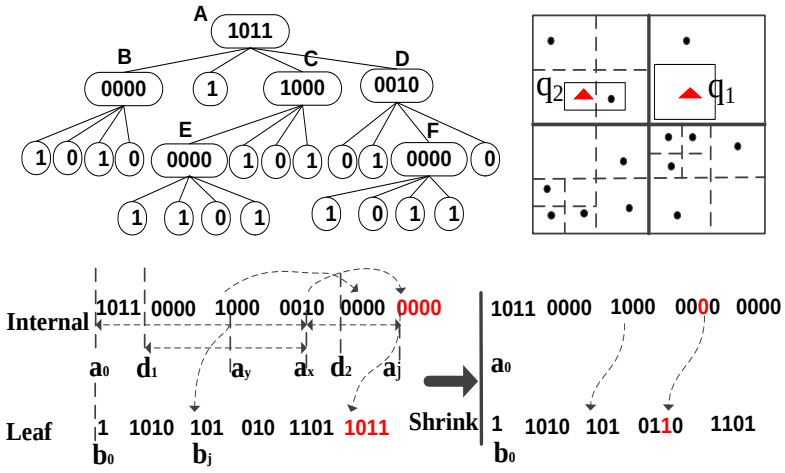


Figure 4.6.: sFilter structure (up left), the data (up right) and binary encoding of sFilter (down)

spatial quadrant corresponding to the leaf. For example, internal node B in Figure 4.6 has four children, and the bit values for B's leaf nodes are 1010 that indicates that the first and third leaf nodes of B (in clock time order) contain data items.

To encode the bit-sequence for all the leaf nodes in an sFilter, during the same Breadth-First Search on the underlying quad-tree of the sFilter to produce the bit-sequence for the internal nodes, we simultaneously construct the bit-sequence for all the leaf nodes. Thus, we produce two encodings, one for a BFS of the internal nodes and one for a BFS of the leaf nodes. For example, the sFilter in Figure 4.6 is encoded into the two binary sequences given in the figure. From the example, the two bit-sequences for the internal and leaf nodes of the sFilter are $\{1011\ 0000\ 1000\ 0010\ 0000\ 0000\}$ and $\{1\ 1010\ 101\ 010\ 1101\ 1011\}$, respectively. Thus, the space usage of the sFilter is bounded by the number of internal and leaf nodes. Formally, the space usage of an sFilter is $O(((4^{d-1}-1)/3) \times 4 + 4^{d-1})$ Bits, where $O(4^{d-1}-1)/3)$ and $O(4^{d-1})$ are the numbers of internal nodes and leaf nodes, respectively, and d is the depth of quadtree equal with $o(\log(n))$. In the next section, we demonstrate how to execute a spatial range query over the bit-sequences of the sFilter.

Query processing

Consider internal node D in Figure 4.6. D's binary code is 0010, and the third bit has a value of 1 at memory address a_x of the internal nodes bit sequence. Thus, this bit refers to D's child F that is also an internal node at address a_j. Because the sFilter has no pointers, we need to compute F's address a_j from a_x. Observe that the number of bits with value 1 from the start address a_0 of the binary code to a_x can be used to compute the address.

Definition 4.5.1 *Let a be the bit sequence that starts at address a_0. $\chi(a_0, a_x)$ and $\tau(a_0, a_x)$ are the number of bits with value 1 and 0, respectively, from addresses a_0 to a_x inclusive.*

$\chi(a_0, a_x)$ is the number of internal nodes up to a_x. Thus, the address a_j of F is $(a_0 + 5 \times 4)$ because there are 5 bits with value 1 from a_0 to a_x. Similarly, if one child node is a leaf node, its address is inferred from $\tau(a_0, a_x)$ as follows:

Proposition 4.5.1 *Let a and b be the sFilter's bit sequences for internal and leaf nodes in memory addresses a_0 and b_0, respectively. To access a node's child in memory, we need to compute its address. The address, say a_j, of the xth child of an internal node at address a_x is computed as follows. If the bit value of a_x is 1, then $a_j = a_0 + 4 \times \chi(a_0, a_x)$. If the bit value of a_x is 0, $a_j = b_0 + \tau(a_0, a_x)$.*

We adopt the following two optimizations to speedup the computation of $\chi(a_0, a_x)$ and $\tau(a_0, a_x)$: (1) Precomputation, and (2) Set counting. Let d_i be the memory address of the first internal node at height (or depth) i of the underlying quadtree when traversed in BFS order. For example, in Figure 4.6, nodes B and E are the first internal nodes in BFS order at depths 1 and 2 of the quadtree, respectively. For all $i \leq$ depth of the underlying quadtree, we precompute $\chi(a_0, d_i)$, e.g., $\chi(a_0, d_1)$ and $\chi(a_0, d_2)$ in Figure 4.6. Notice that $d_0 = a_0$ and $\chi(a_0, d_0) = 0$. Then, address a_j that corresponds to the memory address of the xth child of an internal node at address a_x can be computed as follows. $a_j = a_0 + (\chi(a_0, d_1) + \chi(d_1, a_x)) \times 4$. $\chi(a_0, d_1)$ is precomputed. Thus, we only need to compute on the fly $\chi(d_1, a_x)$. Furthermore, evaluating χ can be optimized by a bit set

counting approach, i.e, a lookup table or a sideways addition [1] that can achieve constant time complexity.

After getting one node's children via Proposition 4.5.1, we apply Depth-First Search (DFS) over the binary codes of the internal nodes to answer a spatial range query. The procedure starts from the first four bits of bit sequence a, since these four bits are the root node of the sFilter. Then, we check the four quadrants, say r_s, of the children of the root node, and iterate over r_s to find the quadrants, say r'_s, overlapping the input query range q_i. Next, we continue searching the children of r'_s based on the addresses computed from Proposition 4.5.1. This recursive procedure stops if a leaf node is found with value 1, or if all internal nodes are visited. For example, Consider range query q_2 in Figure 4.6. We start at the root node A (with bit value 1011). Query q_2 is located inside the northwestern (NW) quadrant of A. Because the related bit value for this quadrant is 1, it indicates an internal node type and it refers to child node B. Node B's memory address is computed by $a_0 + 1 \times 4$ because only one non-leaf node (A) is before B. B's related bit value is 0000, i.e., B contains four leaf nodes. The procedure continues until finding one leaf node of B, mainly the southeastern child leaf node, with value 1 that overlaps the query, and thus returns true.

