Παράλληλη και κατανεμημένη επεξεργασία ερωτημάτων σε χώρο-λεκτικά δεδομένα

(Parallel and distributed processing of spatial preference queries)

Η
ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Υποβάλλεται στην ορισθείσα
από την Γενική Συνέλευση Ειδικής Σύνθεσης
του Τμήματος Μηχανικών Η/Υ και Πληροφορικής
Εξεταστική Επιτροπή

από τον
Μπέστα Δημήτριο

ως μέρος των υποχρεώσεων για την απόκτηση του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Πανεπιστήμιο Ιωαννίνων
Φλεβάρης 2017

ACKNOWLEDGMENTS

Αφιερώνεται στην οικογένειά μου για την στήριξή τους καθ’ όλη την διάρκεια των σπουδών μου.

CONTENTS

Acknowledgments ii
Contents iii
List of tables iv
List of figures v
Εκτεταμένη Περίληψη vi
Abstract viii
Chapter 1. Introduction 11
Chapter 2. Related Work 14
2.1. Spatial preference queries 14
2.2. Top-k spatial preference queries 15
2.3. Set similarity joins 15
2.4. Spatio-textual search 17
Chapter 3. Parallel and distributed processing of spatial preference queries using keywords 19
3.1. Preliminaries 19
3.2. Problem statement and overview of solution 21
3.3. Grid-based partitioning and initial algorithm 26
3.4. Algorithms with early termination 31
3.5. Theoretical results 39
3.6. Experimental evaluation 47
3.7. Performance improvement techniques 66
3.8. Emulator 71
Chapter 4. Conclusions – Further work 74
4.1. Conclusions 74
4.2. Further work 75
References 77
Short Vita 84

LIST OF TABLES

Table 1: Overview of symbols 21
Table 2: Example of datasets and scores for query $q.W$ = {italian} 24
Table 3: Experimental parameters (default values in bold) 47
Table 4: Emulator output 1/4 71
Table 5: Emulator output 2/4 72
Table 6: Emulator output 3/4 73
Table 7: Emulator output 4/4 73

LIST OF FIGURES

Figure 1: Example of spatial preference query using keywords (*SPQ*). 23
Figure 2: Example of grid partitioning 26
Figure 3: Algorithm 1: *pSPQ*: Map function 28
Figure 4: Algorithm 2: *pSPQ*: Reduce function 31
Figure 5: Algorithm 3: *eSPQlen*: Map function 34
Figure 6: Algorithm 4: *eSPQlen*: Reduce function with Early Termination 35
Figure 7: Algorithm 5: *eSPQsco*: Map function 37
Figure 8: Algorithm 6: *eSPQsco*: Reduce function 38
Figure 9: Spatial distribution of flickr dataset 49
Figure 10: Spatial distribution of twitter dataset 49
Figure 11: Spatial distribution of clustered dataset 50
Figure 12: Duplication of feature objects 51
Figure 13: Flickr dataset grid size experiment results 53
Figure 14: Flickr dataset keywords experiment results 53
Figure 15: Flickr dataset range experiment results 54
Figure 16: Flickr dataset top-*k* parameter experiment results 55
Figure 17: Twitter dataset grid size experiment results 56
Figure 18: Twitter dataset keywords experiment results 56
Figure 19: Twitter dataset range experiment results 57
Figure 20: Twitter dataset top-*k* parameter experiment results 57
Figure 21: Scalability of our algorithms performance when varying
the dataset size 58
Figure 22: Uniform dataset grid size experiment results 59
Figure 23: Uniform dataset keywords experiment results 59
Figure 24: Uniform dataset range experiment results 60
Figure 25: Uniform dataset top-*k* parameter experiment results 60
Figure 26: Clustered dataset grid size experiment results 61
Figure 27: Clustered dataset keywords experiment results 62
Figure 28: Clustered dataset range experiment results 63
Figure 29: Clustered dataset top-*k* parameter experiment results 63
Figure 30: *eSPQsco* execution time based on keyword selection (1/2) 64
Figure 31: *eSPQsco* execution time based on keyword selection (2/2) 65
Figure 32: Clustered file with 5x5 static grid 66
Figure 33: Clustered file with 5x5 dynamic grid 67
Figure 34: Gains by using mbrs or dynamic grid 69

