UNIVERSITY OF PIRAEUS

DEPARTMENT OF DIGITAL SYSTEMS

Msc DIGITAL SYSTEMS AND SERVICES

MAJOR: BIG DATA AND ANALYTICS

Subject:

Parallel Processing of Spatio-textual Similarity Join Query

Pavlopoulos Nikolaos

ME 1727

Supervisor: C. Doulkeridis

Athens 2019

Abstract

With the rapid development of mobile Internet technology, Internet users are shifting from desktop to mobile devices. Modern mobile devices (e.g., smartphones and tablets) are equipped with GPS, which can help users to easily obtain their locations, and location-based services (LBS) have been widely deployed. Users that consume location-based services are generating more and more spatio-textual data, which contains both textual descriptions and geographical locations.

A spatio-textual similarity join is an important operation in spatio-textual data integration, which, given two sets of spatio-textual objects, finds all similar pairs from the two sets, where the similarity can be quantified by combining spatial proximity and textual relevancy. As a simpler example of spatio-textual query, a user that wants to find Points of Interest (POIs) (i.e., hotels, restaurants), gives a position, a radius of search and some keywords which describe the POI.

In contrast, the Spatio-Textual Similarity Join (STSJ) query returns all objects which are close enough based on the radius and have high textual relevance. The main problem for this operation is when the two datasets have large volumes and centralized computation is not feasible or even practicable. Therefore, the necessity of scalable processing of large volumes of datasets, motivates to use big data technologies in order to parallelize the computation and achieve scalability.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Christos Doulkeridis, for his support, advising, guidance. Also, I would like to thank the PhD student Panagiotis Nikitopoulos for his help and cooperation during the fulfillment of this thesis.

Last but not least, I would like to thank my family and friends for their support and encouragement.

Nikolaos Pavlopoulos
Athens, March 2019

Table of Contents

1. Introduction

Due to the existence of GPS and the ability of the applications to perform the position of the user with a text, such as Google Maps, Twitter, Foursquare and TripAdvisor, spatio-textual data are in the heart of research. Many researchers have discovered ways to analyze these kinds of data, to find useful conclusions, about points of interest, comments of users about places, and what is the common interest of the users. In addition, a recommendation system would be an important application for users, who desire to find places or information based on their common interests. Because of the popularity of the above applications, spatial and textual data are increasing explosively and centralized systems are not able to manage this volume of data. Thus, the necessity of creating new systems that could process data in parallel, was necessary. There are a few frameworks that can manage data in parallel with batch processing, such as Spark and Hadoop (Map-Reduce), and others with stream processing, which process data in real time, such as Spark Streaming and Storm. These systems are called Parallel / Distributed Systems and present better processing performance. However, the use of these systems requires computing resources because of the parallel processing, which is done in different machines or processors that are connected.

This thesis is divided in two main parts. First of all, the related work of managing spatio-textual data is described. Different type of queries that have been studied is presented and its problem statement are defined. The existing spatio-textual queries have been classified into different categories based on the query that they solve. The two main categories are the queries which have been implemented in Centralized and Parallel/Distributed systems. In this category, there are sub-categories depending on the query that they solve. The creating categories are captured in Figure 1 (Chapter 2).

Secondly, a type of query from the analyzed is selected and has been implemented in Spark. The query is the *Spatio-Textual Similarity Join (STSJ),* which is an important operation for reconciling different representations of an entity. Two objects are said to be similar if their spatial and textual similarity is greater than the given thresholds. There are various ways to quantify the spatial similarity and textual similarity. We have used the Euclidean distance as the spatial similarity and the Jaccard similarity for the textual part. The problem statement of this query is described below.

Given a collection of spatio-textual objects $R = \{r_1, r_2, \cdots, r_n\}$, where each object $r \in R$ includes a textual description r.text, and a spatial location r.loc that is represented by two-dimensional geographical coordinates a textual similarity threshold θ and a spatial distance threshold e, the STSJ(R, θ, e) aims to find all the similar pairs (r_i, r_j) where $sim_t(r_i, r_j) \geq \theta$ and $dist_l(r_i, r_j) \leq e$,

$STSJ(R, \theta, e) = \{(r_i, r_j) \mid r_i, r_j \in R, 1 \leq i, j \leq n, i \neq j, sim_t(r_i, r_j) \geq \theta, dist_l(r_i, r_j) \leq e\}$, where,

$sim_t(r_i, r_j)$ is the Jaccard similarity between two objects, and $dist_l(r_i, r_j)$ is the Euclidean distance of two objects.

This query has been studied in centralized systems (2.2.1.), where they split the space with a regular grid in order to make the computation faster. On the other hand, similarity join has been implemented in MapReduce (2.2.1), so as to manage huge volumes of data.

Due to the large volume of spatial and textual data, Spark is used to deal with this situation and process data in an efficient way. The main problem that must be faced is the load balancing of data, when they shared on different machines or processors. Therefore, different ways of partitioning data based on geographical coordinates are studied, so as to achieve the best performance of load balancing. Moreover, we have implemented two algorithms in order to compute the similarity join query and find which of them has the best performance.

Last but not least, the structure that follows the diploma thesis is listed below.

- Chapter 2: related work on spatio-textual queries.
- Chapter 3: related work on this thesis statement, Spatio-textual Similarity Join Query.
- Chapter 4: the distributed technology that has been used to manage this volume of data (Apache Spark).
- Chapter 5: the problem statement of the implemented query and the functions of similarity join which have been used.
- Chapter 6: analysis of the implementation for the similarity join query.
- Chapter 7: description of the platform, the evaluation metrics and the datasets that have used.
- Chapter 8: experiments for the evaluation of the algorithms.
- Chapter 9: conclusion of thesis.
- Chapter 10: references.

2. Related work on Spatio-textual Queries

In this chapter, the types of existing queries based on spatial and textual elements are described. There are different variations of spatio-textual queries and many researchers are dealing with these operations to improve them.

This thesis focuses on Parallel Processing Spatio-textual Similarity Join Query. Thus, Similarity Join Queries in Centralized systems and in Parallel/Distributed systems will be analyzed in the Section 3.

In the following, the taxonomy of spatio-textual queries is depicted to make an overview of the structure that this chapter has.

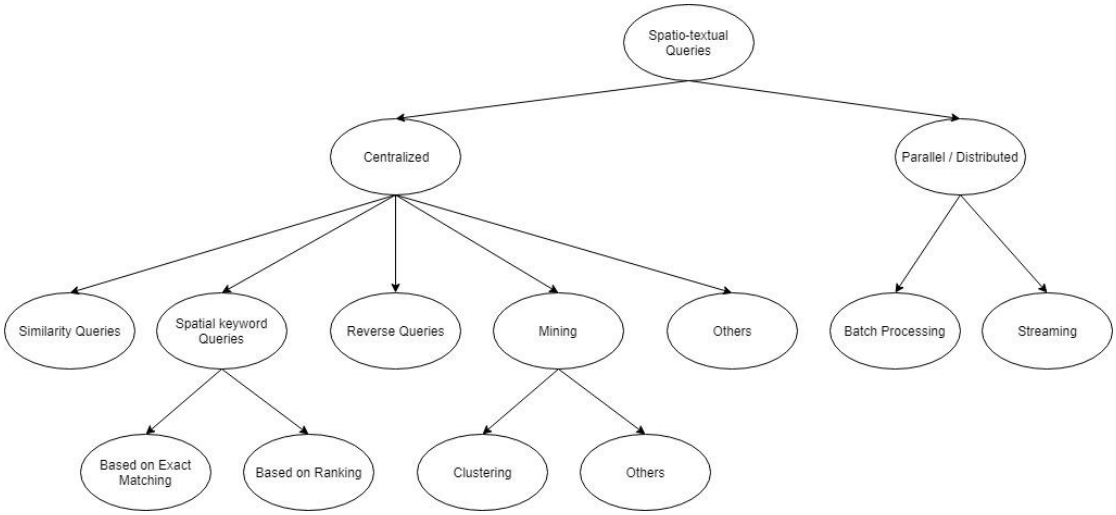

Figure 1: Taxonomy of Spatio-textual Queries.

2.1. Centralized

Centralized system is a system that runs in a single computer system and do not interact with other computer systems. All the aspects of the system are concentrated in a single entity and all calculations are done on one particular computer. There are a lot of variations of spatio-textual queries which use a centralized system and will be analyzed below.

2.1.1. Spatial Keyword Queries

Spatial keyword queries are being supported in real-life applications, such as Google Maps where points of interest can be retrieved, Foursquare where geo-tagged documents can be retrieved, and Twitter where tweets can be retrieved. Spatial keyword querying is also receiving increasing interest in the research community, where a range of techniques have been proposed for efficiently query processing. Many types of spatial database queries have been revisited for geo-textual data; and keyword queries have also been revisited in the context of geo-textual data. A typical spatial keyword query finds the objects that best match the location and keywords in the query. To address different use cases, many types of spatial keyword queries, as well as accompany indexing and query processing techniques have been proposed.

2.1.1.1. Based on Exact Matching

Spatial-keyword queries extend spatial queries by additionally taking into account the textual content. The first category of spatial-keyword queries concerns exact matching queries. Such queries seek objects that satisfy the spatial constraint (range or NN) and also contain *all* query keywords in their textual descriptions.

Representative types of spatial keyword queries using exact matching are reviewed in [11] and contain the Boolean range query and the Boolean kNN query. In the following, definitions are provided along with examples, in order to ease understanding and be more comprehensible.

Boolean Range Query (BRQ): Given a set of query keywords K, a location p, and a radius r, the Boolean range query retrieves the objects o: (a) whose textual description contains all query keywords, and (b) located within distance r from the query location, i.e., d(p,o)<=r. For example, a Boolean range query with K={tasty, pizza, cappuccino} and r=10km would retrieve all objects whose text description contains the keywords tasty, pizza, and cappuccino and whose location is within 10 km of the query location.

Boolean kNN Query (BkQ): Given a set of query keywords Q, a location p, and k the number of objects to retrieve, the Boolean kNN query retrieves a set of k objects, each of which covers all the keywords in Q. Objects are ranked according to their distances to p. For example, a Boolean kNN query with Q={tasty, pizza, cappuccino} and k=10, would retrieve the 10 objects whose text description contains the keywords tasty, pizza, and cappuccino and are the 10-NN to the location p.

These two types of queries have been evaluated with diverse spatial and textual indexing, such as R-Tree based indices (IF-R*, R*-IF, KR*-Tree, IR^2-Tree), grid based spatial-textual indices (ST, TS).

The most queries mainly focus on finding individual objects that each satisfy a query, rather than finding groups of objects, where the objects in a group together satisfy a query.

In [35], the problem of retrieving a *Group of Spatio-Textual Objects (GSTO)* such that the group's keywords cover the query's keywords and such that the objects are nearest to the query location and have the smallest inter-object distances, is defined.

To address the need for collective answers to spatial keyword queries, they assume a database of spatio-textual objects and then consider the problem of how to retrieve a group of spatio-textual objects that collectively meet the user's needs, given as a location and a set of keywords: (a) the textual description of the group of objects cover the query keywords; (b) the objects are close to the query point; and (c) the objects in the group are close to each other.

Specifically, given a set of spatio-textual objects D and a query $q = (\lambda, \psi)$ where λ is a location and ψ is a set of keywords, they consider three instantiations of the spatial group keyword query. It turns out that the subproblems corresponding to the three instantiations are all NP-hard.

- They aim to find a group of objects χ that cover the keywords in q such that the sum of their spatial distances to the query is minimized.
- They aim to find a group of objects χ that cover the keywords in q such that the sum of the maximum distance between an object in χ and q and the maximum distance between two objects in χ is minimized.
- They aim to find a group of objects χ that cover the keywords in q such that the sum of the minimum distance between an object in χ and q and the maximum distance between two objects in χ is minimized.

They use the IR-Tree as the index structure in the algorithms and they illustrate how can implement other indexes.

The same problem is dealt in [36], that retrieves a group of spatial objects which they meet the first two of the three instantiations, that referred above.

Table 1: Overview of Based on Exact Matching Queries.

Query	*Spatial Distance*	*Indexing*
BRQ	Euclidean	R-Tree, Grid
BkQ	Euclidean	R-Tree, Grid
GSTO	Euclidean	IR-Tree

2.1.1.2. Based on Ranking

Apart from exact matching retrieval, several papers consider ranked retrieval of spatio-textual objects, where a score quantifies the relevance of an object to the query. Typically, the user is interested in the subset of objects with highest scores, thus top-k queries are used, where the query result set consists of the k objects with highest scores.

Top-k kNN Query (TkQ) [11]**:** Given a query q=(Q,ρ,k), where Q is a set of query keywords, ρ is the query location, and k is the number of objects to retrieve, the top-k kNN query retrieves a set of k objects ranked according to a score that takes into consideration spatial proximity and text relevance. Specifically, the ranking score of object o is defined in the following equation:

$$ST(o, q) = \alpha \cdot SDist(o.\rho, q.\rho) + (1 - \alpha) \cdot TRel(o.Q, q.Q),$$

where SDist(o.ρ, q.ρ) is the spatial proximity between o.ρ and q.ρ, TRel(o.Q, q.Q) is the text relevance between o.Q and q.Q, and $\alpha \in [0, 1]$ is a query preference parameter that makes it possible to balance the spatial proximity and text relevance. The spatial proximity is defined as the normalized Euclidian distance: $SDist(o.\rho, q.\rho) = \frac{dist(o.\rho,q.\rho)}{distmax}$, where dist(o.ρ, q.ρ) is the Euclidian distance between o and q, and distmax is the maximum distance between any two objects in D. The text relevance TRel(o.Q, q.Q) can be computed using an information retrieval model, here is used the language model. The same problem is dealt by [24] with the same ranking function as previously.

Moreover, [2] presents the problem of *Batch Processed Top-k Spatial-Textual Queries (BPkQ)*, where the main difference with [11], is that this type of processing, answers a set of queries in a single pass. It uses the same ranking functions for the spatial proximity and text relevance, as mentioned above. They assume objects are stored on disk and indexed them using a spatial-textual index, the IR-tree.

Different ways have been proposed in the literature on how to define the score function. Except for the language model that is referred above, there are also various ranking functions. The ranking functions are based on the text relevance between the query and each object. Other information retrieval models are, cosine similarity, BM25 and Jaccard similarity.

In [16], another query type is proposed, called *Top-k Spatio-Textual Preference Query (STPQ)*, for ranked retrieval of data objects based on the textual relevance and the non-spatial score of feature objects in their neighborhood. The score of a data object changes depending on the query keywords, which renders techniques that rely on materialization. In the following, the problem statement that solves this query type, is described.

Given an object dataset O and a set of c feature datasets $\{F_i \mid i \in [1, c]\}$, it is addressed the problem of finding k data objects that have in their spatial proximity highly ranked feature objects that are relevant to the given query keywords. Each data object $p \in O$ has a spatial location. Similarly, each feature object $t \in F_i$ is associated with a spatial location, but also with a non-spatial score t.s that indicates the goodness (quality) of t and its domain of values is the range [0, 1]. Moreover, t is described by set of keywords t.W that capture the textual description of the feature object t. The goal is to find data objects that have in their vicinity feature objects that are of high quality and are relevant to the query keywords posed by the user. Thus, the score of the feature object t captures not only the non-spatial score of the feature, but also its textual similarity to a user specified set of query keywords. Therefore, the problem definition is referred to a given query Q, defined by an integer k, a radius r and c-sets of keywords W_i, find the k data objects $p \in O$ with the highest spatio-textual score τ(p). Below, analyzed the spatio-textual score functions.

The preference score s(t) of feature object t based on a user-specified set of keywords W is defined as: $s(t) = (1 - \lambda) \cdot t.s + \lambda \cdot sim(t, W)$, where $\lambda \in [0, 1]$ and sim() is a textual similarity function. The textual similarity is equal to the Jaccard similarity between the keywords of the feature objects and the user-specified keywords: $sim(t, W) = \frac{|t.W \cap W|}{|t.W \cup W|}$.

The preference score $\tau_i(p)$ of data object p based on the feature set F_i is defined as: $\tau_i(p) = \max\{s(t) \mid t \in F_i : dist(p, t) \leq r \text{ and } sim(t, W_i) > 0\}$. The dist(p, t) denotes the spatial distance between data object p and feature object t and is computed by the Euclidean distance function.

The overall spatio-textual preference score τ(p) of data object p is defined as: $\tau(p) = \sum_{i \in [1,c]} \tau_i(p)$.

It is assumed that the data objects O are indexed by an R-tree, and for the feature objects, it is important that the non-spatial score and the textual description are indexed additionally. Their indexing approach maps the textual description of feature objects to a value based on the Hilbert curve.

Another query based on a ranking, is called *Finding Top-k Relevant Groups of Spatial Web Objects (RGSWO).* [34] proposes a type of query functionality that returns top-k groups of objects, while taking into account aspects such as group density, distance to the query, and relevance to the query keywords. Given a set of spatial web objects with spatial coordinates and text descriptions, the top-k groups spatial keyword query takes a location and a set of query keywords as arguments and returns k groups of objects such that the objects in a group are close to each other, the group is close to the query location, and the objects in the group are textually relevant to the query keywords. Below, is referred the problem statement and are defined the distance and the textual relevance functions.