4.5.2 sFilter in LocationSpark

The depth of the sFilter affects query performance. It is impractical to use only one sFilter in a distributed setting. We embed multiple sFilters into the global and local spatial indexes in LOCATIONSPARK. In the master node, separate sFilters are placed into the different branches of the global index, where the role of each sFilter is to locally answer the query for the specific branch it is in. In the local computation nodes, an sFilter is built and it adapts its structure based on data updates and changes in query patterns.

[1] `https://graphics.stanford.edu/~seander/bithacks.html`

Algorithm 14: Update sFilter in LocationSpark

Input: $LocationRDD$: Distributed/indexed spatial data,
Q: Input set of spatial range queries

Output: R: Results of the spatial queries

```
1  Var index ← LocationRDD.index //get global index with embedded sFilters
2  Var qRDD ← partition(Q,index) // Distribute in parallel the input spatial queries using the
   global index
3  Var update_sFilter ← //function for updating the sFilter in each worker
4  {
5  for each query q_i in this worker do
6      if query q_i's return result is empty then
7          insert(q_i, sFilter) // adapt sFilter given q_i
8      end
9  end
10 if sFilter.space > α then
11     shrink(sFilter) // shrink the sFilter to save space
12 end
13 }
14 R ← LocationRDD.sjoin(qRDD)(update_sFilter) //execute spatial join and update sFilter
   in workers
15 Var sFilters ← LocationRDD.collect_sFilter() //collect sFilter from workers
16 mergesFilters(sFilters, index) // update sfilter in global index
17 return R
```

Spatial Query Processing Using the sFilter

Algorithm 14 gives the procedure for performing the spatial range join using the sFilter. Initially, the outer (queries) table is partitioned according to the global index. The global index identifies the overlapping data partitions for each query q. Then, the sFilter tells which partitions contain data that overlap the query range (Line 2 of the algorithm). After

performing the spatial range join (Line 14), the master node fetches the updated sFilter from each data worker, and refreshes the existing sFilters in the master node (Lines 15-16). Lines 2-13 update the sFilter of each worker (as in Figure 4.2).

The sFilter can improve the kNN search and kNN join because they also depend on spatial range search. Moreover, their query results may enhance the sFilter by lowering the false positive errors as illustrated below.

Query-aware Adaptivity of the sFilter

The build and update operations of the sFilter are first executed at the local workers in parallel. Then, the updated sFilters are propagated to the master node.

The initial sFilter is built from a temporary local quadtree [76] in each partition. Then, the sFilter is adapted based on the query results. For example, consider Query q_1 in Figure 4.6. Initially, the sFilter reports that there is data for q_1 in the partitions. When q_1 visits the related data partitions, it finds that there are actually no data points overlapping with q_1 in the partitions, i.e., a false-positive (+ve) error. Thus, we mark the quadrants precisely covered by q_1 in the sFilter as empty, and hence reduce the false positive errors if queries visit the marked quadrants again. Function `insert` in Algorithm 14 recursively splits the quadrants covered by the empty query, and marks these generated quadrants as empty. After each local sFilter is updated in each worker, these updates are reflected into the master node. The compact encoding of the sFilter saves the communication cost between the workers and the master.

However, the query performance of the sFilter degrades as the size of the index increases. Function `shrink` in Algorithm 14 merges some branches of the sFilter at the price of increasing false +ve errors. For example, one can shrink internal node F in Figure 4.6 into a leaf node, and updating its bit value to 1, although one quadrant of F does not contain data. Therefore, we might track the visit frequencies of the internal nodes, and merge internal nodes with low visiting frequency. Then, some well-known data caching policies, e.g., LRU or MRU, can be used. However, the overhead to track the visit frequen-

cies is expensive. In our implementation, we adopt a simple bottom-up approach. We start merging the nodes from the lower levels of the index to the higher levels until the space constraint is met. In Figure 4.6, we shrink the sFilter from internal node F, and replace it by a leaf node, and update its binary code to 1. F's leaf children are removed. The experimental results show that this approach increases the false +ve errors, but enhances the overall query performance.

4.6 Experimental Study

LOCATIONSPARK is implemented on RDDs, which are the distributed memory abstraction in Spark. LOCATIONSPARK is a library of Spark and provides Class LocationRDD to conduct spatial operations [4]. Statistics are maintained at the driver program of Spark, and the execution plans are generated at the driver. Local spatial indexes are persisted in the RDD data partitions, while the global index is realized by extending the interface of the RDD data partitioner. The data tuples and related spatial indexes are encapsulated into the RDD data partitions. Thus, Spark's fault tolerance naturally applies in LOCATIONSPARK. The spatial indexes are immutable and are implemented based on the path copy approaches. Thus, each updated version of the spatial index can be persisted into disk for fault tolerance. This enables the recovery of a local index from disk in case of failure in a worker. The Spark cluster is managed by YARN, and a failure in the master nodes is detected and managed by ZooKeeper. In case of master node failure, the lost master node is evicted and a standby node is chosen to recover the master. As a result, the global index and the sFilter in the master node are recoverable. Finally, the built spatial index data can be stored into disk, and enable further data analysis without additional data repartitioning or indexing. LOCATIONSPARK is open-source, and can be downloaded from `https://github.com/merlintang/SpatialSpark`.