ΕΚΤΕΤΑΜΕΝΗ ΠΕΡΙΛΗΨΗ

Όνομα: Δημήτριος Μπέστας του Ιωάννη και της Μαρίας
Είδος τίτλου: MSc
Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων
13/2/2017
Τίτλος Διατριβής: Παράλληλη και κατανεμημένη επεξεργασία ερωτημάτων σε χώρο-λεκτικά δεδομένα
Επιβλέποντας: Νικόλαος Μαμουλής

Ο στόχος της συγκεκριμένης διατριβής είναι η ανάπτυξη αλγορίθμων για την επίλυση χωρο-λεκτικών ερωτημάτων (spatio-textual queries) πάνω σε δεδομένα μεγάλου όγκου (big-data) και κατανεμημένο περιβάλλον. Το συγκεκριμένο αντικείμενο έχει μελετηθεί εκτενώς από την ακαδημαϊκή κοινότητα και έχουν προταθεί πολλοί και αποδοτικοί αλγόριθμοι για την αντιμετώπιση του ζητήματος.

Οι υπάρχουσες μελέτες (και κατά συνέπεια και υλοποιήσεις) επικεντρώνονται στην λειτουργία σε κεντρικό υπολογιστικό περιβάλλον, ήτοι την επεξεργασία των δεδομένων σε ένα και μόνο τερματικό μηχάνημα (Η/Υ, κινητό, κτλ). Το γεγονός αυτό επιφέρει ένα σημαντικό μειονέκτημα: Θα πρέπει τα δεδομένα να είναι αρκούντως μικρά ώστε να μπορούν να επεξεργαστούν από τις πολλές φορές περιορισμένες υπολογιστικές δυνατότητες του τερματικού μηχανήματος.

Με το πέρασμα του χρόνου και την ευρύτατη εξάπλωση του διαδικτύου και των κινητών συσκευών που έχουν πρόσβαση σε αυτό, ο όγκος των παραγόμενων δεδομένων (όλων των τύπων) έχει πολλαπλασιαστεί. Μιλάμε πλέον για big-data ήτοι δεδομένα της τάξης των

πολλών gigabytes. Τα δεδομένα αυτά στην πλειοψηφία των περιπτώσεων αφορούν και την τοποθεσία των χρηστών, άρα μιλάμε για χωρικά δεδομένα (location based data).

Στην διατριβή αυτή θα επιλύσουμε το ζήτημα της επεξεργασίας αυτών των δεδομένων σε κατανεμημένο περιβάλλον και θα προτείνουμε αλγορίθμους για το περιβάλλον αυτό. Λέγοντας κατανεμημένο περιβάλλον εννοούμε κάποιο υπολογιστικό σύστημα με 2 η παραπάνω κόμβους. Ενώ όπως είπαμε το αντικείμενο των spatio-textual queries έχει μελετηθεί σε κεντρικό επίπεδο εκτενώς, η μεταφορά του σε κατανεμημένο περιβάλλον παρουσιάζει προκλήσεις.

Στην παρούσα εργασία θα παρουσιάσουμε τις προκλήσεις αυτές και θα δούμε πως αντιμετωπίζονται καθώς και θα προτείνουμε έναν παράλληλο αλγόριθμο που επιλύει με τον βέλτιστο τρόπο spatio-textual queries σε κατανεμημένο περιβάλλον.

Στους επιμέρους στόχους της εργασίας ανήκει και ο σχεδιασμός ενός προσομοιωτή ο οποίος θα προσομοιώνει την λειτουργία του παράλληλου αλγορίθμου σε κεντρικό περιβάλλον.

ABSTRACT

Surname: Bestas
Name: Dimitrios
Initials: B.D
Title: MSc
Computer Science Department, University of Ioannina, Greece
13/2/2017
Parallel and distributed processing of spatio-textual queries
Thesis Supervisor: Nikolaos Mamoulis

In this thesis we aim to implement algorithms for solving spatio-textual queries over big data and thus over a distributed environment.

This topic has been extensively studied by the academic community, while a plethora of algorithms have been proposed and implemented that handle these types of queries.