A top-k groups query q takes a triple of arguments (λ, ϕ, k), where q.λ is a point location, q.ϕ is a set of keywords, and q.k is the number of groups in the result. It returns k subsets G_1 , ..., G_k of D such that $G_i \subseteq D_i$, i = 1,..., k, where $D_1 = D$ and $D_i = \frac{D}{\cup_{j=1}^{i-1} G_j}$, i = 2, ..., k, and such that there does not exist a subset $G \subseteq D_i$ for which $Cost(q, G) < Cost(q, G_i)$, i = 1, ..., k.

The distance between a query location λ and a group G of objects, is defined as the distance between the query location and the nearest object in the group: $d(\lambda, G) = \min_{o \in G} ||\lambda, o.\lambda||$, where $||\cdot, \cdot||$ denotes the Euclidian distance.

The text relevance is evaluated by the language models [25]. A text document is represented by a vector where each dimension corresponds to a distinct term in the document. The relevance of an object o to a query term, $t \in q.\phi$, is defined as follows:

$TR(t, o.\phi) = (1 - \gamma) \cdot \frac{tf(t,\ o.\varphi)}{|o.\varphi|} + \gamma \cdot \frac{tf(t,\ Coll)}{|Coll|}$, where tf(t, o.ϕ) is the number of occurrences of term t in o.ϕ, tf(t, Coll) is the count of term t in the collection Coll of D, and γ is a smoothing parameter.

The indexing technique that used is the Group Extended R-Tree (GER-trees).

Table 2: Overview of Based on Ranking Queries.

Query	*Spatial Distance*	*Textual Similarity*	*Indexing*
TkQ	Normalized Euclidean	Language Model	-
BPkQ	Normalized Euclidean	Language Model	IR-Tree
STPQ	Euclidean	Jaccard	R-Tree, Hilbert Curve
RGSWO	Euclidean	Language Model	GER-Trees

2.1.2. Reverse Queries

In many application scenarios, users cannot precisely formulate their keywords and instead prefer to choose them from some candidate keyword sets. Moreover, in information browsing applications, it is useful to highlight the objects with the tags (keywords) under which the objects have high rankings. Driven by these applications, a query paradigm, namely *Reverse Keyword Search for Spatio-Textual Top-k Queries (RSTQ)* [14], is proposed. It takes a user location and a target object as inputs, and returns the keyword sets, derived from the textual description of the target object, under which the target object will be a spatio-textual top-k query result. Formally, the *RSTQ* is defined as follows.

Point-based RSTQ Query *(PRSTQ):* Given a target object o, a query point q.λ, an argument k, as well as a list of keyword sets Ω, the RSTQ query finds a subset Ω' of Ω, satisfying that:

i) for any q.ψ ϵ (Ω – Ω'), rank(q, o) ≤ k,
ii) for any q.ψ ϵ (Ω – Ω'), rank(q, o) > k.

Region-based RSTQ Query *(RRSTQ)*: Given a target object o, a query region Λ, an argument k, as well as a list of keyword sets Ω, the RSTQ query finds a subset Ω' of Ω, satisfying that:

i) for any q.ψ ϵ Ω', ∀ q.λ ϵ Λ, rank(q, o) ≤ k,
ii) for any q.ψ ϵ (Ω - Ω'), there is at least one point λ ϵ Λ such that if q.λ = λ, rank(q, o) > k.

To efficiently process point-based RSTQ queries, they propose a hybrid indexing structure, called KcR-tree, and an efficient query processing algorithm.

Moreover, an interesting problem encountered in real-life applications that rely on spatio-textual retrieval is how to improve the ranking of a spatio-textual object for as many users as possible. In [5], to address this problem, they capitalize on *reverse top-k queries*, which retrieve the set of users that have a given object in their top-k results. They model the problem as a maximization of the cardinality of the reverse top-k result set, and they explore the different combinations of keywords that will increase the query object's rank for many users,

when added to its textual annotation. They refer to this problem as *Best-Terms Query (BTQ)*. Below, the problem statement is described.

Given a set D of spatio-textual objects, a set of terms A = $\bigcup_{o \in D} o.T$, a set of queries U, a scoring function f, an integer k, and a spatio-textual object q = <q.T, q.L> ∈ D, decide if there is a set BT such that BT ⊆ A, BT ⋂ q.T = ∅, |BT| ≤ b for which it holds that I(q_1) = U where q_1 = <q.T ⋃ BT, q.L>.

They propose a greedy algorithm, termed *Best Term First (BTF),* that provides an approximate solution to the Best-terms problem. BTF takes as input an IR-tree index containing the set of spatio-textual objects D, and an IR-tree index containing the set of user preferences U. BTF extends the textual description of a spatio-textual object iteratively, which forces the algorithm to scan the preferences set U multiple times. Thus, they present an algorithm, named *Graph-Based Term Selection (GBTS),* which examines the set of preferences only once and creates a graph of terms that provides an estimation of the influence gain any combination of terms may provide.

Another approach on reverse queries is to find objects that take the query object as one of their k most spatial-textual similar objects. This type of query is defined as *Reverse Spatial Textual k Nearest Neighbor* (*RSTkNN*) query [15].

The document of an object is treated as a bag of weighted words using vector space model. Formally, a document is defined as {<d_i, w_i>}, i = 1,⋯, m, where w_i is the weight of word d_i. The weight could be computed by the well-known TF-IDF scheme. Let P be a universal spatial object set. Each spatial object p ∈ P is defined as a pair (p.loc, p.vct), where p.loc represents the spatial location information and p.vct is the associated text represented in vector space model. The *RSTkNN* query is defined as follows.

Given a set of objects P and a query point q(loc, vct), RSTkNN(q, k, P) finds all objects in the database that have the query point q as one of the k most “similar” neighbors among all points in P, where the similarity metric combines the spatial distance and textual similarity.

The similarity metric, called spatial-textual similarity, is computed by the following function:

SimST(p_1, p_2) = α ∗ SimS(p_1.loc, p_2.loc) + (1 − α) ∗ SimT(p_1.vct, p_2.vct), where,

- parameter α ∈ [0, 1] is used to adjust the importance of spatial proximity factor and the textual similarity factor.
- SimS(p_1.loc, p_2.loc) = 1 − $\frac{dist\ (p_1.loc, p_2.loc) - \varphi_\varsigma}{\psi_\varsigma - \varphi_\varsigma}$, where the distance is computed by the Euclidean distance, and ϕ_ς, ψ_ς denotes the minimum and maximum distance of pairs of distinct objects in P.
- SimT(p_1.vct, p_2.vct) = $\frac{EJ(p_1.vct, p_2.vct) - \varphi_t}{\psi_t - \varphi_t}$, where the textual similarity is computed by the Jaccard similarity, φ_t and ψ_t are the minimum and maximum textual similarity of pairs of distinct objects in the dataset, respectively

To answer RSTkNN queries efficiently, it is proposed a hybrid index tree called IUR-tree (Intersection-Union R-Tree) that effectively combines location proximity with textual similarity.

Another problem that scientists occupy, is the *Maximized Bichromatic Reverse Spatial Textual k Nearest Neighbor (MaxBRSTkNN)* query [3]. The problem is to find the location and a set of keywords, that maximizes the size of bichromatic reverse spatial textual k nearest neighbors. The most relevant query type to this problem is *RSTkNN* query, where an upper and a lower bound estimation of similarity is computed between each node of the IUR-tree and the k-th most similar object. A branch-and-bound algorithm is then used to answer the *RSTkNN* query. In *MaxBRSTkNN* the computation of the bounds and the algorithm are designed for the monochromatic case only since both the data objects and the query objects belong to the same type, and the nodes of the tree store only one type of object.

Let D be a bichromatic dataset, where U is a set of users and O is a set of objects. Each object o ∈ D is a pair (o.l, o.d), where o.l is the spatial location (point, rectangle, etc.) and o.d is a set of keywords. Each user u ∈ U is also defined as a similar pair (u.l, u.d). A *MaxBRSTkNN* query returns a location l ∈ L and a set W' ⊆ W, |W'| ≤ w_s such that, if o_x.l = l and o_x.d = W'∪o_x.d, the size of BRSTkNN of ox from U is maximized.

An object o is ranked based on a combined score of spatial proximity and textual relevance with respect to a user u, specifically, using the following equation:

STS(o, u) = α · SS(o.l, u.l) + (1−α)·TS(o.d, u.d),

where SS(o.l, u.l) is the spatial proximity between locations, the textual relevance is TS(o.d,u.d), and the preference parameter α ∈ [0,1] defines the importance of one relevance measure relative to the other. The value of both measures are normalized within [0,1].

Spatial proximity: The spatial proximity of an object o with respect to a user u is:

SS(o.l, u.l) = $1 - \frac{\mathrm{dist(o.l,u.l)}}{d_{max}}$, where dist(o.l, u.l) is the minimum Euclidean distance, and d_{max} is the maximum Euclidean distance between any two points in D.

Text relevance: The TF-IDF metric is used, in order to weight a term in a document based on term frequency (TF) and inverse document frequency (IDF). The TF, tf(t, d), is the number of times term t appears in document d, and IDF, idf(t, O) = $\log\frac{|O|}{|d \in O,\ tf(t,d)>0|}$ measures the importance of t in the set O. Here, the text relevance of an object o with respect to a user u is: TS(o.d, u.d) = $\sum_{t \in u.d} \mathrm{tf(t, o.d)} \times \mathrm{idf(t, O)}$.

To answer a *MaxBRSTkNN* query in the baseline approach, the score of the k-th ranked object for each user must be determined. Computing the top-k objects for each user requires retrieving the objects from disk, and the same objects might be retrieved multiple times for different users. To overcome this drawback, they have extended the IR-tree, as MIR-tree, and they compute the top-k objects of the users jointly using different pruning strategies, and ensure that an object is retrieved only once.

In [38], the *Reverse Top-k Keyword-Based Location Query* (RTkKL), is defined. This type of query takes a set of keywords, a query object q, and a number k as arguments, and it returns

a spatial region such that any top-k spatial keyword query with the query keywords and a location in this region would contain object q in its result.

Specifically, given a set ψ of query keywords, a query object $q \in O$, and a number k, the *RTkKL* returns the maximum spatial region V_q such that q is contained in the result of any top-k spatial keyword query with ψ as the keywords and any location in V_q as arguments.

Formally, $V_q = \{p \in \Omega \; q \in S_k(p, \psi) \wedge q \in O\}$, where $S_k(p, \psi)$ is the result of a top-k spatial keyword search with p and ψ as arguments.

Top-k spatial keyword query: Given a point $p \in \Omega$, a set ψ of keywords, and a number k, the top-k spatial keyword query $S_k(p, \psi)$ returns a set S_k of k spatial web objects with lowest ranking scores:

$\forall o \in S_k, \forall o' \in O - Sk, score(p, \psi, o) < score(p, \psi, o')$

Function score(), combines spatial proximity and textual relevance. When ψ is clear from the context, $|p, o|$ represents the $score(p, \psi, o)$.

$score(p, \psi, o) = w_s \cdot |p, o|_E + w_t \cdot (1 - tr(\psi, o.\psi))$, where $|p, o|_E$ denotes the Euclidean distance between p and o.loc, and function tr(), computes the textual relevance between its two arguments, by the cosine similarity function. The smaller the score of an object is, the more relevant the object is to the query.

They propose an algorithm capable of computing approximate V-regions with quality guarantees, based on Voronoi concepts, which are usually used for defining such kinds of spatial regions. Given a set S of spatial point objects, the Voronoi cell for object $q' \in S$ is the part of the space that contains all points in the underlying space that have q' as their nearest neighbor. Object q' is called the cell's site. This concept can be extended to k-nearest neighbors. If the site of a Voronoi cell is a set s of objects ($s \subseteq S$ and $|s| = k$), the cell is called order-k Voronoi cell. Every point of the underlying space in the cell takes s as their k nearest neighbors. To further accelerate V-region computation, they use a quad-tree for indexing the solution space and an IR-tree for indexing the objects, and they show how to use the two indexes in combination to enable pruning.

Table 3: Overview of Reverse Queries.

Query	***Spatial Distance***	***Textual Similarity***	***Indexing***
PRSTQ, RRSTQ	-	-	KcR-Tree
BTQ	Normalized Euclidean	Intersection of terms	IR-Tree
RSTkNN	Euclidean	Jaccard, TF-IDF	IUR-Tree
MaxBRSTkNN	Normalized Euclidean	TF-IDF	MIR-Tree
RTkKL	Euclidean	Cosine	Quad-Tree, IR-Tree

2.1.3. Mining

Mining spatio-textual data for knowledge discovery is a cumbersome task due to the complexity of the data type and its representation. Spatial-Textual data can be mined or analyzed to improve various location-based services. There are a lot of algorithms solving mining problems, such as the problem of Clustering, Classification and Regression.

2.1.3.1. Clustering

Clustering itself is a central problem in computer science, so spatio-textual clustering can play a key role in many applications.

The problem of *Clustering Spatio-Textual (CST)* data is studied in [26]. In particular, they focus on extending the k-means algorithm for a massive volume of spatio-textual dataset to be efficiently processed. K-means still remains one of the most popular data processing algorithm over half a century especially due to its simplicity and scalability [25]. Since the key process of k-means is to compute and update the mean value of each cluster, most k-means family algorithms assume that each data object only contains numeric attributes. This assumption makes a big challenge in applying the k-means algorithm to spatio-textual data as each object contains both numeric (spatial) and non-numeric (textual) attributes. To address the challenge above, they first observe that it suffices to compute the expected distance between a random object in each cluster and the object under consideration rather than measuring the distance from the virtually constructed spatio-textual centroid of a cluster. By doing so, they can reduce the cost of computing pairwise textual distances. Furthermore, they devise an effective technique for initializing k-means for spatio-textual data, which is commonly the most important and challenging task for k-means derivatives to improve not only the quality of resulting clusters but also the efficiency.

Their problem environment follows the many works in the literature of spatio-textual similarity search, which is summarized as follows:

- We consider a set of spatio-textual objects, denoted by $O = \{o_1, o_2, \ldots, o_{|O|}\}$.
- Each object $o \in O$ consists of two attributes, namely $<loc, \tau>$, where loc is a geographic location and $\tau = \{t_1, t_2, \ldots, t_{|\tau|}\}$ is a set of keywords.
- Each keyword $t \in o.\tau$ is associated with a weight w(t), which represents the significance of the keyword and is global for all objects. The most widely used value for the keyword weight is the inverted document frequency (idf).
- The distance between two spatio-textual objects o_1 and o_2 is defined as:
  $Dist(o_1, o_2) = \alpha \cdot DistS(o_1, o_2) + (1 - \alpha) \cdot DistT(o_1, o_2)$,
  where α is a user parameter to adjust the importance of spatial dimension or textual dimension, DistS(*, *) is the Euclidean distance between $o_1.loc$ and $o_2.loc$, and DistS(*, *) is the weighted Jaccard distance between $o_1.\tau$ and $o_2.\tau$ defined as follows:
  $1 - \frac{\sum_{t \in o1.\tau \cap o2.\tau} w(t)}{\sum_{t \in o1.\tau \cup o2.\tau} w(t)}$.

Note that α is not a parameter to be optimized in advance, but rather it represents a user's intention on whether s/he is interested in the spatial aspect or the textual aspect of the underlying dataset.

Then, their problem is formally defined as follows:

Given O, DistS(*, *), and a positive integer k, partition O into k disjoint clusters such that the total intra cluster distance is minimized and the total inter cluster distance is maximized with respect to DistS(*, *).

Furthermore, another approach of spatio-textual clustering is the *Top-k Spatial Textual Clusters* (k-STC) query [37], that returns the top-k clusters that are located close to a given query location, contain relevant objects with regard to given query keywords, and have an object density that exceeds a given threshold. This query aims to support users who wish to explore nearby regions with many relevant objects. It is used density-based clustering for finding clusters. The two basic steps to compute the top-k STC query are to obtain the objects that are relevant to the query keywords and to apply clustering to these objects. They consider a cluster scoring function that favors clusters close to the query location and that contain objects with high relevance with regard to the query keywords.

A top-k Spatial Textual Cluster (k-STC) query q = (λ, ψ, k, e, minpts) takes five arguments: a point location λ, a set of keywords ψ, a number of requested object sets k, a distance constraint e on neighborhoods, and the minimum number of objects minpts in a dense e-neighborhood. It returns a list of k spatial textual clusters that minimize a scoring function and that are in ascending order of their scores. The maximality of each cluster implies that the top-k clusters do not overlap. The density requirement parameters e and minpts are able to capture how far the user is willing to move before reaching another object.

Intuitively, a cluster with high text relevance and that is located close to the query location should be given a high ranking in the result. Thus, they use the following scoring function:

$$score_q(R) = \alpha \cdot d_{q.\lambda}(R) + (1 - \alpha) \cdot (1 - tr_{q.\psi}(R)),$$

where $d_{q.\lambda}(R)$ is the minimum spatial distance between the query location and the objects in R and $tr_{q.\psi}(R)$ is the maximum text relevance in R. The approaches we present are applicable to scoring functions that are monotone with respect to both spatial distance and text relevance. Parameter α is used to balance the spatial proximity and the text relevance of the retrieved clusters. All spatial distances and text relevances are normalized to [0, 1].

Table 4: Overview of Clustering Queries.

Query	***Clustering***
CTS	K-Means
k-STC	DBScan

2.1.3.2. Others

Another pattern mining problem is proposed in [1], called Spatial-Textual Sequence Pattern Mining. A Spatial-Textual sequence is a trajectory of locations with each location having associated with it a set of activities/events or some other attributes. Mining Spatial-Textual frequent sub-sequential patterns is one of the major challenge due to the complexity of the data type, as we have to deal with not only two different dimensions, but also ordered data and localization error of GPS.