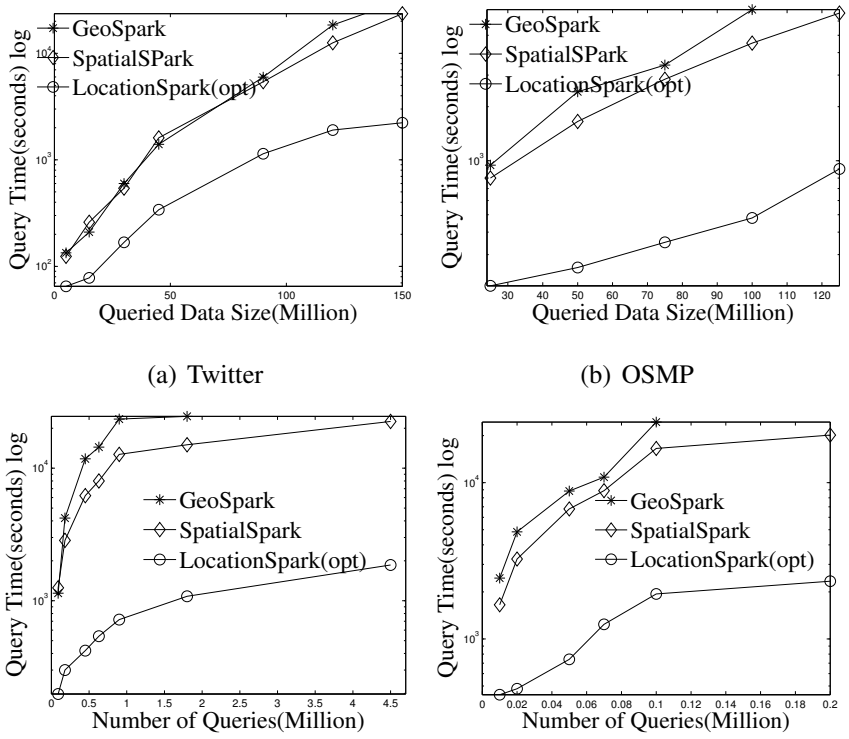
4.6.1 Experimental Setup

Experiments are conducted on two datasets. **Twitter**: 1.5 Billion Tweets (around 250GB) are collected over a period of nearly 20 months (from January 2013 to July 2014) and is restricted to the USA spatial region. The format of a tweet is: identifier, timestamp, longitude-latitude coordinates, and text. **OSMP**: is shared by the authors of SpatialHadoop [14]. OSMP represents the map features of the whole world, where each spatial object is identified by its coordinates (longitude, latitude) and an object ID. It contains 1.7 Billion points with a total size of 62.3GB. We generate two types of queries. (1) Uniformly distributed (USA, for short): We uniformly sample data points from the corresponding dataset and generate spatial queries from the samples. These are the default queries in our experiments. (2) Skewed spatial queries: These are synthesized around specific spatial areas, e.g., Chicago, San Francisco, New York (CHI, SF, NY, correspondingly, for short). The spatial queries and data points are the outer table Q and the inner table D for the experimental studies of the spatial range and kNN joins presented below.

Our study compares LOCATIONSPARK with the following: (1) **GeoSpark** [70] uses ideas from SpatialHadoop but is implemented over Spark. (2) **SpatialSpark** [69] performs partition-based spatial joins. (3) **Magellan** [71] is developed based on dataframe of Spark to benefit from Spark SQL's plan optimizer. However, Magellan does not have spatial indexing. (4) **State-of-art kNN-join**: Since none of the three systems support kNN join, we compare LOCATIONSPARK with a state-of-art kNN-join approach (PGBJ [78]) that is provided by PGBJ's authors. LocationSpark(opt) refers to the optimized query scheduler and sFilter.

We use a cluster with six physical nodes, namely Hathi [2].Hathi consists of 6 Dell compute nodes with two 8-core Intel E5-2650v2 CPUs, 32 GB of memory, and 48TB of local storage per node for a total cluster capacity of 288TB. The Spark version is 1.5.0 with Yarn cluster resource management. The performance is mainly measured by the average query execution time for various spatial operators.

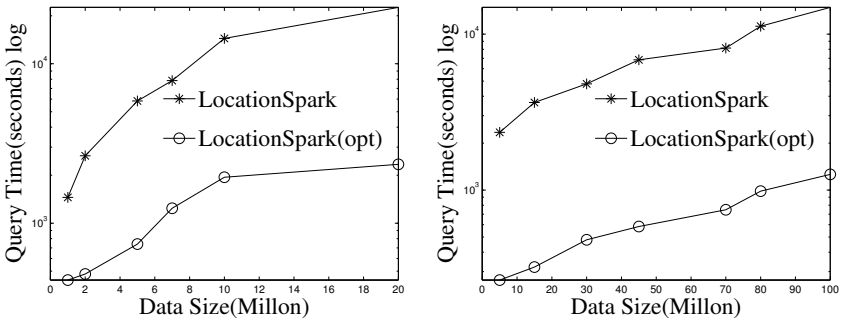
[2]https://www.rcac.purdue.edu/compute/hathi/



(a) Twitter (b) OSMP

(c) Twitter (d) OSMP

Figure 4.7.: The performance of Spatial-Range-Join



(a) Twitter (b) OSMP

Figure 4.8.: Performance of kNN-Join by increasing the number of data points

Table 4.1.: Comparison with the spatial range search

Dataset	System	Query time(ms)	Index build time(s)
Twitter	LocationSpark(R-tree)	390	32
	LocationSpark(Qtree)	**301**	16
	Magellan	15093	/
	SpatialSpark	16874	35
	SpatialSpark(no-index)	14741	/
	GeoSpark	4321	45
OSMP	LocationSpark(R-tree)	1212	67
	LocationSpark(Qtree)	**734**	18
	Magellan	41291	/
	SpatialSpark	24189	64
	SpatialSpark(no-index)	17210	/
	GeoSpark	4781	87

4.6.2 Performance of Spatial Range Search and Join

Table 4.1 summarizes the spatial range search and spatial index build time by the various approaches. For a fair comparison, we cache the indexed data into memory, and record the spatial range query processing time. From Table 4.1, observe the following: (1) LOCATIONSPARK is 50 times better than Magellan on query execution time for the two datasets, mainly because the spatial index (e.g., Global and Local index) of LOCATIONSPARK can avoid visiting unnecessary data partitions. (2) LOCATIONSPARK with different local indexes, e.g., the R-tree and Quadtree, outperforms SpatialSpark. The speedup is around 50 times, since SpatialSpark (without index) has to scan all the data partitions. SpatialSpark(with index) stores the global indexes into disk, and finds data partitions by scanning the global index in disk. This incurs extra I/O overhead. Also, the local index is not utilized during local searching. (3) LOCATIONSPARK is around 10 times faster than GeoSpark in spatial range search execution time because GeoSpark does not utilize the built global indexes and scans all data partitions. (4) The local index with Quadtree for LOCATIONSPARK achieves superior performance over the R-tree one in term of index construction and query execution time as discussed in Section 4.4. 5) The index build time among the three systems is comparable because they all scan the data points, which is the dominant factor, and then build the index in memory.

Performance results (mainly, the execution times of the spatial range join) are listed in Figure 4.7. For fair comparison, the runtime is counted as end to end, which includes the time to initiate the job, build indexes, execute the join query, and save results into HDFS. Performance results for Magellan are not shown because it performs Cartesian product and hence has the worst execution time. Figures 4.7(a) and 4.7(b) present the results by varying the data sizes of D (the inner table) from 25 million to 150 million, while keeping the size of Q (the outer table) to 0.5 million. The execution time of GeoSpark shows quadratic increase as the data size increases. GeoSpark's running time is almost 3 hrs when the data size is 150 million, which is extremely slow. SpatialSpark shows similar trends. The reason is that both GeoSpark and SpatialSpark suffer from (1) the spatial skew issue where

some workers process more data and take longer time to finish. (2) the local execution plan based on the R-tree and the Grid is slow. (3) processing of queries go to data partitions that do not contribute to the final results. LOCATIONSPARK with the optimized query plans and the sFilter outperforms the two other systems by an order of magnitude. A detailed analysis for this speedup is presented below. Also, we study the effect of the outer table size on performance. Figures 4.7(c) and 4.7(d) give the run time, and demonstrate that LOCATIONSPARK is 10 times faster than the other two systems.

4.6.3 Performance of kNN Search and Join

Performance of kNN search is listed in Table 4.2. LOCATIONSPARK outperforms GeoSpark by an order of magnitude. GeoSpark broadcasts the query points to each data partition, and accesses each data partition to get the kNN set for the query. Then, GeoSpark collects the local results from each partition, then sorts the tuples based on the distance to query point of kNN. This is prohibitively expensive, and results in large execution time. LOCATIONSPARK only searches for data partitions that contribute to the kNN query point based on the global and local spatial indexes and the sFilter. It avoids redundant computations and unnecessary network communication for irrelevant data partitions.

For kNN join, Table 4.3 presents the performance results when varying k on the Twitter and OSMP datasets. In terms of runtime, LOCATIONSPARK with optimized query plans and with the sFilter always performs the best. LOCATIONSPARK without any optimizations gives better performance than that of PGBJ. The reason is due to having in-memory computations and avoiding expensive disk I/O when compared to MapReduce jobs. Furthermore, LOCATIONSPARK with optimization shows around 10 times speedup over PGBJ, because the optimized plan migrates and splits the skewed query regions.

We test the performance of the kNN join operator when increasing the number of data points while having the number of queries fixed to 1 million around the Chicago area. The results are illustrated in Figure 4.8. Observe that LOCATIONSPARK with optimizations performs an order of magnitude better than the basic approach. The reason is that the

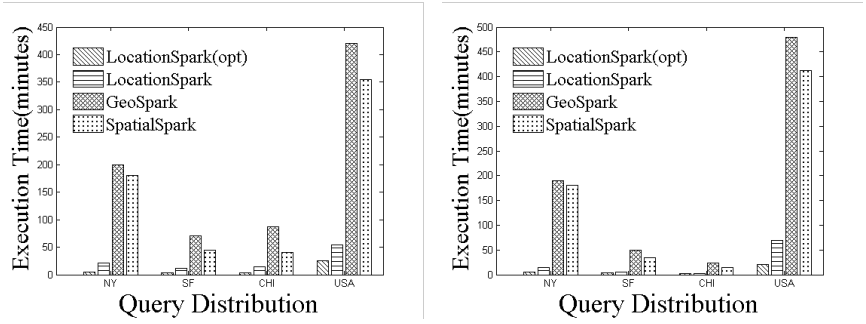
optimized query plan identifies and repartitions the skew partitions. In this experiment, the top five slowest tasks in LOCATIONSPARK without optimization take around 33 minutes, while more than 75% tasks only take less than 30 seconds. On the other hand, with an optimized query plan, the top five slowest tasks take less than 4 minutes. This directly enhances the execution time.

Table 4.2.: Runtime of kNN search with microseconds unit

Dataset	System	k=10	k=20	k=30
Twitter	LocationSpark(R-tree)	81	82	83
	LocationSpark(Q-tree)	74	75	74
	GeoSpark	1334	1813	1821
OSMP	LocationSpark(R-tree)	183	184	184
	LocationSpark(Q-tree)	73	73	74
	GeoSpark	4781	4947	4984

Table 4.3.: Runtime of kNN-Join with second unit

Dataset	System	k=50	k=100	k=150
Twitter	LocationSpark(Q-tree)	340	745	1231
	LocationSpark(Opt)	165	220	230
	PGBJ	3422	3549	3544
OSMP	LocationSpark(Q-tree)	547	1241	1544
	LocationSpark(Opt)	260	300	340
	PGBJ	5588	5612	5668



(a) Twitter (b) OSMP

Figure 4.9.: Performance of Spatial-Range-Join on various query distribution

4.6.4 Effect of Query Distribution

We study the performance under various query distributions. As illustrated before, the query execution plan is the most effective factor in distributed spatial computing. From the experimental results for spatial range join and kNN join above, we already observe that the system with optimization achieves much better performance over the unoptimized versions. In this experiment, we study the performance of the optimized query scheduling plan in LOCATIONSPARK under various query distributions. The performance of the spatial range join operator over query set Q (the outer table) and dataset D (the inner table) is used as the benchmark. The number of tuples for D is fixed as 15 million and 50 million for Twitter and OSMP, respectively, while the size of Q is 0.5 million, and each query in Q is generated from different spatial regions, e.g., CHI, SF, NY and USA. We do not plot the runtime of Magellan on spatial join, as it uses Cartesian join, and hence has the worst performance. Figure 4.9 gives the execution runtimes for the spatial range join operators in different spatial regions. From Figure 4.9, GeoSpark performs the worst, followed by SpatialSpark and then LOCATIONSPARK. LOCATIONSPARK with the optimized query plan achieves an order of magnitude speedup over GeoSpark and SpatialSpark in terms of execution time. LOCATIONSPARK with optimized plans achieves more than 10 times speedup over LOCATIONSPARK without the optimized plans for the skewed spatial queries.