Let $I = (l_1, i_1),(l_2, i_2), \ldots ,(l_m, i_n)$ be a set of all items along with their locations. An itemset is a non-empty subset of I and a sequence is an ordered list of itemsets. A sequence s is denoted by $<s_1, s_2, \ldots , s_l>$, where s_j is an itemset and s_j is also called an element of the sequence, and denoted as $(x_1, x_2, \ldots , x_m)$, where x_k is an item-pair (with location). The number of instances of item-location pairs in a sequence is called the length of the sequence. A sequence with length l is called an l−sequence. A sequence $\alpha = <a_1, a_2, \ldots, a_n>$ is called a subsequence of another sequence $\beta = <b_1, b_2, \ldots, b_m>$ and β a supersequence of α, denoted as $\beta \subseteq \alpha$, if there exist integers $1 \leq j_1 < j_2 < \ldots < j_n \leq m$ such that $a_1 \subseteq b_{j1}$, $a_2 \subseteq b_{j2}$, . . . , $a_n \subseteq b_{jn}$.

A sequence database S is a set of tuples <sid, s>, where sid is a sequence_id and s a sequence. A tuple <sid, s> is said to contain a sequence α, if α is a subsequence of s. The support of a sequence α in a sequence database S is the number of tuples in the database containing α, i.e., $support_s(\alpha) = |<sid, s> | (<sid, s> \in S) \wedge (\alpha \subseteq s)|$.

Given a positive integer min_support as the support threshold, a sequence α is called a sequential pattern in sequence database S if $support_s(\alpha) \geq$ min_support. A sequential pattern with length l is called an l–pattern.

The problem here is to find frequent Spatial-Textual patterns from given Spatial-Textual data. The locations can be of any type like the actual location or it can be the zip-code of the area or it can be the city name/state name/country name. For assigning label to the locations DBScan algorithm is used which work for given ρ value and minimum points. It gives the clusters of locations and assign labels to them. They use these labels in the dataset for locations. Also, Grid-based algorithm is used for labeling the locations. They use PrefixSpan algorithm to solve this type of problem, in which item-location pair is taken as a prefix instead of a single item.

2.1.4. Others

Apart from the categories that referred above, there are a lot of query variations that do not match to any of these categories. These types of queries will be analyzed below, without having any similarity to each other.

In [20], the problem of matching point sets based on the spatio-textual objects they contain, is addressed. This is highly relevant for users associated with geolocated photos and tweets. It is formally defined this problem as a *Spatio-Textual Point-Set Join (STPSJoin)* query,

and it is introduced its top-k variant. Given sets of spatio-textual objects, each one belonging to a specific entity, this query seeks pairs of entities that have similar spatio-textual objects. The problem statement is defined as follows.

Given a database D of spatio-textual objects s created by different users U, where each object o ∈ D is a triple o = <u, loc, doc>, where u ∈ U is the user associated with this object, loc = <x, y> is a spatial point and doc is a set of keywords belonging to a set of users U, the *STPSJoin* query is a tuple Q = <q.loc, q.doc, q.u> which returns a set R containing all pairs of users (u, u') such that u, u' ∈ U, u ≺ u', and $\sigma(D_u, D_{u'}) \geq$ q.u with respect to the spatial and textual thresholds q.loc and q.doc. The spatial distance between two objects is calculated as the Euclidean distance between their spatial locations, and their textual similarity as the Jaccard similarity.

An extension of the *STPSJoin* query in which it is seeked only the k best pairs of users, in terms of spatial and textual similarity of their objects, is the *Top-k STPSJoin (kSTPSJoin)* query. Formally, the *kSTPSJoin* query is defined as follows.

Given a database D of spatio-textual objects belonging to a set of users U, the *Top-k STPSJoin* query is a tuple Q = <q.loc, q.doc, k> which returns a set R containing k pairs of users (u, u') such that u, u' ∈ U, u ≺ u', and for any pair of users (v, v') ∉ R it holds that $\sigma(D_u, D_{u'}) \geq \sigma(D_v, D_{v'})$ for each (u, u') ∈ R with respect to the spatial and textual thresholds q.loc and q.doc.

They present a baseline algorithm using grid partition for the evaluation of the *STPSJoin* query, and they introduce methods that exploit a filter and refine strategy in combination with spatio-textual indexes in order to direct the search. In addition, they explain how their methods can be adapted to account for the *kSTPSJoin* query.

Another interesting query type is named *Clue-Based Spatio-Textual Query (CSTQ)* [23]. In many scenarios, a user cannot provide enough information to pinpoint the POI (Point of Interest) except some clue. Motivated by this observation, this work allows user providing clue, i.e., some nearby POIs and the spatial relationships between them, in POI retrieval. The objective is to retrieve k POIs from a POI database with the highest spatio-textual context similarities against the clue.

The POI database is denoted as D. Each POI o ∈ D is represented as (o.id, o.loc, o.cid), where o.id is the identity of o, o.loc refers to the location of o, and o.cid is the category identity to indicate o. When querying a POI in a POI database, *Clue* is the user-provided information which specifies the spatio-textual context of the querying POI. It includes the categories of nearby POIs around the querying POI, called clue POIs, and the spatial relationships (i.e., distances and relative directions) between them (including the querying POI and clue POIs). The definition of the *CSTQ* is referred below.

Given a POI database D, a *CSTQ* query $Q_R(q, N, E)$ retrieves k POIs, $A \subseteq D_R(q.cid)$, such that $SCsim(q, o_i) > SCsim(q, o_j)$, $o_i \in A$, $o_j \in D_R(q.cid) \setminus A$, where,

q is the querying POI, $N \setminus \{q\}$ is the set of clue POIs, E is the set of edges which represent the relative spatial relationships between the POIs in N, R is the region where the POIs in N are located, $D_R(q.cid)$ be the POIs in D with the same category as q in region R, and the spatio-textual context similarity between q and o is denoted as follows:

SCsim(q, o) = $max_{q^m \in T1,\ o^m \in T2} SCsim(q, o, q^m, o^m)$, where, T1 = N \ {q} and T2 = $D_R(q^m.cid)$ \ {o}, SCsim(q, o, q^m, o^m) := $\gamma \cdot max_{t \subseteq \Phi} SCsim(N, I)$, and Φ is the set of all possible matching instances of N in D_R.

Also, this work has developed an index called roll-out-star R-tree (RSR-tree) to improve the query processing efficiency.

In some cases, it is difficult for users to identify the exact keywords that describe their query intent. After a user issues an initial query and gets back the result, the user may find that some expected objects are missing and may wonder why. Specifically, objects that the user expected to be in the result are missing. This suggests to the user that other useful objects, that are as yet unknown to the user, may be missing from the result as well, and the user has reason to question the overall utility of the query and its result. In this setting, the utility of spatial keyword querying can be improved by offering functionality that explains to the user why one or more expected objects are missing and how to minimally modify the initial query so that the missing objects, and then potentially also other useful objects, become part of the result.

Such scenarios call for support for why-not questions, which were first introduced by Chapman and Jagadish [40]. In [39], is applied the query refinement model [41] to solve the problem of answering *why-not questions on spatial keyword top-k queries* via keyword adaption.

After a user issues a query q = (loc, doc_0, k_0, α) and receives the result, the user may observe that one or more objects that were expected to be in the result are missing. The user may then pose a why-not question with a set of missing objects M = {m_1, m_2, ..., m_j}, asking the system for a refined query q ′ = (loc, doc′, k′, α) the result of which contains the missing objects. Since it is possible that no modified set of keywords can revive the missing objects, we also consider the enlargement of k. It is adopted a penalty model [41], [42], that associates a penalty with a refined query. It is defined as the weighted sum of the modifications of the two parameters, i.e., Δk and Δdoc. The penalty (cost) of a query q ′ that refines an original query q is defined as follows:

Penalty(q, q′) = $\lambda \cdot \frac{\Delta k}{R(M,q) - k_0} + (1-\lambda) \cdot \frac{\Delta doc}{|doc_0 \cup M.doc|}$,

where λ is a user preference on the modification of q.k versus q.doc and R(M, q) = $max_{mi \in M}$ R(mi , q). Next, Δk = max (0, k′ − k_0) since for a refined query q ′, if R(M, q′) > k_0, k′ must be no smaller than R(M, q′) to revive the missing objects; otherwise, k′ can remain at k_0. They normalize Δk by R(M, q) − k_0, as a basic refined query is to keep the original query keywords and enlarge k_0 to R(M, q); for other refined queries that modify the query keywords to achieve a lower penalty than that of this basic one, Δk must not exceed R(M, q) − k_0. Using the principle of edit distance, the modification of query keywords Δdoc is quantified as the minimum count, denoted as ED(doc_0,doc′), of edit operations needed to transform doc_0 to doc′ . For simplicity, we consider two edit operations: insertion and deletion. Similarly, we normalize ED(doc0,doc′) by the maximum possible number of edit operations needed to modify doc_0 into a doc′ that yields a query that retrieves all objects in M. This quantity is estimated as |doc_0∪M .doc|, where M.doc =$\cup_{i=1}^{j} m_i.doc$. In other words, they just consider the keywords in M.doc, as

adding a keyword not in M.doc would make the set of query keywords less relevant to the user's query intention, i.e., less relevant to the missing objects.

In addition, the *Keyword-Adapted Why-Not Spatial Keyword Top-k Queries (KAWNQ)* is defined as follows:

Given a set D of spatial objects, a missing object set $M \subset D$, an original spatial keyword query $q = (loc, doc_0, k_0, \alpha)$, the *KAWNQ* query returns the refined query $q' = (loc, doc', k', \alpha)$, with the lowest penalty cost that referred above, and the result of which contains all objects in M.

They employ a hybrid index that estimates bounds on spatial distance and textual similarity at the same time. This index, called the SetR-tree, is a variant of the IR-tree.

As it is observed, most of the queries during their processing use separate indices for space and text, thus incurring the overhead of storing separate indices and joining their results. Others proposed a combined index that either inserts terms into a spatial structure or add a spatial structure to an inverted index.

In [17], a *Spatio-Temporal Textual Index (ST2I)* structure that supports the efficient evaluation of both range and top-k queries with multiple constraint types is presented. This indexing strategy uniformly handles text, space and time in a single structure, and is thus able to efficiently evaluate queries that combine keywords with spatial and temporal constraints. By using a single structure to index spatial, temporal and textual attributes together, ST2I is able to uniformly handle different constraint types and filter over multiple dimensions simultaneously, thus, reducing the number of irrelevant documents retrieved and consequently, query execution time. ST2I extends kd-trees, in which it employes a block-based storage at the leaf node level. This approach retains the flexibility of the kd-tree in supporting multiple dimensions and at the same time scales to large data sets that do not fit in main memory. Also, to incorporate text into this structure, ST2I uses an efficient technique to map textual information (terms) into numbers. This mapping must be strictly monotone so as to allow the inclusion of the mapped terms into a space-partitioning structure such as the kd-tree. They employ two algorithms to encode and decode the terms that have linear complexity in the size of the terms. The encoding and decoding operations are context free and can be applied on the fly, without requiring intermediate storage or hash tables. In addition, the approach supports evolving collections, where new terms are added dynamically.

Table 5: Overview of Other Queries.

Query	***Partitioning***
STPSJoin, kSTPSJoin	Grid
CSTQ	RSR-Tree
KAWNQ	SetR-Tree

2.2. Parallel / Distributed

The era of Big Data has created the need of parallel and distributed processing. Parallel and Distributed systems, are systems where computation is done in parallel, on multiple concurrently used computing units. They may be different cores of the same processor, different processors, or even different machines connected over a network. Apart from the problems that may have a parallel system, they must cope with many new problems, such as time synchronization, delays, communication problems between computing units, non-shared memory, load balance between the machines and so on. Thus, there are a lot of new frameworks, that support parallel batch processing such as, Spark and Hadoop (MapReduce), and others that support parallel stream processing such as, Spark Streaming and Storm.

2.2.1. Batch Processing

With the popularity of GPS and their applications, the size of spatio-textual data is increasing explosively, while the existing methods cannot deal with the spatio-textual massive data. MapReduce [33], a distributed shared-nothing data-processing framework, provides a method to deal with vast amount of data in a highly scalable and efficient fashion. The MapReduce framework is built on top of a cluster that is composed of multiple commodity machines. It allows to process massive data in parallel by splitting them into independent chunks.

In the sub-section 2.1.1.2., the Top-k Spatio-Textual Preference Query is presented. Due to the proliferation of the data, this query type must be implemented in a parallel and distributed system. Thus, in [31], the problem of *Parallel and Distributed Processing of Spatial Preference Queries Using Keywords (SPQ)* is studied, where the input data is stored in a distributed way. Given a set of keywords, a set of spatial data objects and a set of spatial feature objects that are additionally annotated with textual descriptions, the spatial preference query using keywords retrieves the top-k spatial data objects ranked according to the textual relevance of feature objects in their vicinity. This query type is processing-intensive, especially for large datasets, since any data objects may belong to the result set while the spatial range defines the score, and the k data objects with the highest score need to be retrieved. The proposed solution has two notable features:

(a) they propose a grid-based partitioning method that uses careful duplication of feature objects in selected neighboring cells and allows independent processing of subsets of input data in parallel, thus establishing the foundations for a scalable query processing algorithm, and

(b) they boost the query processing performance in each partition by introducing an early termination mechanism that delivers the correct result by only examining few data objects. Capitalizing on this, they implement parallel algorithms that solve the problem in the MapReduce framework.

In the following, the problem statement that solves this type of query is analyzed.

Given an object dataset O and a feature dataset F, which are horizontally partitioned and distributed to a set of servers, the *SPQ* returns the k data objects {p1, ..., pk} from O with the highest $\tau(p_i)$ scores.

The score $\tau(p)$ of p based on feature dataset F, given the range-based neighborhood condition r is defined as: $\tau(p) = \max\{w(f, q) \mid f \in F : d(p, f) \leq r\}$, where w(f, q) is the textual score which computed by the Jaccard similarity, and d(p, f) is the distance between p and f.

They propose an algorithm for solving the SPQ query, which relies on a grid-based partitioning of the 2-dimensional space in order to identify subsets of the original data that can be processed in parallel. It uses careful duplication of feature objects in selected neighboring cells, in order to create independent work units, by the following assumption.

Given a parallel/distributed spatial preference query using keywords with radius r, any feature object $f \in C_j$ must be assigned to all other grid cells $C_i (C_i \neq C_j)$, if $MINDIST(f, C_i) \leq r$.

In addition, they have proposed a thresholding mechanism that allows early termination of query processing, that guarantees the correctness of the result.

Another framework, that supports a distributed in-memory data management system for big spatio-textual data, is presented in [6], named as *LocationSpark*. It is built on top of Apache Spark [10], a widely used distributed data processing system. Spark is a distributed computation framework that allows users to work on distributed in-memory data without worrying about data distribution and fault-tolerance. LocationSpark offers a rich set of spatial query operators, e.g., range search, kNN, spatio-textual operation, spatial-join, and kNN-join.

LocationSpark stores spatial data as key-value pairs. A spatial tuple t_i, contains a spatial geometric key and a related value, namely k_i and v_i, respectively. The spatial data type of 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 can be specified by the user, e.g., a text type if the data tuple is a tweet.

It builds two layers of spatial indexes, global and local. The global index partitions data among the various nodes. To build a global index, LocationSpark samples the underlying data to learn the data distribution in space. Then, LocationSpark builds the global index to ensure that each data partition has the same amount of data. LocationSpark provides a grid and a region quadtree as the global index. In addition, each data partition has a local index, which is specified by the user. Here, for spatio-textual data, the IR-tree index is used.

Also, spatial data and queries are usually skewed. Some data partitions receive more queries than others. Thus, LocationSpark has a query scheduler to mitigate query skew. LocationSpark's query executor is responsible for choosing proper spatial algorithms based on the available spatial indexes and the registered queries. Skew is handled by distributing the load over the slave nodes. Furthermore, in order to reduce the communication cost when reading data that spans multiple partitions, LocationSpark uses a simple but efficient bloom filter, termed as sFilter. sFilter can detect whether a spatial object is inside the spatial range or not.

Memory is a precious resource for distributed in-memory data management systems. To deal with this situation, access frequencies and corresponding time stamps are recorded in the spatial index. Then, LocationSpark detects the frequently accessed data by aggregating access frequencies. Finally, it dynamically caches frequently accessed data into memory, and stores the less frequently used data into HDFS.

2.2.2. Streaming

The problem of processing a large amount of continuous spatial-keyword queries over streaming data, is essential in many applications such as location-based recommendation and advertising, thanks to the proliferation of geo-equipped devices and the ensuing location-based social media applications. While, there are several prior approaches aiming at providing efficient query processing techniques for the problem, their approaches belong to spatial-first indexing method which cannot well exploit the keyword distribution [30,28]. In addition, their textual filtering techniques are built upon simple variants of traditional inverted indexes, which do not perform well for the textual constraint imposed by the problem.

In [27], they investigate the problem of continuous spatial-keyword queries over spatial-textual stream, they address the above limitations and provide a highly efficient solution based on a adaptive index, named adaptive spatial-textual partition tree (AP-Tree). The AP-Tree adaptively groups registered queries using keyword and spatial partitions, guided by a cost model. Consider N denotes an AP-Tree node and there are three types of nodes: keyword node (k-node), spatial node (s-node), and query node (q-node). An intermediate node is a keyword (resp. spatial) node if keyword partition (resp. spatial partition) is adopted. They use f to denote the fanout of the intermediate node. A leaf node of AP-Tree corresponds to a q-node, and each query will be assigned to one or multiple query nodes according to its query region and ordered query keywords.