4.6.5 Effect of sFilter

In this experiment, we measure the query processing time, the index construction time, the false positive ratio and the space usage of the sFilter. Table 4.4 gives the performance of various indexes in a local computation node. The Bloom filter is tested using breeze [3]. The sFilter(ad) represents the sFilter with adaptation of its structure given changes in the queries, and with the merging to reduce its size as introduced in Section 4.5.2. From Table 4.4, observe that sFilter achieves one and two orders of magnitude speedup over the Quadtree- and R-tree-based approaches in terms of spatial range search execution time, respectively. The sFilter(ad) improves the query processing time over the approach without optimization, but the sFilter(ad) has the overhead to merge branches of the index to control its size, and increases the false positive ratio. The Bloom filter does not support spatial range queries. Table 4.4 also gives the space usage overhead for various local indexes in each worker. The sFilter is 5-6 orders of magnitude less than the other types of indexes, e.g., the R-tree and the Quadtree. This is due to the bit encoding of the sFilter that eliminates the need for pointers. Moreover, the sFilter reduces the unnecessary network communication. We study the shuffle cost for redistributing the queries. The results are given in Figure 4.10. The sFilter reduces the shuffling cost for both spatial range and kNN join operations. The shuffle cost reduction depends on the data and query distribution. Thus, the more unbalanced the distribution of queries and data in the various computation nodes, the more shuffle cost is reduced. For example, for kNN join, the shuffle cost is improved from 1114575 to 928933 when k is 30, achieving 18% reduction in network communication cost.

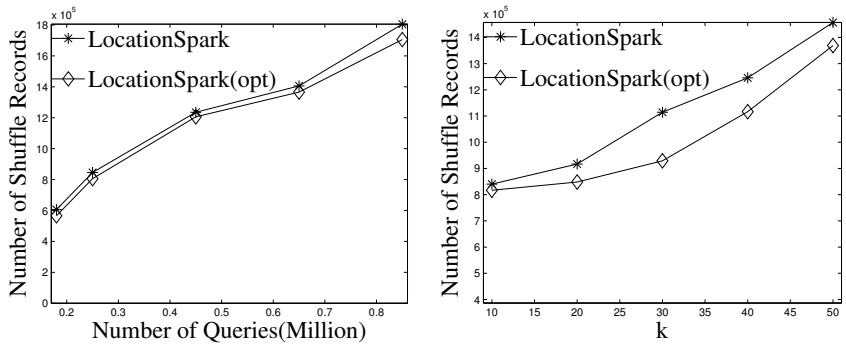
4.6.6 Effect of the Number of Workers

Spark's parallel computation ability depends on the number of executors and number of CPU cores assigned to each executor, that is, the number of executors times the number of CPU cores per executor. Therefore, to demonstrate the scalability of the proposed

[3] `https://github.com/scalanlp/breeze`

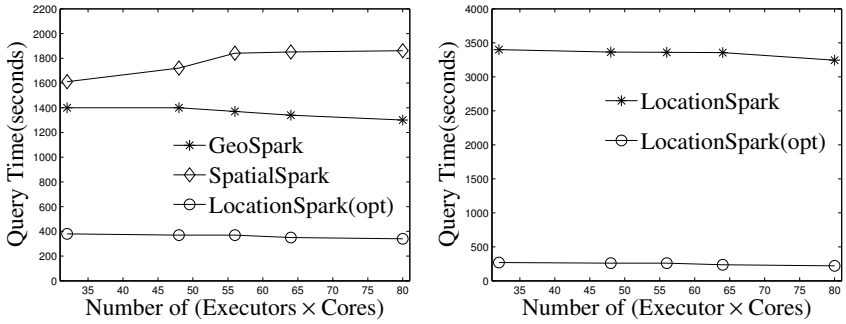
Table 4.4.: Performance of sFilter

Dataset	Index	Query time(ms)	Index build(s)	False positive	Memory usage(MB)
Twitter	R-tree	19	17	/	112
	Q-tree	0.4	1.8	/	37
	sFilter	0.022	2	0.07	0.006
	sFilter(ad)	0.018	2.3	0.09	0.003
	bloom filter	0.004	1.54	0.01	140
OSMP	R-tree	4	32	/	170
	Q-tree	0.5	1.2	/	63
	sFilter	0.008	2.4	0.06	0.008
	sFilter(ad)	0.006	6	0.10	0.006
	bloom filter	0.002	2.7	0.01	180



(a) Spatial-Range-Join (b) kNN join

Figure 4.10.: The effect of sFilter to reduce shuffle cost



(a) Spatial-Range-Join (b) kNN join

Figure 4.11.: Performance of Spatial-Range-Join and kNN-Join by various of number of executors

approach, we change the number of executors from 4 to 10, and fix the number of CPU cores assigned to each executor. We study the runtime performance for spatial range join and kNN join operations using the Twitter and OSMP datasets. Because the corresponding performance on the OSMP dataset gives similar trends as the Twitter dataset, we only present the performance for the Twitter dataset in Figure 4.11, where the outer table size is fixed to 1 million around Chicago area, and the inner table size is 15 million. We observe that the performance of LOCATIONSPARK for the spatial range join and the kNN join improves gradually with the increase in the number of executors. In contrast, GeoSpark and SpatialSpark do not scale well in comparison to LOCATIONSPARK for spatial range join. The performance of Magellan for spatial join is not shown because it is based on Cartesian product and shows the worst performance.