Keyword Node: They assume there is a total order among keywords in the vocabulary V, and keywords in each object and query are sorted accordingly. They delay the discussion of the effect of keyword order strategy to the experimental part. Queries assigned to a node N are partitioned into f ordered cuts according to their N_l-th keywords, where N_l is called the partition offset of the node N. They have $N_l \leq N_l^*$ if N^* is a descendant keyword node of N. An ordered cut is an interval of the ordered keywords, denoted as $c[w_i , w_j]$, where w_i and w_j ($w_i \leq w_j$) are boundary keywords. For presentation simplicity, they use $c[w_i]$ to denote $c[w_i , w_i]$ if there is only one keyword in the cut.

Spatial Node: The space is recursively partitioned by spatial nodes. Let N_r denote the region of a spatial node N, which will be divided into f grid cells. A query (q) on a spatial node N is pushed to a grid cell c if q.r overlaps c or contains c. Note that, unlike the keyword node in which a query is assigned to an unique cut, a spatial node may assign a query to multiple cells.

Cost model: Given a set Q of queries, AP-Tree is constructed in a top-down manner. Thus, they need to evaluate the goodness of a keyword or spatial partition such that the AP-Tree is

adaptive to query workload. In this subsection, we propose a cost model to quantitatively measure the matching cost for two partition methods. Given a node N and a set Q of queries assigned to N, without further partition the matching cost contributed by N is IQI assuming the average query verification cost is a unit time. Clearly, they can partition IQI queries into a set P of f buckets by keyword partition or spatial partition to reduce the matching cost. Throughout this paper, they might use bucket and cut, bucket and cell interchangeably for better understanding of the idea.

Let B denote a bucket of the partition, they use w(B) to record its weight which is the number of queries associated to B. By p(B) they mean the hit probability of the bucket B, (i.e., the probability that B is explored during the object matching). The expected matching cost regarding partition p, denoted by C(P), is as follows.

C(P) = $\sum_{i=1}^{f} w(B_i) \times p(B_i)$.

Given a partition P and a set of queries Q on the node, the calculation of w(B) is immediate for each bucket B. They may derive the hit probability p(B) based on some distribution assumptions or object workload. For analysis simplicity, they assume that p(B) = $\sum_{w \in B} p(w)$ for keyword node, where p(w) is the hit probability of the keyword w. In case a set O of the objects is available, it is trivial to derive hit probability of each individual keyword. Otherwise, they assume the query keyword with high frequency among Q has better chance to appear in object keywords; that is, they use query workload to simulate object workload. Specifically, they set p(w) = $\frac{freq(w)}{\sum_{w \in P} freq(w)}$, where *freq(w)* is the frequency of keyword w among all queries in Q. Regarding spatial partition, we may simply assume the uniform distribution of the object location, and hence p(B) = $\frac{Area(B)}{Area(N)}$, where Area(B) is the area of the bucket (i.e., cell) B and Area(N) is the region size of the node N. The hit probability calculation of each cell (bucket) is immediate when object workload is available.

The AP-Tree also naturally indexes ordered keyword combinations. They present index construction algorithm that seamlessly and effectively integrates keyword and spatial partitions.

Moreover, in [19], are described the *Publish/subscribe systems*, that enable efficient and effective information distribution by allowing users to register continuous queries with both spatial and textual constraints. However, the explosive growth of data scale and user base has posed challenges to the existing centralized publish/subscribe systems for spatio-textual data streams. Thus, they propose a distributed publish/subscribe system, called *PS^{2}Stream*, which digests a massive spatio-textual data stream and directs the stream to target users with registered interests. It achieves a better workload distribution in terms of both minimizing the total amount of workload and balancing the load of workers. To achieve this, they propose a new workload distribution algorithm considering both space and text properties of the data.

A spatio-textual object is defined as o = <text, loc>, where o.text is the textual content of object o and o.loc is the geographical coordinate, i.e., latitude and longitude, of object o.

They aim at building a distributed publish/subscribe system over a stream of spatio-textual objects. Users may express their interests on the spatio-textual objects with subscription

queries. Each subscription query contains a Boolean keyword expression and a region. If a new spatio-textual object falls in the specified region and satisfies the Boolean keyword expression specified by a subscription query, the object will be pushed to the user who submits the query. A subscription query is valid until the user drops it.

Spatio-Textual Subscription (STS) Query: A Spatio-Textual Subscription (STS) Query is defined as q = <K, R>, where q.K is a set of query keywords connected by AND or OR operators, and q.R denotes a rectangle region. A spatio-textual object o is a result of an STS query q if o.text satisfies the boolean expression of q.K and o.loc locates inside q.R.

The large number of STS queries and high arrival rate of spatio-textual objects call for a distributed solution. They build their system on a cluster of servers with several servers playing the role of dispatchers, which distribute the workload to other servers. The workload to their system includes the insertions and deletions of STS queries, and the matching operations between STS queries and spatio-textual objects.

- Query insertion: On receiving a new STS query, the worker inserts it into an in-memory index maintained in the worker.
- Query deletion: On receiving the indication of deleting an existing STS query, the worker removes the query from the index.
- Matching a spatio-textual object: On receiving a spatio-textual object, the worker checks whether the object can be a match for any STS query stored in the worker. If yes, the matching result is forwarded to the merger.

In [7], a distributed in-memory spatio-textual stream processing system that extends Storm, named as *Tornado*, is presented. To efficiently process spatio-textual streams, Tornado introduces a spatio-textual indexing layer to the architecture of Storm [8]. The indexing layer is adaptive, i.e., dynamically re-distributes the processing across the system according to changes in the data distribution and/or query workload. In addition to keywords, higher-level textual concepts are identified and are semantically matched against spatio-textual queries. Tornado provides data de-duplication and fusion to eliminate redundant textual data.

Stream processing in Storm is implemented using three main components; spouts, bolts, and topologies. A spout is a source of input data streams. A bolt is a data processing unit. A topology is a directed graph that connects spouts and bolts to form a stream processing pipeline. Apart from these three components, Tornado uses an adaptive indexing layer which ensures that queries are not replicated and that the data is sent only to the relevant bolts.

Indexing in Tornado is distributed and is composed of a global spatial index, and local spatio-textual indexes. All incoming data and queries navigate through the global index to be assigned to a query processing unit (i.e., a bolt). To avoid performance bottlenecks, the global index is replicated across several bolts. A local spatio-textual index is composed of multiple in-memory k-d trees. Each non-leaf node in the k-d tree is augmented with an inverted list that summarizes the textual contents of its child nodes. The inverted lists help speed-up the processing of the spatio-textual queries.

Tornado is adaptive to changes in both the data and query workload. It uses Apache ZooKeeper [9], an open-source distributed configuration and synchronization service, to

synchronize the changes in the global index bolt. Zookeeper stores usage statistics (i.e., the number of data objects and queries processed) from the data processing bolts. The index bolts access these usage statistics from the zookeeper to detect when a change in the index is needed.

3. Spatio-textual Similarity Join Query

In this section, we study the Spatio-textual Similarity Join Query which this thesis implements. We have referred the existing related work for this type of query and the systems that it has implemented. Moreover, we study the proposed techniques of partitioning in the sub-section 3.3.

3.1. Similarity Join on Centralized system

The set-similarity join has attracted significant interest. *Spatio-Textual Similarity Join Queries (ST-SJOIN)* find application in a wide range of domains, where spatial and textual information is available for a set of entities. The definition statement of a ST-SJOIN query is described below.

Given two collections of objects R and S that carry both spatial and textual information, the *ST-SJOIN* retrieves the subset J of R × S, such that for every (r, s) ∈ J, r is spatially close to s, based on a distance threshold (i.e., $dist_l(r, s) \leq \epsilon$, where $dist_l$ denotes distance between locations), and the set similarity between r and s also exceeds a threshold θ (i.e., $sim_t(r, s) \geq \theta$, where sim_t denotes textual similarity).

In [29], they propose an evaluation of such *ST-SJOIN* queries, where their techniques are orders of magnitude faster than baseline solutions. They define a spatio-textual object x as a triplet (x.id, x.loc, x.text), modeling the identity, the location, and the textual description of x, respectively. The entry x.loc takes values from the two-dimensional geographical space, while x.text is a set of terms drawn from a finite global dictionary $T = \{t_1, t_2, \ldots, t_n\}$. Each term t in x.text could carry a weight (default weights are 1 for unweighted sets), modeling the relevance of t to object x.

For every pair of spatio-textual objects x and y, they compute their spatial distance, $dist_l(x,y)$, with respect to x.loc and y.loc, as the Euclidean distance and their textual similarity, $sim_t(x,y)$, the set similarity between sets x.text and y.text, as the Jaccard similarity, $sim_t(x,y) = \frac{|x.text \cap y.text|}{|x.text \cup y.text|}$. Formally, given a collection of spatio-textual objects R, a spatial distance threshold e and a textual similarity threshold θ, ST-SJOIN(R, e, θ) retrieves all pairs (x, y) with x, y ∈ R, such that $dist_l(x, y) \leq e$ and $sim_t(x, y) \geq \theta$.

They use a dynamic grid partitioning, by extending the algorithm PPJoin to PPJoin-I.

Table 6: Overview of Similarity Queries.

Query	***Spatial Distance***	***Textual Similarity***	***Partitioning***
ST-SJOIN	Euclidean	Jaccard	Grid

3.2. Similarity Join on Parallel/Distributed system

In 3.1, the *Spatio-Textual Similarity Join (STSJ)* query has referred, and in [32], is proposed an efficient processing of this query using MapReduce. Given two collections of spatio-textual objects with a spatial location and textual descriptions, STSJ is to finds out all similar object pairs that have similar textual descriptions and are spatially close to each other. In [32], Jaccard coefficient and Euclidean distance are used as the measures to qualify the textual similarity and spatial similarity respectively, and their approaches can be easily extended to support other similarity measures. They are proposed several approaches for spatio-textual similarity join using MapReduce. The problem statement that has to solve the proposed methods is the following.

Given a collection of spatio-textual objects $R = \{r_1, r_2, \cdots, r_n\}$, where each object $r \in R$ includes a textual description r.text, which contains one or multiple tokens, and a spatial location r.loc that is represented by two-dimensional geographical coordinates a textual similarity threshold θ and a spatial distance threshold e, the STSJ(R, θ, e) aims to find all the similar pairs (r_i, r_j) where $sim_t(r_i, r_j) \geq \theta$ and $dist_l(r_i, r_j) \leq e$,

$STSJ(R, \theta, e) = \{(r_i, r_j) \mid r_i, r_j \in R, 1 \leq i, j \leq n, i \neq j, sim_t(r_i, r_j) \geq \theta, dist_l(r_i, r_j) \leq e\}$, where,

$sim_t(r_i, r_j)$ is the Jaccard similarity between two objects, and $dist_l(r_i, r_j)$ is the Euclidean distance of two objects.

Firstly, the tokens must be computed using MapReduce, which consists of two phases. In the first phase, the original records are read by the map function, and it extracts the textual part, tokenizes it into tokens and produces the output pairs for each token in the form of (token, 1). Then, the second phase, the reduce function, receives as input the tokens with a list of values, sums the values as count in the list for each token and outputs the results as (token, count), sorted them in the ascending order of the frequencies in the shuffle. In the following, the proposed methods are described.

- *Baseline Spatio-Textual Similarity Join (STSJ-B)*
- *Spatio-Textual Similarity Join with Cprefix (STSJ-C)*
- *Spatio-Textual Similarity Join with KD-Tree (STSJ-K)*

In the referred paper, the proposed methods are evaluated and the pros and cons are analyzed.

3.3. Partitioning Strategies for Spatio-Textual Similarity Join

In [18], partitioning strategies over spatio-textual objects for tackling STJoin, are explored. They propose two approaches and they evaluate each against the other. One approach is to start with a spatial data structure, traverse regions and apply the algorithm for identifying similar pairs of textual documents, called All-Pairs. The second approach is to construct a

global index, but partition postings spatially and modify the All-Pairs algorithm to prune candidates based on distance.

As it is mentioned before, in the spatio-textual similarity join (STJoin), we are given a collection of objects with both texts and geo coordinates and wish to efficiently identify all pairs of similar objects that are physically close. The two approaches are called Local Index Approach and Global Index Approach. In both strategies, we could construct either a quadtree or a grid, as the spatial data structure.

In the Local Index Approach, firstly, they build a PR-quadtree over the dataset. They use the spatial range of (-180, -90, 180, 90). Given a distance threshold t, they recursively decompose each node into four child nodes until the node contains less than b objects (by default, one) or the size of the node is about to less than the threshold t. Each node maintains a list of object ids. With this partitioning approach, when searching for similar objects for a target object x, only objects in the same or neighbor nodes of the target object need to be checked.

In the Global Index Approach, they build a global inverted index, and then partition each postings list spatially. Objects are sorted by its z-order in each postings list. Thus, when they are applying the All-Pairs algorithm, instead of iterating over the entire postings list, they only need to consider the objects within certain range of z-orders. An efficient optimization, is that we can avoid checking the pair (y, x) if we've already checked (x, y).

4. Technologies

Due to the large volume of spatio-textual data, we need a big data technology to manage these data in parallel. In this thesis, we have used Apache Spark, a distributed / parallel system.

4.1. Apache Spark

Spark was initially started by Matei Zaharia at UC Berkeley's AMPLab in 2009, and open sourced in 2010 under a BSD license. Apache Spark has as its architectural foundation the resilient distributed dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way. In Spark 1.x, the RDD was the primary application programming interface (API), but as of Spark 2.x use of the Dataset API is encouraged even though the RDD API is not deprecated. The RDD technology still underlies the Dataset API.

Spark and its RDDs were developed in 2012 in response to limitations in the MapReduce cluster computing paradigm, which forces a particular linear dataflow structure on distributed programs: MapReduce programs read input data from disk, map a function across the data, reduce the results of the map, and store reduction results on disk. Spark's RDDs function as a working set for distributed programs that offers a (deliberately) restricted form of distributed shared memory.

Spark facilitates the implementation of both iterative algorithms, that visit their data set multiple times in a loop, and interactive/exploratory data analysis, i.e., the repeated database-style querying of data. The latency of such applications may be reduced by several orders of magnitude compared to a MapReduce implementation (as was common in Apache Hadoop stacks). Among the class of iterative algorithms are the training algorithms for machine learning systems, which formed the initial impetus for developing Apache Spark.

Apache Spark requires a cluster manager and a distributed storage system. For cluster management, Spark supports standalone (native Spark cluster), Hadoop YARN, or Apache Mesos. For distributed storage, Spark can interface with a wide variety, including Hadoop Distributed File System (HDFS), MapR File System (MapR-FS), Cassandra, OpenStack Swift, Amazon S3, Kudu, or a custom solution can be implemented. Spark also supports a pseudo-distributed local mode, usually used only for development or testing purposes, where distributed storage is not required and the local file system can be used instead; in such a scenario, Spark is run on a single machine with one executor per CPU core.

Apache Spark consists of five main parts, Spark Core, Spark SQL, Spark Streaming, Machine Learning Library MLlib, and GraphX.

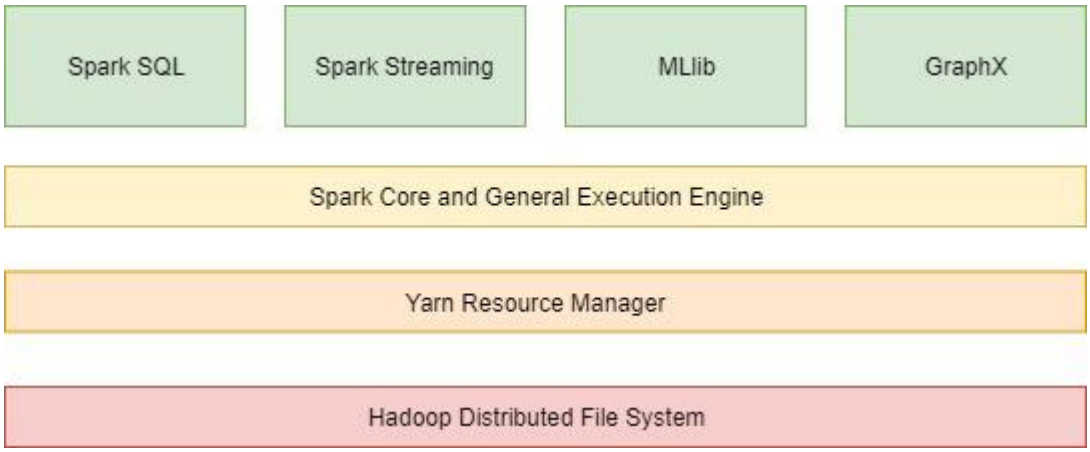

Figure 2: Architecture of Apache Spark.

4.1.1. Spark core

Spark Core is the foundation of the overall project. It provides distributed task dispatching, scheduling, and basic I/O functionalities, exposed through an application programming interface (for Java, Python, Scala, and R) centered on the RDD abstraction (the Java API is available for other JVM languages, but is also usable for some other non-JVM languages, such as Julia, that can connect to the JVM). This interface mirrors a functional/higher-order model of programming: a "driver" program invokes parallel operations such as map, filter or reduce on an RDD by passing a function to Spark, which then schedules the function's execution in parallel on the cluster. These operations, and additional ones such as joins, take RDDs as input and produce new RDDs. RDDs are immutable and their operations are lazy; fault-tolerance is achieved by keeping track of the "lineage" of each RDD (the sequence of operations that produced it) so that it can be reconstructed in the case of data loss. RDDs can contain any type of Python, Java, or Scala objects.

Besides the RDD-oriented functional style of programming, Spark provides two restricted forms of shared variables: broadcast variables reference read-only data that needs to be available on all nodes, while accumulators can be used to program reductions in an imperative style.

4.1.2. Spark SQL

Spark SQL is a component on top of Spark Core that introduced a data abstraction called DataFrames, which provides support for structured and semi-structured data. Spark SQL provides a domain-specific language (DSL) to manipulate DataFrames in Scala, Java, or Python. It also provides SQL language support, with command-line interfaces and ODBC/JDBC server. Although DataFrames lack the compile-time type-checking afforded by RDDs, as of Spark 2.0, the strongly typed DataSet is fully supported by Spark SQL as well.

4.1.3. Spark Streaming

Spark Streaming uses Spark Core's fast scheduling capability to perform streaming analytics. It ingests data in mini-batches and performs RDD transformations on those mini-batches of data. This design enables the same set of application code written for batch analytics to be used in streaming analytics, thus facilitating easy implementation of lambda architecture. However, this convenience comes with the penalty of latency equal to the mini-batch duration. Other streaming data engines that process event by event rather than in mini-batches include Storm and the streaming component of Flink. Spark Streaming has support built-in to consume from Kafka, Flume, Twitter, ZeroMQ, Kinesis, and TCP/IP sockets.

In Spark 2.x, a separate technology based on Datasets, called Structured Streaming, that has a higher-level interface is also provided to support streaming.

4.1.4. MLlib Machine Learning Library

Spark MLlib is a distributed machine-learning framework on top of Spark Core that, due in large part to the distributed memory-based Spark architecture, is as much as nine times as fast as the disk-based implementation used by Apache Mahout (according to benchmarks done by the MLlib developers against the alternating least squares (ALS) implementations, and before Mahout itself gained a Spark interface), and scales better than Vowpal Wabbit. Many common machine learning and statistical algorithms have been implemented and are shipped with MLlib which simplifies large scale machine learning pipelines, including:

- summary statistics, correlations, stratified sampling, hypothesis testing, random data generation.
- classification and regression: support vector machines, logistic regression, linear regression, decision trees, naive Bayes classification.
- collaborative filtering techniques including alternating least squares (ALS).
- cluster analysis methods including k-means, and latent Dirichlet allocation (LDA).
- dimensionality reduction techniques such as singular value decomposition (SVD), and principal component analysis (PCA).
- feature extraction and transformation functions.
- optimization algorithms such as stochastic gradient descent, limited-memory BFGS(L-BFGS).

4.1.5. GraphX

GraphX is a distributed graph-processing framework on top of Apache Spark. Because it is based on RDDs, which are immutable, graphs are immutable and thus GraphX is unsuitable for graphs that need to be updated, let alone in a transactional manner like a graph database. GraphX provides two separate APIs for implementation of massively parallel algorithms (such as PageRank): a Pregel abstraction, and a more general MapReduce style API. Unlike its predecessor Bagel, which was formally deprecated in Spark 1.6, GraphX has full support for property graphs (graphs where properties can be attached to edges and vertices).

GraphX can be viewed as being the Spark in-memory version of Apache Giraph, which utilized Hadoop disk-based MapReduce.

Like Apache Spark, GraphX initially started as a research project at UC Berkeley's AMPLab and Databricks, and was later donated to the Apache Software Foundation and the Spark project.

5. Problem Statement

This thesis deals with finding similar spatio-textual objects from two sets of spatio-textual data, based on given radius and similarity thresholds. The type of query is the *Spatio-Textual Similarity Join (STSJ)* which is analyzed in Section 3 for centralized systems and for the parallel/distributed systems with batch processing. This type of query has been studied in previous research and the problem statement is referred below.

Given a collection of spatio-textual objects $R = \{r_1, r_2, \cdots, r_n\}$, where each object $r \in R$ includes a textual description r.text, and a spatial location r.loc that is represented by two-dimensional geographical coordinates a textual similarity threshold θ and a spatial distance threshold e, the STSJ(R, θ, e) aims to find all the similar pairs (r_i, r_j) where $sim_t(r_i, r_j) \geq \theta$ and $dist_l(r_i, r_j) \leq e$

$$STSJ(R, \theta, e) = \{(r_i, r_j) \mid r_i, r_j \in R, 1 \leq i, j \leq n, i \neq j, sim_t(r_i, r_j) \geq \theta, dist_l(r_i, r_j) \leq e\}$$

where $sim_t(r_i, r_j)$ is the Jaccard similarity between two objects, and $dist_l(r_i, r_j)$ is the Euclidean distance of two objects.

Spatio-textual objects are described from their location and a text, as it is shown in *Figure 3*. These objects achieve a similarity join depending on the distance and textual thresholds. For example, let distance threshold be 0,3 and textual threshold be 0,65. In the next figure, the only points which achieve a similarity join are x_1 and x_2, because:

$dist(x_1, x_2) <= 0,3$ and

$sim(x_1, x_2) = 2/3 = 0,66 >= 0,65$.

All other occasions do not achieve a similarity join, because as we can notice, the textual similarity is lower than 0,65.

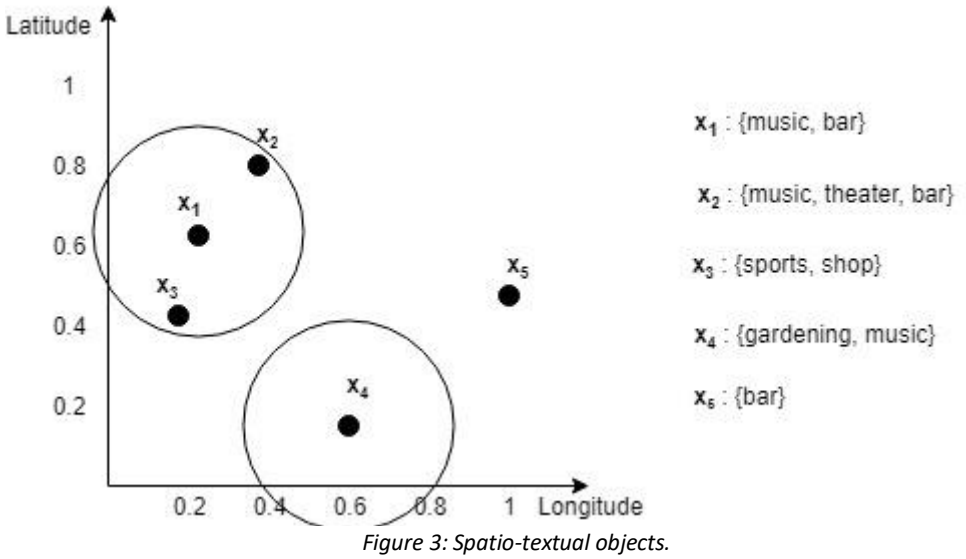

Figure 3: Spatio-textual objects.

These two functions that we have used for the similarity join, could vary depending on the user desires. There are a lot of these measure functions, which some of them are described in chapter 2.

5.1. Euclidean distance

The Euclidean distance between points p and q is the length of the line segment connecting them. In Cartesian coordinates, if p = (x_1, y_1) and q = (x_2, y_2) are two points in Euclidean 2-space, then the distance d(p, q) from p to q, or from q to p, is given by the formula:

d(p, q) = d(q, p) = $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

For instance, let p = (2, -1) and q = (-4, 7). The Euclidean distance of these two points is:

d(p, q) = $\sqrt{(2 - (-4))^2 + (-1 - 7)^2}$ = $\sqrt{36 + 64}$ = 10

5.2. Jaccard similarity

The Jaccard similarity of the text of two sets A and B, which have a whole set of words, is the size of intersection divided by size of union of two sets. The formula that computes the Jaccard similarity is the following:

J (A, B) = $\frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A|+|B|-|A \cap B|}$,

where,

|A ⋂ B| is the number of the intersecting words of A and B,

|A| is the number of words of A,

|B| is the number of words of B.

For example, given two sentences:

A: AI is our friend and it has been friendly.

B: AI and humans have always been friendly.

In order to calculate similarity using Jaccard similarity, we will first perform lemmatization to reduce words to the same root word. In our case, “friend” and “friendly” will both become “friend”, “has” and “have” will both become “has”. Drawing a Venn diagram of the two sentences we get:

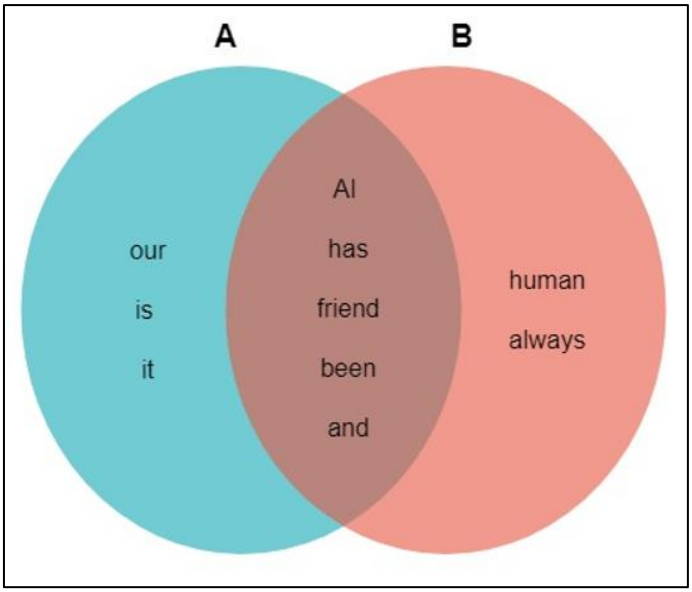

Figure 4: Venn diagram of Jaccard similarity.

For the above two sentences, depicted in Figure 3, we get Jaccard similarity $\frac{5}{5+3+2}$ = 0.5, which is size of intersection of the set divided by total size of set.

6. Analysis of implementation

In this thesis, partitioning strategies for spatio-textual objects are studied in order to achieve the best load balancing. The two partitioning strategies which have been implemented are based on the spatial part of the object. The first implementation is the partitioning of data based on an Horizontal separation, whereas the second implementation is the partitioning of data based on a Regular Grid.

In each partition, two algorithms which compute the STSJ query have been developed in order to find similar objects. The first algorithm is a “Brute Force” algorithm, which checks Euclidean distance and Jaccard similarity for each point of dataset “A” against all points of dataset “B”. The second algorithm is based on Plane Sweep, which checks for each point only those that are nearest based on a given radius.

It is important to be cleared that if we desire to compute the Spatio-textual similarity join query with parallel processing, one technique of partitioning and one algorithm for the query processing is necessary.

6.1. Techniques of partitioning

Two techniques of partitioning have been developed, in order to find which of these achieve the best load balancing to the workers. These techniques are presented below, called as Horizontal Separation and Regular Gird.

6.1.1. Partitioning data based on Horizontal Separation

During the first implementation, an horizontal separation of spatial data based on latitude is defined. First of all, for the Spatio-Textual Similarity Join Query, we have two datasets, A and B. Every dataset is read as a Spark Dataframe. For each dataset, a new column filling with “A” or “B” is created that represents if the dataset is the A or the B. Then, the two Dataframes are united and sorted by the latitude column.

Afterwards, we perform lemmatization in the column that corresponds to the text. Lemmatization is a process in which words are cut and only the main body of each word retains. For example, if we have the words “Human”, “humanity”, and “humanities”, the main body of each word is the “human”, and it remains. As can be seen, the capital letters become small. Moreover, there are a lot of stop-words in each sentence, which must be excluded from the textual similarity, because they do not give any information. These kinds of words are usually articles, pronouns and conjunctions. We used a list of 179 English stop-words.

Then, a column that shows the number of the row must be created, in order to make the partitions. As it is referred, the table is sorted by latitude and is cut based on the number of partitions, which the user gives. We first compute the size of the Dataframe and we divide it

with the number of partitions and save this number as “x”. Therefore, the corresponding field of a record depends on the integral part of the division “x” with the number of the row. For example, let the size of the Dataframe be 3.000 records and the defined number of partitions is 10, and so “x” = 3.000/10 = 300. Then, the partition in which each point corresponds is computed by the integer part of the division row_number/300. Thus, the first 300 points will be matched to the partition “0”, the next 300 points will be matched to the partition “1”, and so on.

Figure 5, captures the finally table that is created after above processing. The explanation of the datasets which have used, are presented in Section 7. The first column is the id of the record, latitude and longitude are the geographical coordinates, DatasetID shows if the record belongs to dataset A or B, Lemma is column that shows the keywords after lemmatization, IncreasingID captures the row number of the record and grid shows the partition which this record belongs.

Id	latitude	longitude	DatasetID	Lemma	IncreasingId	grid
50634	1.0E-4	60.0365	A	kv3 kv5 kv2 kv6 k...	0	0
50634	1.0E-4	60.0365	B	kv1 kv5 kv3 kv6 kv4	1	0
45957	0.0014	59.3364	A	kv2 kv4 kv3 kv1 k...	2	0
191027	0.0014	60.4163	A	kv2 kv3 kv1 kv4 k...	3	0
45957	0.0014	59.3364	B	kv2 kv1 kv6 kv4 kv3	4	0
52160	0.0017	80.4599	A	kv2 kv4 kv3 kv5 k...	5	0
52160	0.0017	80.4599	B	kv6 kv5 kv2 kv3 kv1	6	0
95659	0.0018	44.9637	A	kv3 kv1 kv5 kv6	7	0
95659	0.0018	44.9637	B	kv2 kv4 kv6 kv5 kv3	8	0
57457	0.0023	34.328	A	kv5 kv4 kv7 kv3 k...	9	0
57457	0.0023	34.328	B	kv1 kv3 kv6 kv2 k...	10	0
22922	0.0024	10.4518	A	kv1 kv2 kv7 kv3 k...	11	0
22922	0.0024	10.4518	B	kv2 kv1 kv5	12	0
139159	0.0026	55.6212	A	kv3 kv2 kv7 kv1 kv4	13	0
42673	0.0031	21.7334	A	kv1 kv4 kv5 kv6	14	0
42673	0.0031	21.7334	B	kv5 kv6 kv2 kv3 k...	15	0

Figure 5: Structure of the creating table.

Next step is the duplication of the points of the dataset B, based on the radius of search that the user gives. There are some points that may be close enough, but they are on different partitions. In the next figure *(Figure 6)***,** every point p_i is described with its coordinates (x_i, y_i) and “A” or “B”, depending on the dataset that it belongs. There are three partitions, “0”, “1” and “2”. As you can see, the point p_1 belongs in the partition “1”, as the p_2. Given a radius (r), a circle with center the point p_1, which pertains to dataset B, is formed. As it appears in the next figure, the points p_2, p_3 and p_4 of dataset A belong in this circle. This means, that these three points have Euclidean distance from point p_1 lower than r. However, p_2 and p_4, belong in different partitions, so p_1 must be duplicated to partitions “0” and “2”, so as it can be processed with other points of dataset A.

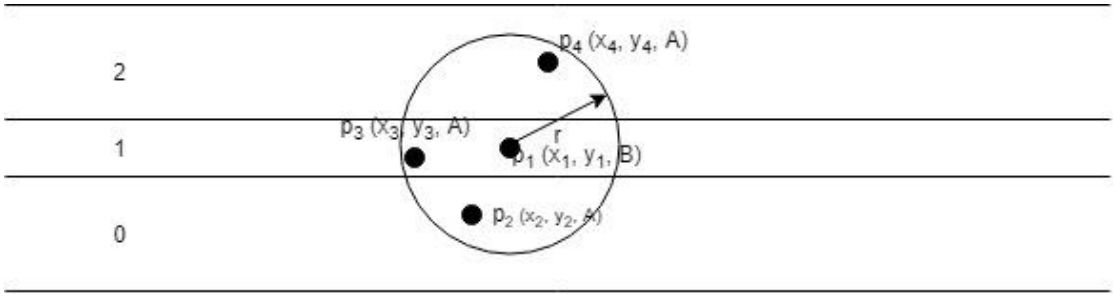

Figure 6: Duplication of points in Horizontal separation.

Thus, based on the initial creation of partitions, we have to duplicate these points to the corresponding partitions which are nearest because of the radius, in order to have these points in the same partition, regardless of the fact that they are not. Firstly, we have to compute the upper value of the latitude in each partition. Then, given the number of the cell (i.e. partition) that corresponds each point of the dataset B, we check for each point of the dataset B, all the previous cells in sequence if the distance (latitude – radius) is lower than the upper value of the latitude of the previous cell. If it is true, the number of the cell (i.e. partition) is appending in the list of cells that is nearest to the point. If it is false, we break the repetitive search. Subsequently, we check all the next cells in sequence if the distance (latitude + radius) of the point is greater than the upper value of the latitude of the next cell. If it is true, the number of the cell (i.e. partition) is appending in the list of cells that is nearest to the point. If it is false, we break the repetitive search.

In the next figure *(Figure 7)*, there are 10 partitions and diverse points of dataset A or B. We can observe the partition that each point belongs. As a result of the above processing, p_1 which belongs to dataset B and to partition "0", will be duplicated in "1", because of the defined radius. Such kind of duplication will be done for points p_4, p_6. However, p_8 will not be duplicated to partition "8", though the radius shows that it had to, because it belongs to dataset A and it has been referred that only the points of dataset B will be copied.