4.7 Related Work

Spatial data management has extensively been studied for decades and some surveys give good overviews. Gaede and Günther [82] provide a summary of data indexing for points. Sowell et al. give a survey and experimental study for iterative spatial-join in memory [74]. Recently, there has been considerable interest in supporting spatial data

management over Hadoop MapReduce. Afrati and Ullman [83] have proposed a framework that computes a multi-join query in a single computation round. Liu et al. [78] study how to use the Voronoi diagram data partition to speedup kNN join over MapReduce. Hadoop-GIS [66] supports spatial queries in Hadoop by using a uniform grid index. SpatialHadoop [14] builds global and local spatial indexes, and modifies the HDFS record reader to read data more efficiently. MD-Hbase [15] extends HBase to support spatial data update and queries. Hadoop MapReduce is good at data processing for high throughput and fault-tolerance. Yet, Hadoop MapReduce has to write intermediate data into HDFS, and hence impedes the performance of applications that require pipelines of multiple MapReduce jobs.

Taking advantage of the very large memory pools available in modern machines, Spark and Spark-related systems (e.g., Graphx, Spark-SQL, and DStream) [67, 84] are developed to overcome the drawbacks of MapReduce in specific application domains. In order to process big spatial data more efficiently, it is natural to develop an efficient spatial data management systems based on Spark. Several prototypes have been proposed to support spatial operations over Spark, e.g., GeoSpark [70], SpatialSpark [69], Magellan [71], Simba [72]. However, some important factors impede the performance of these systems, mainly, query skew, lack of adaptivity, and excessive and unoptimized network and I/O communication overheads. For exist Spatial-Range-Join [74, 75] and kNN join approaches [77–79], we conduct experimental study to analyze their performance in section 4.4.

Kwon *et al.* [73, 85] propose a skew handler to address the computation skew in a MapReduce platform. AQWA [86] is a disk-based approach that handles spatial computation skew in MapReduce. In LocationSpark, we overcome the spatial query skew for spatial range join and kNN join operators, and provide an optimized query execution plan. These operators are not addressed in AQWA. The query planner in LocationSpark is different from relational query planners, i.e., join order and selection estimation. ARF [81] supports one dimensional range query filter for data in disk. Calderoni *et al.* [87] study spatial Bloom filter for private data. Yet, it does not support spatial range querying.

4.8 Summary

In this chapter, we develop a system for efficient spatial computing. We presented a query executor and optimizer to improve the query execution plan generated for spatial queries. We conduct an extensive experimental study for local execution plan generation. We introduce a new spatial bloom filter to reduce the redundant network communication cost. Empirical studies on various real datasets demonstrate the superiority of our approaches compared with existing systems.

5 CONCLUSIONS

In this dissertation, we study similarity query processing in various data platforms. Therefore, one question arises, should we develop systems specifically for certain targeted applications, or build one system that fits all applications? This dissertation demonstrates that, in many cases, we cannot apply one platform for the broad spectrum of applications.

While application requirements are rapidly evolving, our experiments demonstrate that general systems without optimization usually perform worse. For example, systems without optimization (e.g., Similarity group by query based on RDBMS and similarity query based on Spark) show worse performance than optimized tailored systems. Thus, we believe that the developed techniques in this dissertation could be applied for various types of applications. For example, we propose and implement the similarity group by operators inside an RDBMS to query relational data (e.g., bank and stock transaction data). We propose a new index to speedup Hamming-distance search over high dimensional data (e.g., images and web pages) based on Map-Reduce. We develop an in-memory distributed system for similarity query processing over in-memory distributed spatial data. Overall, we believe these specific designs and optimization based on the type of data and the specific platforms can enable rapid innovation. Below, I present some future directions for research:

- **Spatial data analysis**: LOCATIONSPARK provides basic similarity query operators, e.g., spatial range select, and kNN select. However, spatial data analysis tasks, e.g., spatial data clustering, spatial skyline computation, and detection of abnormal spatial regions, are also crucially important. Therefore, we plan to support spatial data analysis based on LOCATIONSPARK. For example, spatial data clustering can be built upon the spatial join operator, the similarity group by operator can reuse the spatial range select operator. More importantly, we find that there are room to improve these

spatial data analysis tasks, because most of the similarity-aware operators are not specifically designed for data analysis.

- **Approximate similarity query processing**: Similarity query processing with approximate results is vital for the following reasons: (1) The run-time for computing exact query results is high, (2) In most cases, approximate query results are good enough for a specific application. Therefore, developing new approximate similarity query operators, e.g., approximate spatial range operators, approximate kNN operators, and approximate similarity group-by operators for in-memory spatial data is a promising future direction. Therefore, we plan to develop new semantics for approximate similarity queries, and study how approximation factors, e.g., confidence intervals, influence the run-time performance of a query. More importantly, an approximate query operator would change the query plan that is originally designed for exact query operators. Thus, query plan optimization for approximate similarity query operators is expected to attract more attention in the future.

- **Interaction with big spatial data**: Building a system to enable humans to interact with big spatial data is a promising direction. It is impractical to display large amounts of spatial data in one map. Furthermore, reading and transferring big spatial data to users have the drawback of expensive runtime and network communication cost. To interact with big spatial data efficiently, several new operators will be need to be developed, e.g., a query operator to read big spatial data from higher to lower resolution on the fly. Meanwhile, the system has to schedule more computing cycles towards spatial regions dynamically, as users may change their focus over big spatial data on the fly.

- **Spatio-textual and Spatio-image data**: Spatio-textual data represents an object by its spatial location and textual information. There is an emerging need for efficient data analytic techniques to make use of spatio-textual data. For example, with the increase in awareness about the environment and the sense of belonging, individuals may wish to find friends in their vicinity, and vice versa, queries can also satisfy the

needs of new friends that share similar interests (e.g., posting similar text). Mobile photo sharing services, e.g., Flicker, Instagram, WeChat, and WhatsApp, facilitate for its users to take pictures and videos while sharing them on a variety of social networking platforms. It has been reported that mobile image-sharing services (e.g., Instagram) are growing far more quickly than traditional social network services (e.g., Facebook). Similar to spatio-textual processing, a spatial-image object can also be converted into a high-dimensional vector (e.g., GIST) with a geo-location annotation. Similarity operations, e.g., similarity-join or kNN-join, over the transformed high-dimensional data is a core operation in many applications, e.g., in collaborative filtering and web search. Extending LOCATIONSPARK to support similarity operations over spatio-textual and spatial-image data is interesting and important. The reason is that traditional spatial indexing is not able to handle high-dimensional data.

- **Memory management for distributed in-memory spatial data**: Memory is a precious resource. The system has to allocate this precious resource to different jobs based on their priorities and users' access patterns. We plan to develop techniques to identify the frequently visited or less frequently visited spatial data with lower computation overhead. We plan to develop new memory placement policies to move data between memory and disk in order to achieve the best runtime performance for the various workloads.

We hope that continued experience with different applications will help us address these challenges, and lead to solutions that are applicable to many systems.

REFERENCES

REFERENCES

[1] MingJie Tang, Ruby Y. Tahboub, Walid G. Aref, Mikhail J. Atallah, Qutaibah M. Malluhi, Mourad Ouzzani, and Yasin N. Silva. Similarity group-by operators for multi-dimensional relational data. *IEEE Trans. Knowl. Data Eng.*, 2016.

[2] MingJie Tang, Yongyang Yu, Walid G. Aref, Qutaibah M. Malluhi, and Mourad Ouzzani. Efficient processing of Hamming-distance-based similarity-search queries over MapReduce. In *EDBT*, 2015.

[3] MingJie Tang, Yongyang Yu, Walid G. Aref, Qutaibah Malluhi, and Mourad Ouzzani. LocationSpark: A distributed in-memory data management system for big spatial data. In *VLDB*, 2016.

[4] MingJie Tang, Yongyang Yu, Walid G. Aref, Qutaibah M. Malluhi, Mourad Ouzzani, and Ahmed R.Mahmood. LocationSpark: A distributed in-memory data management system for big spatial data. *CoRR*, 2016.

[5] Shashi Shekhar, Steven K. Feiner, and Walid G. Aref. Spatial computing. *Commun. ACM*, 2016.

[6] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *WWW*, 2007.

[7] Andrew Kachites McCallum. MALLET: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[8] Yasin N Silva, Ahmed M Aly, Walid G Aref, and Per-Ake Larson. SimDB: A similarity-aware database system. In *SIGMOD*, 2010.

[9] Yasin N. Silva, Walid G. Aref, Per-Ake Larson, Spencer Pearson, and Mohamed H. Ali. Similarity queries: Their conceptual evaluation, transformations, and processing. *VLDB J.*, 2013.

[10] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: Concepts and techniques*. Morgan Kaufmann, 2006.

[11] Jingkuan Song, Yang Yang, Yi Yang, Zi Huang, and Heng Tao Shen. Inter-media hashing for large-scale retrieval from heterogeneous data sources. In *SIGMOD*, 2013.

[12] Yair Weiss, Antonio Torralba, and Robert Fergus. Spectral hashing. In *NIPS*, 2008.

[13] Michael M. Bronstein, Er M. Bronstein, Fabrice Michel, and Nikos Paragios. Data fusion through crossmodality metric learning using similaritysensitive hashing. In *CVPR*, 2010.

[14] A. Eldawy and M.F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. In *ICDE*, 2015.

[15] S. Nishimura, S. Das, D. Agrawal, and A.E. Abbadi. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In *MDM*, 2011.

[16] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 2001.

[17] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe LSH: Efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.

[18] Mohamed Y. Eltabakh, Mourad Ouzzani, and Walid G. Aref. BDBMS: A database management system for biological data. In *CIDR*, 2007.

[19] Mohamed Y. Eltabakh, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Yasin Laura-Silva, Muhammad U. Arshad, David E. Salt, and Ivan Baxter. Managing biological data using BDBMS. In *ICDE*, 2008.

[20] Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-Join: An efficient algorithm for similarity joins with edit distance constraints. In *VLDB*, 2008.

[21] Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. An efficient k-means clustering algorithm: Analysis and implementation. *PAMI*, 2002.

[22] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *SIGKDD*, 1996.

[23] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: An efficient data clustering method for very large databases. In *SIGMOD*, 1996.

[24] F. Gray. Pulse code communication. In *U.S. Patent 2,632,058*, 1953.

[25] MingJie Tang, Ruby Y. Tahboub, Walid G. Aref, Qutaibah M. Malluhi, and Mourad Ouzzani. On order-independent semantics of the similarity group-by relational database operator. *CoRR*, 2014.

[26] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[27] Mark De Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational geometry*. Springer, 2008.

[28] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 1984.

[29] Mikhail J. Atallah. Computing the convex hull of line intersections. *J. Algorithms*, 1986.

[30] TPC-H version 2.15.0.

[31] Eunjoon Cho, Seth A. Myers, and Jure Leskovec. Friendship and mobility: User movement in location-based social networks. In *SIGKDD*, 2011.

[32] Elke Achtert, Hans-Peter Kriegel, Erich Schubert, and Arthur Zimek. Interactive data mining with 3D-parallel-coordinate-trees. In *SIGMOD*, 2013.

[33] Sibel Adali, Piero Bonatti, Maria Luisa Sapino, and VS Subrahmanian. A multi-similarity algebra. In *SIGMOD*, 1998.

[34] Solomon Atnafu, Lionel Brunie, and Harald Kosch. Similarity-based operators and query optimization for multimedia database systems. In *Proceedings of the 12th Australasian Database Conference*, 2001.

[35] Bernhard Braunmuller, Martin Ester, H-P Kriegel, and Jörg Sander. Multiple similarity queries: A basic DBMS operation for mining in metric databases. *IEEE Trans. Knowl. Data Eng.*, 2001.

[36] MingJie Tang, Yuanchun Zhou, Peng Cui, Weihang Wang, Jinyan Li, Haiting Zhang, YuanSheng Hou, and Baoping Yan. Discovery of migration habitats and routes of wild bird species by clustering and association analysis. In *ADMA*, 2009.