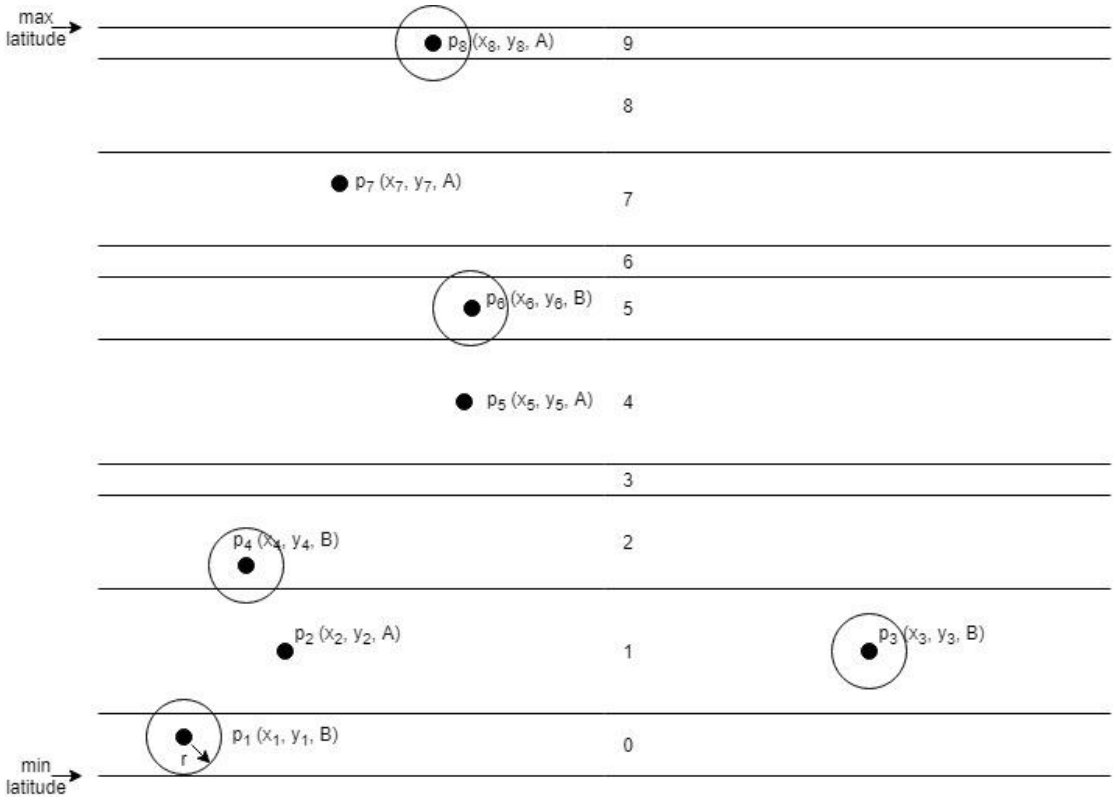

Figure 7: Duplication of points of Dataset B in the whole space, that is created with Horizontal separation.

Last step of the duplication of the points in the cells, is to duplicate the entire row with each value of the partition list which has created. For instance, if a point due to the above processing has the list [0, 1, 2] in the column that shows the corresponding partition, we duplicate the entire row of the point 3 times. One for the partition "0", one for the partition "1" and one for the partition "2".

With the above processing, we have created the number of partitions that corresponds each value. Therefore, we can send each point to the partition, after first finding the distinct values of the partitions.

Algorithm 1 describes the way that Horizontal Separation make the partitioning to the workers.

```
1: Input p(d1, d2, r, splits)
2: Function Horizontal_Separation:
3: union d1, d2 as d
4: sort d   // by latitude
5: for x ϵ d do
6:     l <- lemma(d(text))   //perform lemmatization on text
7: end for
8: size <- size(d) / splits
9: for x ϵ d do
10:     cellid <- int(row_number(x) / size)
11: end for
12: // duplicate points of dataset B to the nearest cells
13: bounds <- bounds(cells)
14: for x ϵ d do
15:     if x ϵ d(B) then
16:         if x(lat) + r > bounds or x(lat) – r < bounds then
17:             duplicate point to these cells
18:         end if
19:     end if
20: end for
21: end Function
```

Algorithm 1: Pseudocode of Horizontal Separation.

6.1.2. Partitioning data based on Regular Grid

In the second implementation, data are partitioned by a regular grid, which is created based on the coordinates of the first dataset. A Regular Grid is consisted of equal parallelograms, which are the result of splitting the longitude axis to equal parts, and the latitude axis to equal parts. In the case that the parts of longitude and latitude are equal, then the parallelograms become squares.

First of all, we define which dataset will be the “A” and which the “B”. Then, in the dataset “A”, we compute the minimum and the maximum values of the longitude and latitude. The minimum point of the Regular grid, will be the (min longitude, min latitude). The maximum point of the Regular grid, would be the (max longitude, max latitude).

Furthermore, we compute the equal parts that the longitude and the latitude axis have been divided. Initially, we define the number of parts in which the two axis will be splitted. If the splits of the longitude are 10, and so for latitude, the dimension of the Regular grid will be 10x10. As a result of this creation of grid, the Regular grid will contain 100 cells. The length of the parts (i.e. the step) in longitude and latitude is computed by the following two formulas:

- stepLon = (maxLon - minLon) / splits
- stepLat = (maxLat - minLat) / splits

where,

maxLon and maxLat are the maximum values of latitude,

minLon and minLat are the minimum values of longitude, and

splits are the number of parts, in which axis is separated.

Moreover, we use another formula in order to compute in which cell of the Regular grid, the point matches. Each cell will be named with a unique id number, from 0 to dimension of Regular Grid minus 1. Thus, if we have a 5x5 Regular grid, the ids of the cell will range from 0 to 24. The formula which computes the above processing is the following:

cellID = int((x - minLon) / stepLon) * splits + int((y - minLat) / stepLat)

where,

minLon, stepLon, splits, minLat and stepLat, have referred above, and

x, y are the coordinates of the checked point.

The above formula has a peculiarity, which corresponds to the maximum point of the regular grid to another cell outside the grid. In order to avoid this, we marginally increase the maximum values of longitude and latitude of the Regular grid. Thus, the maximum point of the grid has become (max longitude + 0.0000001, max latitude + 0.0000001).

In the following, we have to duplicate points of dataset B to the corresponding cells based on the radius, such as in the first implementation. Each partition in the cluster will be a unique cell of the grid. The computation in each cell of the nearest and similar points is independent on the others. Thus, there may be points that are close enough and have high Jaccard similarity, but they belong to different cells. To avoid this problem, we have to duplicate points of dataset B to the cells that are nearest because of the radius.

Next figure *(Figure 8)*, captures the reason why some points must be duplicated. For example, p_1 belongs to cell with number 8. Given a radius by a user, a circle is created which captures the point that is near enough. Thus, p_1 seems to be near to some points of the cells 7, 9, 12, 13, 14, and the point is copied to these cells. For the same reason, p_2 must be copied to cells 10, 11, 15, 17, 20, 21.

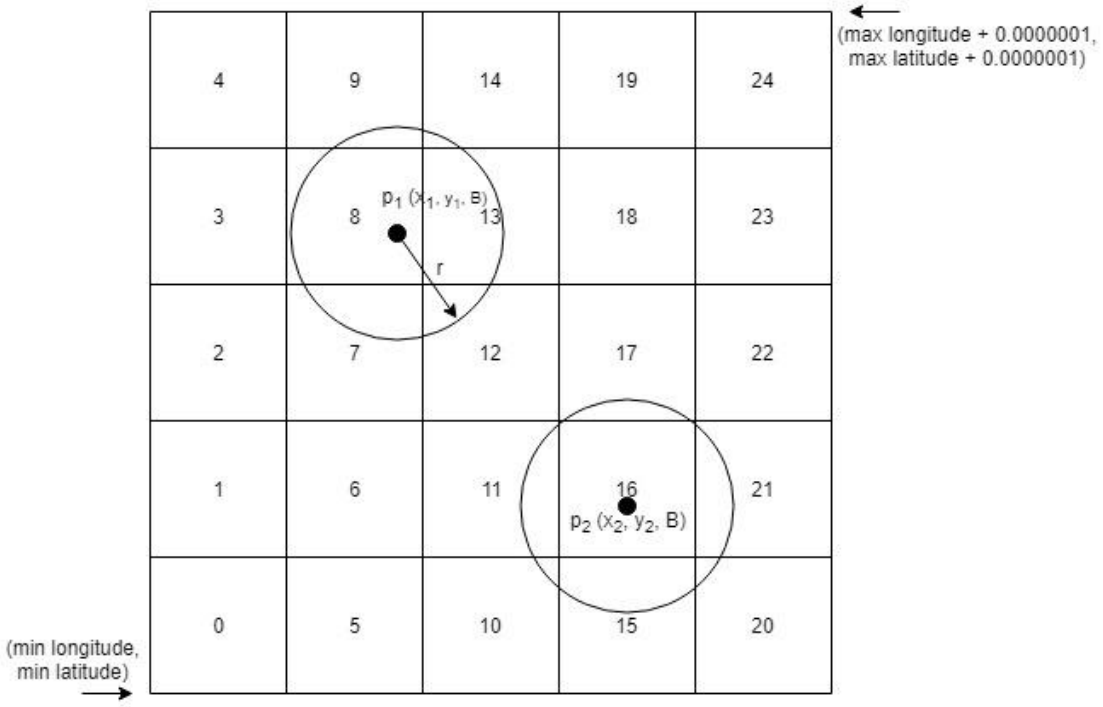

Figure 8: Duplication of points in Regular Grid.

Algorithm 2 describes the way that Regular Grid technique achieves the partitioning of data.

```
1: Input p(d1, d2, r, splits)
2: Function Regular_Grid:
3: // find the minimum and maximum values of longitude and latitude in dataset A
4: maxLat <- max(d1(lat)) + 0.000001
5: maxLon <- max(d1(lon)) + 0.000001
6: minLat <- min(d1(lat))
7: minLon <- min(d1(lon))
8: // compute the step in each axis
9: stepLon <- (maxLon - minLon) / splits
10: stepLat <- (maxLat - minLat) / splits
11: for x ϵ d1 do
12:     cellid <- int((x(lon) - minLon) / stepLon) * splits + int((x(lat) - minLat) / stepLat)
13: end for
14: for x ϵ d2 do
15:     cellid <- int((x(lon) - minLon) / stepLon) * splits + int((x(lat) - minLat) / stepLat)
16:     // duplicate points of dataset B to the nearest cells
17:     if x near enough to other cells based on r then
18:             duplicate point to these cells
19:     end if
20: end for
21: union d1, d2 as d
22: for x ϵ d do
23:     l <- lemma(d(text))   //perform lemmatization on text
24: end for
25: end Function
```

Algorithm 2: Pseudocode of Regular Grid.

6.2. Query processing

The query processing has been developed with two different algorithms, in order to find which of these achieve the best performance. These algorithms are performed below, called as Brute Force and Plane Sweep.

6.2.1. Brute Force Algorithm

The Brute Force algorithm computes the Euclidean distance and Jaccard similarity in each partition, for each point of dataset B, against all points of dataset A.

First of all, the records of each partition are stored in a table. Then, each point of dataset "B" is checked in sequence against all points of dataset "A". Each time, the Euclidean distance and the Jaccard similarity is computed in the same step. Thus, if a point of dataset "B" is near and similar enough with a point of dataset "A", we have a join.

In the next figure *(Figure 9)*, let the parallelogram be a partition which has diverse points of datasets A and B. Due to the above explanation, each point of dataset A is checked with all points of dataset B. This exhaustive search is terminated when all points of dataset A are checked against all of dataset B.

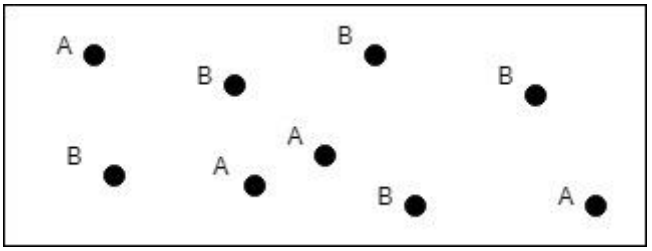

Figure 9: Query processing with Brute Force.

The complexity of this algorithm is O(M*N), where M is the number of points of dataset A and N is the number of points of dataset B.

Algorithm 3 shows how Brute Force computes the similarity join query.

```
1: Input q(pos, text, d, r, e)
2: Function Brute_Force:
3: for x ϵ d(A) do
4:     for y ϵ d(B) do
5:         if dist(x,y) <= r and sim(x,y) >= e then
6:             Achieve a join
7:         end if
8:     end for
9: end for
10: end Function
```

Algorithm 3: Pseudocode of Brute Force.

6.2.2. Plane Sweep Algorithm

Plane Sweep algorithm sort all values by longitude and compute the distance until the longitude of a point is greater than the longitude of the checked point plus the radius. Then, for these points, the Jaccard similarity is computed.

Firstly, the records in each partition are stored in a table. The table is sorted by the longitude coordinate of the points. Then, we check each point against its next, till the longitude of a point is greater than the longitude of the checked point adding the given radius. Initially, for these points, the Euclidean distance is computed and if the points are close enough, the Jaccard similarity is computed. If the Jaccard similarity is equal or greater than a given number, we have a join.

The advantage of this algorithm is that we do not compute Euclidean distance and Jaccard similarity for each point against all, but we stop the searching when the longitude of a point is greater than the longitude of the checked point adding the defined radius. Moreover, an efficient optimization, is that we can avoid checking the pair (y, x) if we've already checked (x, y). Thus, we reduce the complexity of the algorithm.

In the next figure *(Figure 10)*, we will explain with an example the way that Plane Sweep compute the similarity join query. All points are sorted by the longitude coordinate of the points. Given a radius (r), a circle which has in the center the first point is formed. As you can observe, in the created circle, there are two points of dataset B which belongs to. Thus, the Plane Sweep algorithm will only check these two points, and will terminate the searching for the first point. This process is repeated for all the points.

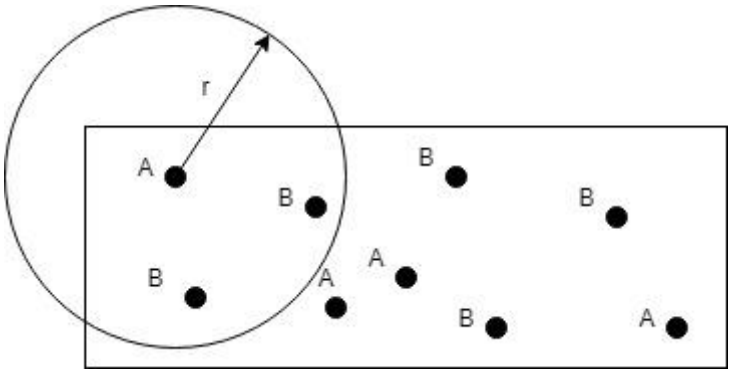

Figure 10: Query processing with Plane Sweep.

The complexity of Plane Sweep will vary depending on the defined radius. The initially sort of the table has a complexity of O(nlog(n)), where n is the total number of points in the partition. Because of the early termination, we expect a lower complexity than the complexity of Brute Force.

Algorithm 4 captures how Plane Sweep algorithm computes the similarity join query.

```
1: Input q(pos, text, d, r, e)
2: Function Plane_Sweep:
3: sort d    //by longitude
4: for x ϵ d do
5:     for y ϵ d+1 do   // y takes values from next points of x
6:         If y(lon) < x(lon) + r then
7:             if dist(x,y) <= r then
8:                 if sim(x,y) >= e then
9:                     Achieve a join
10:                end if
11:            end if
12:        else
13:            break
14:        end if
15:    end for
16: end for
17: end Function
```

Algorithm 4: Pseudocode of Plane Sweep.

7. Experimental setup

In this section, we present the platform in which we run our experiments and the hardware specs. Also, we will describe the metrics which have used in order to evaluate the performance of the algorithms. Furthermore, we will present the datasets that have used for the experiments in section 8.

All algorithms are implemented in Python and more specific we have used the PySpark library for the parallel processing.

7.1. Platform

We deployed our algorithms in a Spark cluster consisting of 5 nodes. Each of the nodes has 8 GB of RAM, 1 disk with 60 GB for HDFS and 4 CPUs. All nodes run Ubuntu 16.04.

7.2. Evaluation metrics

There are diverse metrics that could be evaluated, such as the load balancing in each partition, to avoid some workers having a high load of computation. Moreover, we measure the time that each algorithm needs to be terminated and compute the similarity join. Also, we measure the number of times that the functions Euclidean distance, Jaccard similarity are used.

In the following, the metrics are listed following by the reason of choosing them.

- Time: we used this metric in order to find which algorithm achieves the fastest computation.
- Load balancing: it is used so as to evaluate the techniques of partitioning.
- Standard deviation: this metric will also show which technique of partitioning is the best.
- Counts of Euclidean distance function: this metric presents the number of times that this function is used.
- Counts of Jaccard similarity function: this metric presents the number of times that this function is used.
- Counts of similarity joins: it measures the number of points that are similar.

7.3. Datasets

We have evaluated our implementations in diverse datasets, in order to measure the metrics which have referred above. We have used datasets form the real-world life, so as to observe the performance in a real-life problem. Moreover, we have created a synthetic dataset which follows a uniform distribution, in order to achieve our algorithms in these datasets.