[37] MingJie Tang, Yuanchun Zhou, Jinyan Li, Weihang Wang, Peng Cui, YuanSeng Hou, Ze Luo, Jianhui Li, Fuming Lei, and Baoping Yan. Exploring the wild birds' migration data for the disease spread study of H5N1: A clustering and association approach. *Knowl. Inf. Syst.*, 2011.

[38] Yuanchun Zhou, MingJie Tang, Weike Pan, Jinyan Li, Weihang Wang, Jing Shao, Liang Wu, Jianhui Li, Qiang Yang, and Baoping Yan. Bird flu outbreak prediction via satellite tracking. *IEEE Intelligent Systems*, 2014.

[39] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: An efficient clustering algorithm for large databases. In *SIGMOD*, 1998.

[40] Eike Schallehn, Kai-Uwe Sattler, and Gunter Saake. Efficient similarity-based operations for data integration. *Data & Knowledge Engineering*, 2004.

[41] Chengyang Zhang and Yan Huang. Cluster by: A new SQL extension for spatial data aggregation. In *GIS*, 2007.

[42] Maria Camila N Barioni, Humberto Razente, Agma Traina, and Caetano Traina Jr. SIREN: A similarity retrieval engine for complex data. In *VLDB*, 2006.

[43] Denise Guliato, Ernani V de Melo, Rangaraj M Rangayyan, and Robson C Soares. POSTGRESQL-IE: An image-handling extension for postgreSQL. *Journal of Digital Imaging*, 2009.

[44] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

[45] Minsky Marvin and A. Papert Seymour. Perceptrons. *MIT Press*, 1969.

[46] D. Greene, M. Parnas, and F. Yao. Multi-index hashing for information retrieval. In *FOCS*, 1994.

[47] AX. Liu, Ke Shen, and E. Torng. Large scale Hamming distance query processing. In *ICDE*, 2011.

[48] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD*, 2010.

[49] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using MapReduce. In *VLDB*, 2012.

[50] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel kNN joins for large data in MapReduce. In *EDBT*, 2012.

[51] H. Kllapi, B. Harb, and Cong Yu. Near neighbor join. In *ICDE*, 2014.

[52] Donald R. Morrison. Patricia: Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 1968.

[53] Christos Faloutsos. Multiattribute hashing using gray codes. In *SIGMOD*, 1986.

[54] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 2008.

[55] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 1985.

[56] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD*, 2010.

[57] Xiaoyang Zhang, Jianbin Qin, Wei Wang, Yifang Sun, and Jiaheng Lu. HmSearch: An efficient Hamming distance query processing algorithm. In *SSDBM*, 2013.

[58] Ramzi Nasr, Rares Vernica, Chen Li, and Pierre Baldi. Speeding up chemical searches using the inverted index: The convergence of chemoinformatics and text search methods. *Journal of Chemical Information and Modeling*, 2012.

[59] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the distance: An efficient method to KNN processing. In *VLDB*, 2001.

[60] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[61] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 2008.

[62] Aude Oliva and Antonio Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 2001.

[63] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Trans. Database Syst.*, 2010.

[64] Jinyang Gao, Hosagrahar Visvesvaraya Jagadish, Wei Lu, and Beng Chin Ooi. DSH: Data sensitive hashing for high-dimensional k-NN search. In *SIGMOD*, 2014.

[65] Wadha J. Al Marri, Qutaibah M. Malluhi, Mourad Ouzzani, MingJie Tang, and Walid G. Aref. The similarity-aware relational intersect database operator. In *SISAP*, 2014.

[66] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop GIS: A high performance spatial data warehousing system over MapReduce. In *VLDB*, 2013.

[67] Matei Zaharia. *An Architecture for Fast and General Data Processing on Large Clusters*. Association for Computing Machinery and Morgan, 2016.

[68] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *MSST*, 2010.

[69] SpatialSpark. `http://simin.me/projects/spatialspark/`.

[70] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. GeoSpark: A cluster computing framework for processing large-scale spatial data. In *ACM SIGSPATIAL*, 2015.

[71] Magellan. `https://github.com/harsha2010/magellan`.

[72] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*, 2016.

[73] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *SoCC*, 2010.

[74] Benjamin Sowell, Marcos Vaz Salles, Tuan Cao, Alan Demers, and Johannes Gehrke. An experimental analysis of iterated spatial joins in main memory. In *VLDB*, 2013.

[75] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-trees. *SIGMOD Rec.*, 1993.

[76] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2005.

[77] Chenyi Xia, Hongjun Lu, Beng Chin, and Ooi Jing Hu. GORDER: An efficient method for KNN join processing. In *VLDB*, 2004.

[78] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using MapReduce. In *VLDB*, 2012.

[79] Georgios Chatzimilioudis, Constantinos Costa, Demetrios Zeinalipour-Yazti, Wang-Chien Lee, and Evaggelia Pitoura. Distributed in-memory processing of all k nearest neighbor queries. *IEEE Trans. Knowl. Data Eng.*, 2016.

[80] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, 2013.

[81] Karolina Alexiou, Donald Kossmann, and Per-Ake Larson. Adaptive range filters for cold data: Avoiding trips to Siberia. In *VLDB*, 2013.

[82] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 1998.

[83] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. Technical report, National Technical University of Athens, Stanford University, December 2009.

[84] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[85] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. SkewTune: Mitigating skew in mapreduce applications. In *SIGMOD*, 2012.

[86] Ahmed M. Aly, Ahmed R. Mahmood, Mohamed S. Hassan, Walid G. Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. AQWA: Adaptive query-workload-aware partitioning of big spatial data. In *VLDB*, 2015.

[87] Luca Calderoni, Paolo Palmieri, and Dario Maio. Location privacy without mutual trust. *Comput. Commun.*, 2015.

VITA

VITA

Mingjie Tang has an M.S.(2010) in computer science from University of Chinese Academy of Sciences, and a B.S.(2007) in computer science from Sichuan University, China. His research interests include database system, data mining and machine learning.