7.3.1. Real datasets

We evaluate the implementation in a real-life problem, in which the real-life points do not typically follow uniform distribution. The datasets have been downloaded from TripAdvisor.

The first dataset contains the ratings of users in diverse locations and the structure that this file has is presented below:

- Rating: from 0 to 5 stars.
- Review: description of location.
- Title: name of location.
- Rtitle: summary of rating in a few words.
- Address: address of location.
- Latitude: geographical latitude of location.
- Longitude: geographical longitude of location.

The second dataset contains the ratings of users for diverse restaurants and the structure of this dataset is seemed below:

- Rating: from 0 to 5 stars.
- Review: description of restaurant.
- Title: name of restaurant.
- Rtitle: summary of rating in a few words.
- Address: address of restaurant.
- Latitude: geographical latitude of restaurant.
- Longitude: geographical longitude of restaurant.

Table 7: Summary table of real datasets.

Parameter	Dataset A	Dataset B
Size	20 MB	11 MB
Records	50.000	30.000
Distribution	Random	Random

7.3.2. Synthetic datasets

Also, we have evaluated our implementation in datasets that follow a uniform distribution and have high textual similarity. Datasets have been created with a spatio-textual data generator, given the parameters that show below *(Table 8)*.

Table 1: Given parameters in the data generator.

Parameter	Description	Value A	Value B
Entries	How many entries are generated	200.000	100.000
Precision	Precision of the decimal part	4	4
Max dimension	Maximum dimension of points	100	100
Min textual objects	Minimum number of keywords that contains a point	4	3
Max textual objects	Maximum number of keywords that contains a point	7	6

The structure of these datasets is the same and it is presented below:

- Id: id of the record.
- Latitude: the latitude coordinate of the point.
- Longitude: the longitude coordinate of the point.
- Text: keywords which describe the record.

Table 8: Summary table of synthetic datasets.

Parameter	Dataset A	Dataset B
Size	10 MB	5 MB
Records	200.000	100.000
Distribution	Uniform	Uniform

8. Experimental study

In this chapter, we evaluate the performance of the two technics of partitioning and the two algorithms which compute the similarity join query.

It is divided in two experiments, where in the first we use real datasets for the evaluation, in order to notice the performance in a real-life problem.

Afterwards, the best technique of partitioning and the best algorithm of similarity join are chosen, and they are evaluated on synthetic data, which follow a uniform distribution.

8.1. First experiment

In the first experiment, two real datasets that have been downloaded from TripAdvisor are used. We evaluate the implementation in a real-life problem, in which the real-life points do not typically follow uniform distribution.

In the next table *(Table 9)*, the parameters that have been used for the first experiment are described.

Table 9: Parameters of the first experiment.

Parameter	Description	Value
r	Radius of search	1°
e	Threshold of textual similarity	0.7
d	Splits of space	25

8.1.1. Comparison of Horizontal Separation VS Regular Grid

In this subsection, we compare the techniques of partitioning, in order to achieve the best load balancing.

Figure 11 shows the load balancing achieved by the Horizontal separation in contrast with Regular grid. Each partition captures the volume of data which contains.

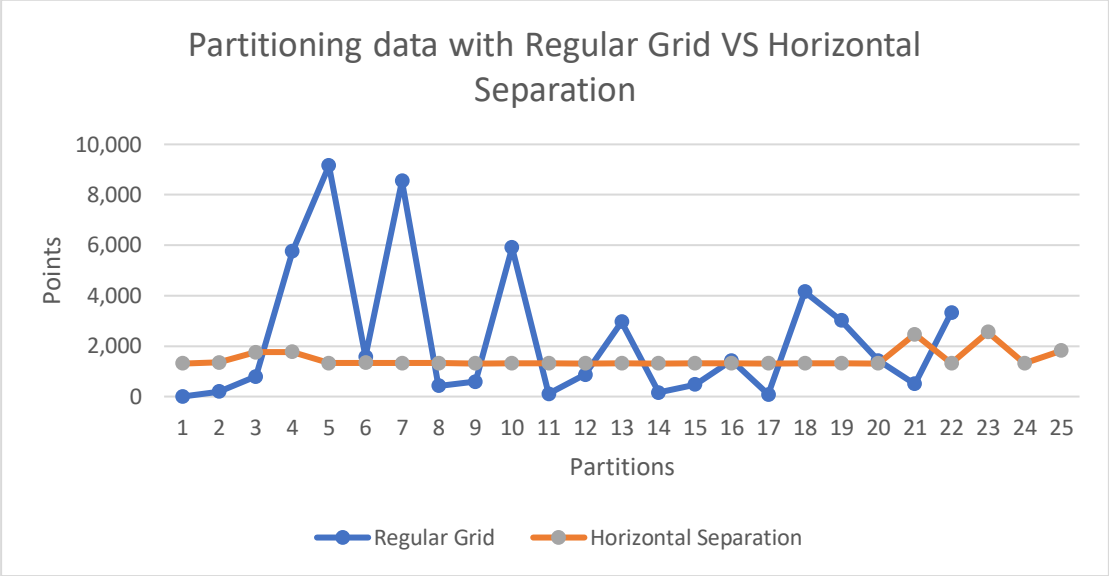

Figure 11: Comparison of partitioning techniques.

We can monitor that the Horizontal separation has achieved a very good load balancing, because the line, which describes the number of elements in each partition, seems to be almost straight.

Partition data by Regular Grid, seems to be an unbalanced way, because the line which describes the volume of data in each partition is a crooked line. Also, even if we have defined the splits equal to 25 (d =25), which means that we have a 5x5 regular grid, there are three parallelograms in grid which have been empty.

Another measure that proves which technique achieves the best load balancing is the standard deviation *(Table 10).* We observe that Horizontal Separation achieves low standard deviation in contrast with the Regular Grid.

Table 10: Standard deviation of partitioning.

Partitioning Technique	Standard Deviation
Horizontal Separation	348
Regular Grid	2758

For the above explanation, we choose that the best technique, which achieves the best load balancing, is the Horizontal Separation. This technique will be used in the second experiment.

8.1.2. Comparison of Plane Sweep VS Brute Force

In this subsection, we compare the two algorithms which compute the similarity join query, Plane Sweep and Brute Force. We will compare the time that each algorithm needs in order to be terminated and show the results. Also, we will measure the number of times that each function (Euclidean distance and Jaccard similarity) is applied to.

In the following figure *(Figure 12)*, the execution time that each algorithm needs until the termination, is captured. We can observe that the fastest algorithm is the Plane Sweep and the best partitioning technique which make the fastest results is the Horizontal Separation. Furthermore, we can notice huge differences between time in each algorithm and the fastest is the Plane Sweep.

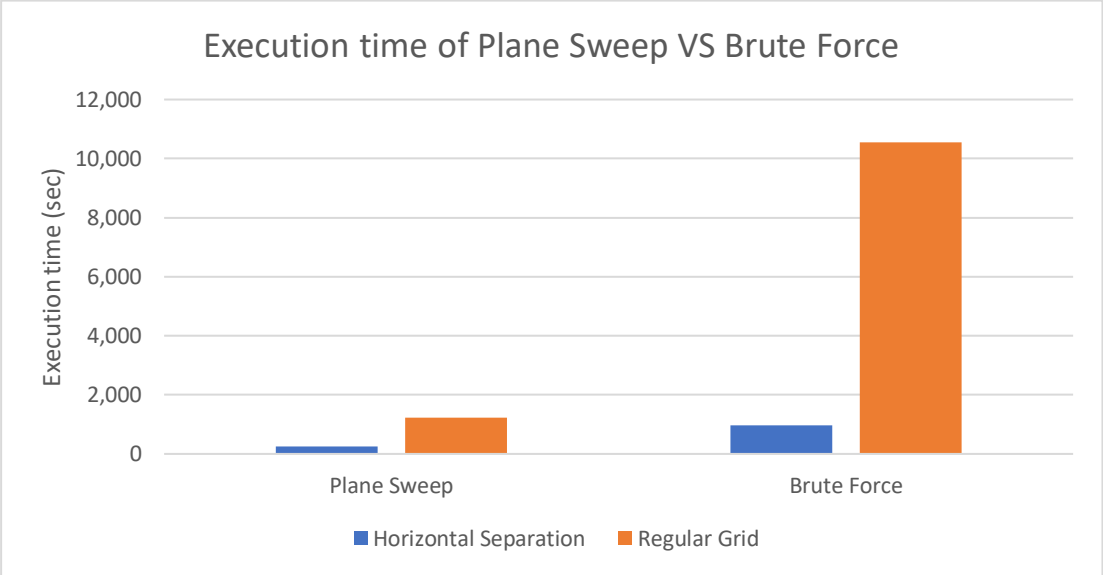

Figure 12: Total execution time of Plane Sweep VS Brute Force.

Moreover, the total counts of Euclidean vs Jaccard function that has applied to the Horizontal Separation, are captured in *Figure 13*. It has applied a logarithmic scale to the y-axis, because of the large deviations. As it has referred above, in Plane Sweep, the Euclidean distance is applied first, and if the points are close enough given a radius, then the Jaccard distance is applied. Thus, it is logical that the Euclidean function has used more times than the Jaccard.

On the other hand, in Brute Force, the Euclidean and Jaccard function are applied at the same time. So, the total counts for these functions are the same.

We can observe that in Horizontal Separation, the Plane Sweep algorithm needs less computation to achieve the result, as we expect given the way that Plane Sweep works with the early termination.

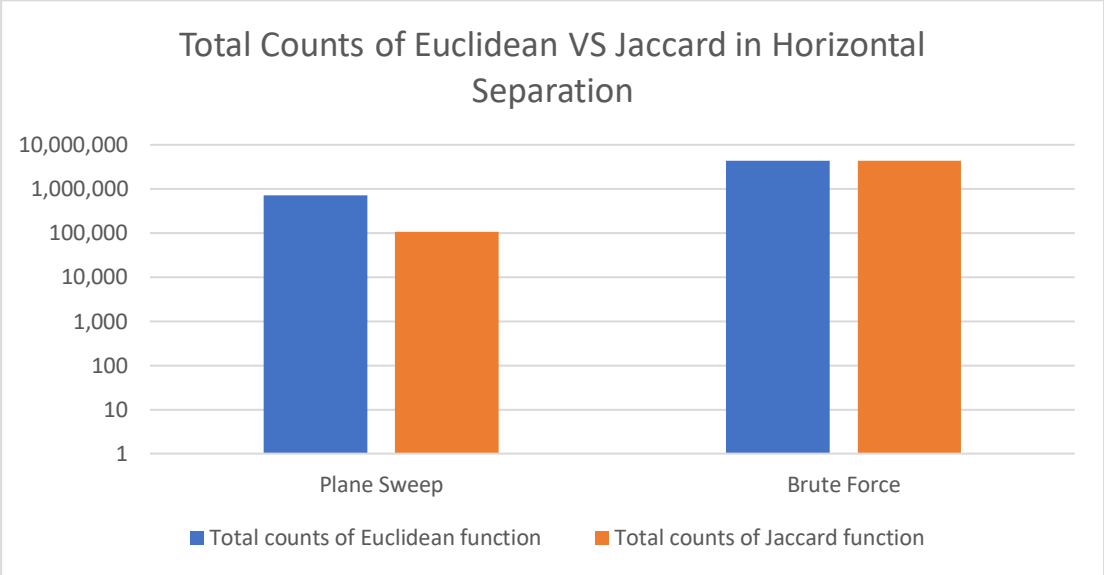

Figure 13: Total counts of Euclidean VS Jaccard function in Horizontal Separation.

For the Regular Grid, we can see the total number of counts for the functions in the next figure *(Figure 14)*. Again, it has applied a logarithmic scale in y-axis, in order to have a better visualization. Because of the dividing of data has become in a blind way, without taking the data into account, the points may be far enough to each other. Thus, we can observe that counts in Plane Sweep are a few, because the searching terminates faster when the points are far enough. On the other hand, in Brute Force, the counts of functions are not affected from the distance of the points, because we check each point against all of the same partition that they belong.

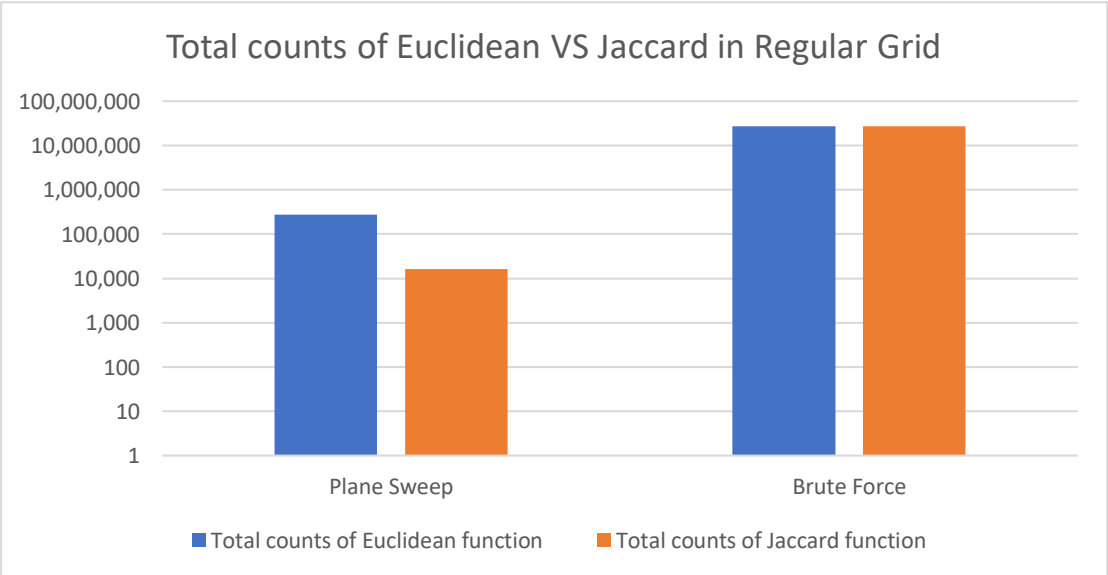

Figure 14: Total counts of Euclidean VS Jaccard function in Regular Grid.

Last but not least, the dataset that we have used in the first experiment, has in the text description a lot of words. So, it is difficult enough to achieve a similarity join, when the similarity threshold is high. For this reason, we will execute a second experiment, using data that follow a uniform distribution and has in the text description a few keywords, which in many times would have a high similarity.

Due to the first experiment, taking account the performance of the algorithms and the techniques of data dividing, we choose the Horizontal Separation technique for the partitioning and the Plane Sweep for the similarity join.

8.2. Second experiment

In the next table *(Table 9)*, the parameters of the second experiment are described. We have chosen a small radius, because of the distribution that the datasets follow and we divide the space into 20 parts.

Table 11: Parameters of the second experiment.

Parameter	Description	Value
r	Radius of search	0.5°
e	Threshold of textual similarity	0.7
d	Splits of space	20

The load balancing of the partitions is captured in *Figure 15*. As we can observe, the partitioning of the data seems to be balanced enough. The anomalies that we notice, are because of the duplication of the data in the nearest cells based on the radius. In these partitions, may there are a lot of points that are near to the limits that have been created during the Horizontal separation.

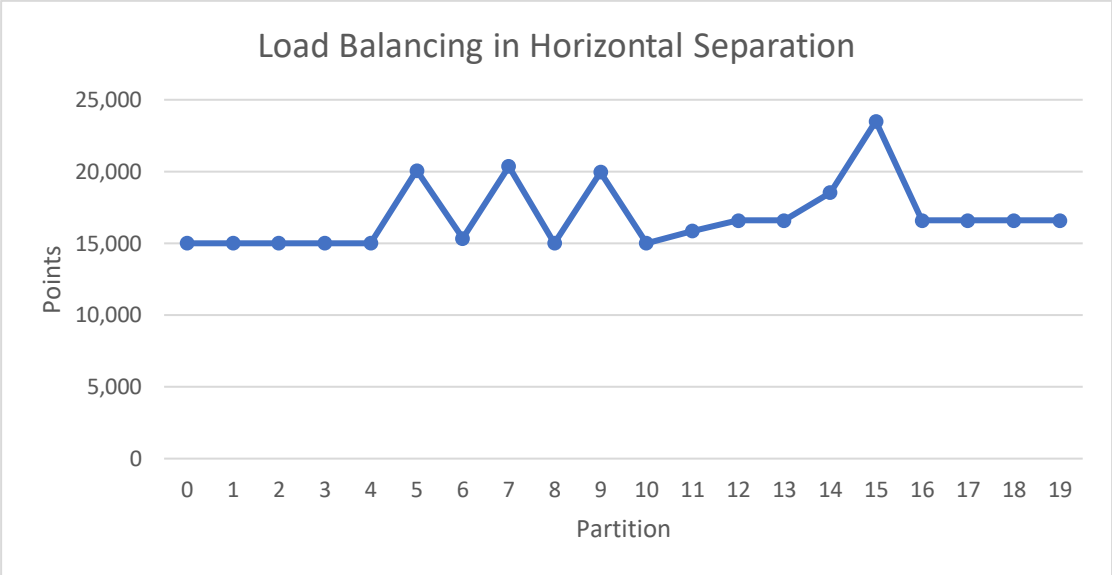

Figure 15: Load balancing in Horizontal Separation.

Figure 16-17, shows the number of times that the Euclidean distance and Jaccard similarity functions have been used. As it has referred above, in Plane Sweep, the Euclidean distance is applied first and if the data are near enough, then the Jaccard similarity is applied. Thus, it is logical that the counts of Euclidean function are high enough than the counts of the Jaccard. Also, the number that the functions are applied to, are independent of the data that each partition contains, because the points may not be close enough.

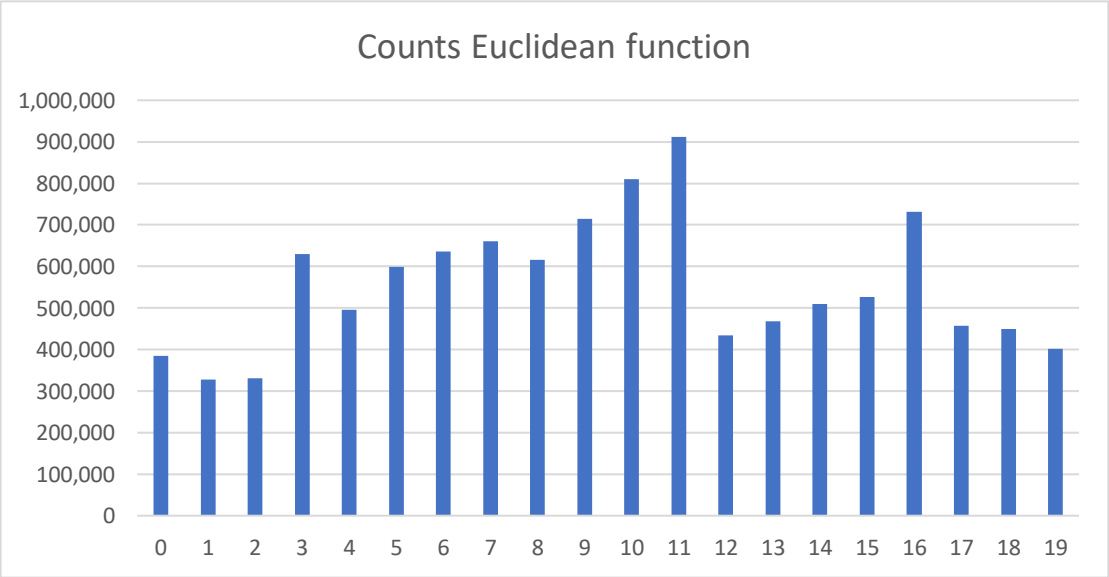

Figure 16: Counts of Euclidean distance function in each partition.

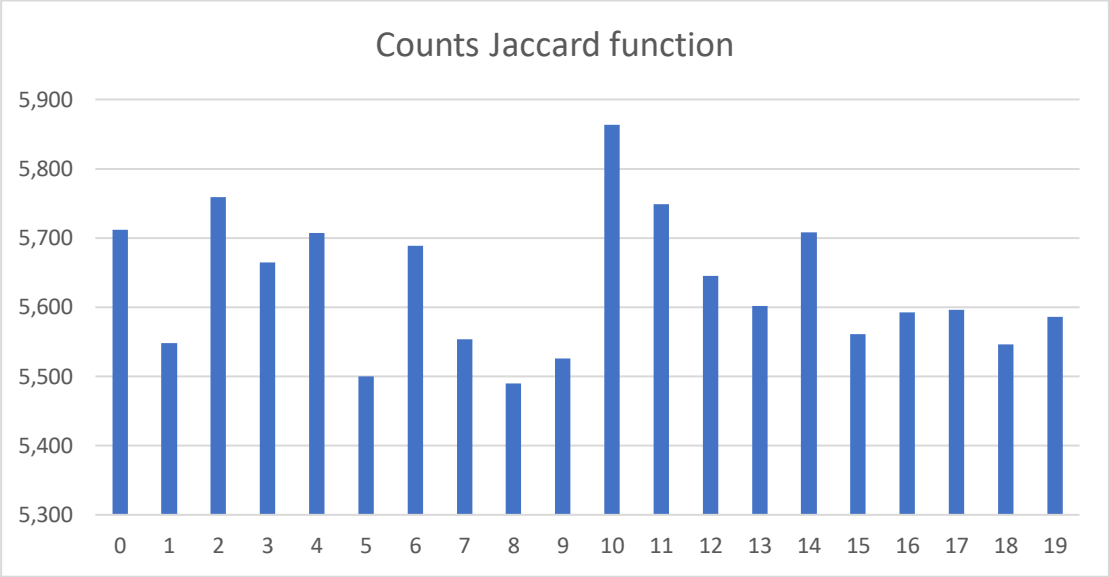

Figure 17: Counts of Jaccard similarity function in each partition.

In *Figure 18*, the number of similarity points in each partition are figured. This number of joins is dependent on the similarity threshold that is defined.

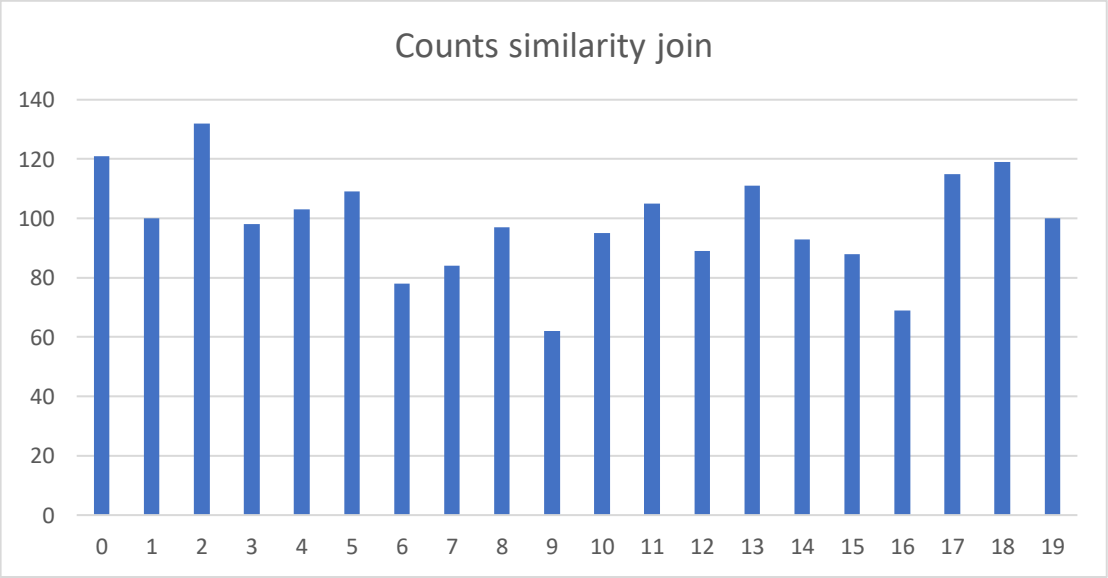

Figure 18: Counts of similarity join points in each partition.

Last figure (Figure 19), is the execution time that each partition needs in order to compute the similarity join query.

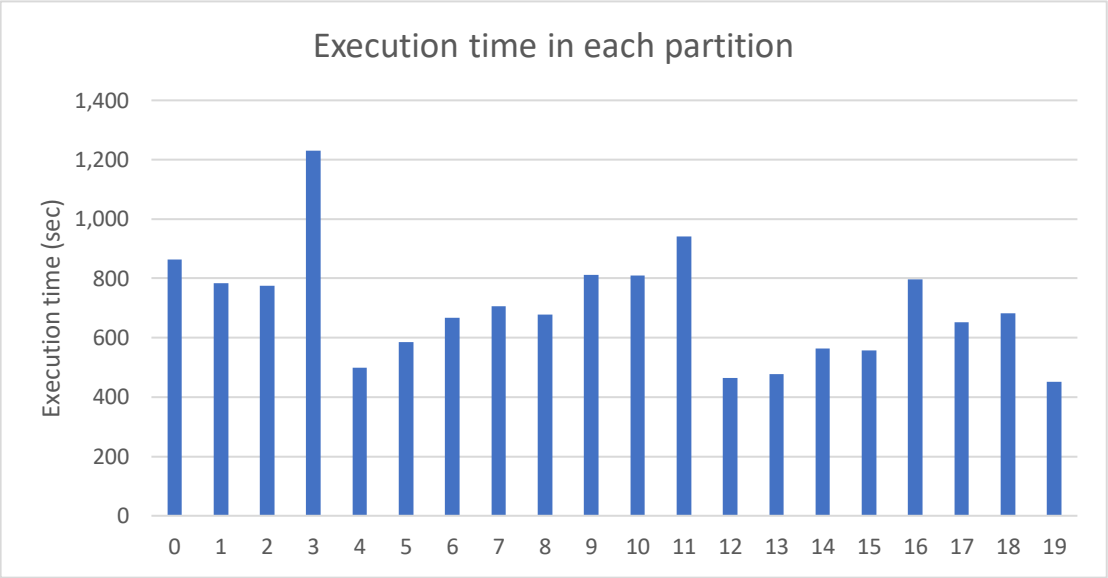

Figure 19: Execution time of Plane Sweep in each partition.

9. Conclusion

In conclusion, we have studied the related work that have been done in the research community in spatio-textual queries. We have implemented the similarity join query, and we examine different algorithms, Plane Sweep and Brute Force, to compute this type of query, in order to find which of these achieve the best performance.

Due to the large volume of spatio-textual data, the main problem which we have met, was the load balancing of data in the workers. We have implemented two different techniques of partitioning, Horizontal Separation and Regular Grid. The partitioning is based on the spatial part of the records, which after a lot of experiments seem to achieve better load balancing than the partitioning based on the text. Regular Grid makes the partitions without see the data, whereas Horizontal Separation is dynamically creates the partitions based on the points of the datasets.

Because of the experiments that have been done in the chapter 5, we propose that the best technique of partitioning is the Horizontal Separation. Also, the fastest algorithm of computing the similarity join query is the Plane Sweep.

Future work in this implementation, may be to examine the performance of the algorithm in a really huge volume dataset.

10. References

[1] Krishan K. Arya, Vikram Goyal, Shamkant B. Navathe, Sushil K. Prasad: Mining Frequent Spatial-Textual Sequence Patterns. DASFAA (2) 2015: 123-138.

[2] Farhana Murtaza Choudhury, J. Shane Culpepper, Zhifeng Bao, Timos Sellis: Batch Processing of Top-k Spatial-Textual Queries. ACM Trans. Spatial Algorithms and Systems 3(4): 13:1-13:40 (2018).

[3] Farhana Murtaza Choudhury, J. Shane Culpepper, Timos K. Sellis, Xin Cao: Maximizing Bichromatic Reverse Spatial and Textual k Nearest Neighbor Queries. PVLDB 9(6): 456-467 (2016).

[4] Farhana Murtaza Choudhury, J. Shane Culpepper, Timos K. Sellis: Batch processing of Top-k Spatial-textual Queries. GeoRich@SIGMOD 2015: 7-12.

[5] Orestis Gkorgkas, Akrivi Vlachou, Christos Doulkeridis, Kjetil Nørvåg: Maximizing Influence of Spatio-Textual Objects Based on Keyword Selection. SSTD 2015: 413-430.

[6] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, Walid G. Aref: LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. PVLDB9(13): 1565-1568 (2016).

[7] Ahmed R. Mahmood, Ahmed M. Aly, Thamir Qadah, El Kindi Rezig, Anas Daghistani, Amgad Madkour, Ahmed S. Abdelhamid, Mohamed S. Hassan, Walid G. Aref, Saleh M. Basalamah: Tornado: A Distributed Spatio-Textual Stream Processing System. PVLDB 8(12): 2020-2023(2015).

[8] Storm. https://storm.apache.org/, 2015.

[9] ZooKeeper. https://zookeeper.apache.org/, 2015

[10] Spark. https://spark.apache.org/, 2015.

[11] Lisi Chen, Gao Cong, Christian S. Jensen, Dingming Wu: Spatial Keyword Query Processing: An Experimental Evaluation. PVLDB 6(3): 217-228 (2013).

[12] Gao Cong, Christian S. Jensen: Querying Geo-Textual Data: Spatial Keyword Queries and Beyond. SIGMOD Conference 2016: 2207-2212.

[13] Gary Marchionini: Exploratory search: from finding to understanding. Commun. ACM 49(4): 41-46 (2006).

[14] Xin Lin, Jianliang Xu, Haibo Hu: Reverse Keyword Search for Spatio-Textual Top-k Queries in Location-Based Services. IEEE Trans. Knowl. Data Eng. 27(11): 3056-3069 (2015).

[15] Jiaheng Lu, Ying Lu, Gao Cong: Reverse spatial and textual k nearest neighbor search. SIGMOD Conference 2011: 349-360.

[16]George Tsatsanifos, Akrivi Vlachou: On Processing Top-k Spatio-Textual Preference Queries. EDBT 2015: 433-444.

[17] Tuan-Anh Hoang-Vu, Huy T. Vo, Juliana Freire: A Unified Index for Spatio-Temporal Keyword Queries. CIKM 2016: 135-144.

[18] Jinfeng Rao, Jimmy J. Lin, Hanan Samet: Partitioning strategies for spatio-textual similarity join.BigSpatial@SIGSPATIAL 2014: 40-49.

[19] Zhida Chen, Gao Cong, Zhenjie Zhang, Tom Z. J. Fu, Lisi Chen: Distributed Publish/Subscribe Query Processing on the Spatio-Textual Data Stream. ICDE 2017: 1095-1106.

[20] Christodoulos Efstathiades, Alexandros Belesiotis, Dimitrios Skoutas, Dieter Pfoser: Similarity Search on Spatio-Textual Point Sets. EDBT 2016: 329-340.

[21] Sitong Liu, Yaping Chu, Huiqi Hu, Jianhua Feng, Xuan Zhu: Top-k Spatio-textual Similarity Search. WAIM 2014: 602-614.

[22] Ahmed R. Mahmood, Walid G. Aref: Query Processing Techniques for Big Spatial-Keyword Data. SIGMOD Conference 2017: 1777-1782.

[23] Junling Liu, Ke Deng, Huanliang Sun, Ge Yu, Xiaofang Zhou, Christian S. Jensen: Clue-based Spatio-textual Query. PVLDB 10(5): 529-540 (2017).

[24] Gao Cong, Christian S. Jensen, Dingming Wu: Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects. PVLDB 2(1): 337-348 (2009).

[25] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, Sergei Vassilvitskii: Scalable K-Means++. PVLDB 5(7): 622-633 (2012).

[26] Dong-Wan Choi, Chin-Wan Chung: A K-partitioning algorithm for clustering large-scale spatio-textual data. Inf. Syst. 64: 1-11(2017).

[27] Xiang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, Wei Wang: AP-Tree: Efficiently support continuous spatial-keyword queries over stream. ICDE 2015: 1107-1118.

[28] Guoliang Li, Yang Wang, Ting Wang, Jianhua Feng: Location-aware publish/subscribe. KDD 2013: 802-810.

[29] Panagiotis Bouros, Shen Ge, Nikos Mamoulis: Spatio-textual similarity joins. PVLDB 6(1): 1-12 (2012).

[30] Lisi Chen, Gao Cong, Xin Cao: An efficient query indexing mechanism for filtering geo-textual data. SIGMOD Conference 2013: 749-760.

[31] Christos Doulkeridis, Akrivi Vlachou, Dimitris Mpestas, Nikos Mamoulis: Parallel and Distributed Processing of Spatial Preference Queries using Keywords. EDBT 2017: 318-329.

[32] Yu Zhang, Youzhong Ma, Xiaofeng Meng: Efficient Spatio-textual Similarity Join Using MapReduce. WI-IAT (2) 2014: 52-59.

[33] Jeffrey Dean, Sanjay Ghemawat: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1): 107-113 (2008).

[34] Anders Skovsgaard, Christian S. Jensen: Finding top-k relevant groups of spatial web objects. VLDB J. 24(4): 537-555 (2015).

[35] Xin Cao, Gao Cong, Tao Guo, Christian S. Jensen, Beng Chin Ooi: Efficient Processing of Spatial Group Keyword Queries. ACM Trans. Database Syst. 40(2): 13:1-13:48 (2015).

[36] Xin Cao, Gao Cong, Christian S. Jensen, Beng Chin Ooi: Collective spatial keyword querying. SIGMOD Conference 2011: 373-384.

[37] Dingming Wu, Christian S. Jensen: A Density-Based Approach to the Retrieval of Top-K Spatial Textual Clusters. CIKM 2016: 2095-2100.

[38] Xike Xie, Xin Lin, Jianliang Xu and Christian S. Jensen. Reverse Keyword-Based Location Search. ICDE 2017: 375-386.

[39] Lei Chen, Jianliang Xu, Xin Lin, Christian S. Jensen, Haibo Hu: Answering why-not spatial keyword top-k queries via keyword adaption. ICDE 2016: 697-708.

[40] Adriane Chapman, H. V. Jagadish: Why not? SIGMOD Conference 2009: 523-534.

[41] Lei Chen, Xin Lin, Haibo Hu, Christian S. Jensen, Jianliang Xu: Answering why-not questions on spatial keyword top-k queries. ICDE2015: 279-290.

[42] Yunjun Gao, Qing Liu, Gang Chen, Baihua Zheng, Linlin Zhou: Answering Why-not Questions on Reverse Top-k Queries. PVLDB 8(7): 738-749 (2015).

[43] Thaleia Dimitra Doudali, Ioannis Konstantinou, Nectarios Koziris: Spaten: A spatio-temporal and textual big data generator. BigData 2017: 3416-3421.