The problem with the existing solutions is that they focus on a centralized environment, or processing of the available data on a single terminal device (PC, mobile phone, etc). This fact incurs a significant disadvantage: The data to be processed should be sufficiently small in order to be able to be processed by the sometimes limited processing capabilities of the terminal device.

With the advent of time and the huge growth of the internet and the widespread availability of devices that can access it, the amount of data produced has increased exponentially. We now talk about big-data, or data in the magnitude of hundreds of gigabytes. Depending on the

application that generates the data (eg Facebook, Flickr, twitter, foursquare, etc.) more often than not, the geographical location of the user is also included. Thus we are faced with location based data.

In this thesis we will tackle the problems that arise when trying to process such data on a distributed environment and we will propose algorithms that run on such environments. A distributed environment is a computational system that is comprised of at least two computing nodes.

As mentioned before, spatio-textual queries have been studied in a centralized environment, but transferring the existing knowledge to a distributed environment poses challenges. We will study those challenges and propose parallel algorithms that solve spatio-textual queries in an optimum way.

Motivated by this trend, in this thesis, we study the novel problem of parallel and distributed processing of *spatial preference queries using keywords*, 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.

We propose a solution that has two notable features:

- we propose a deliberate re-partitioning mechanism of input data to servers, which allows parallelized processing, thus establishing the foundations for a scalable query processing algorithm, and

- we 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, we implement parallel algorithms that solve the problem in the MapReduce framework. Our experimental study using both real and synthetic data in a cluster of sixteen physical machines demonstrates the efficiency of our solution.

CHAPTER 1. INTRODUCTION

With the advent of modern applications that record the position of mobile users by means of GPS, and the extensive use of mobile smartphones, we have entered the era of Big Spatial Data. The fact that an increasing amount of user-generated content (e.g., messages in Twitter, photos in Flickr, etc.) is geotagged also contributes to the daily creation of huge volumes of location-based data. Apart from spatial locations, the data typically contain textual descriptions or annotations.

Analyzing and exploiting such textually annotated location-based data is estimated to bring high economic benefits in the near future. In order to extract useful insights from this wealth of Big Spatial Data, advanced querying mechanisms are required that retrieve interesting results from massively distributed spatio-textual data.

Advanced queries that combine spatial constraints with textual relevance to retrieve objects of interest have attracted increased attention recently due to the ever-increasing rate of user-generated spatio-textual data.

Such queries are processing-intensive, especially for large datasets, since any object may belong to the result set if it complies with the query restraints.

In this thesis, we study such a query that retrieves data objects based on the textual relevance of other (feature) objects in their spatial neighborhood.

In particular, given a keyword-based query, 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 is generic, as it can be used to retrieve locations of interest based on the relevance of Tweets in their vicinity, based on popular places (bars, restaurants, etc.), and/or based on the comments of other people in the surrounding area.

However, processing this query raises significant challenges. First, due to the query definition, every data object is a potential result and cannot be pruned by spatial or textual constrains. Second, the size of the data ranges from a few gigabytes to hundreds of gigabytes (or more). This means that processing the data on a central level, i.e. on a single machine, is impossible taking into account the limited hardware capabilities such a machine could possess. This in turn leads to the need of parallelizing the problem. The initial dataset must be split into pieces and each piece must be processed independently by a distributed system, while ensuring that the final result is not affected by this procedure.

Similar queries have been extensively studied by academia, while efficient algorithms have been proposed that solve that queries [1],[2],[71] etc. However those works focus exclusively on a centralized processing environment and thus cannot be classified under the "big-data" category.

In this thesis, we address the technical challenges described above and provide the first solution to parallel/distributed processing of the spatial preference query using keywords.

Our approach has two notable features:

- We propose a method to parallelize processing by deliberately re-partitioning input data, in such a way that the partitions can be processed in parallel, independently from each other, and
- Within each partition, we apply an early termination mechanism that eagerly restricts the number of objects that need to be processed in order to provide the correct result set.

In more detail, we make the following contributions in this thesis:

- We formulate and address a novel problem, namely parallel/distributed evaluation of spatial preference queries using keywords over massive and distributed spatio-textual data.
- We propose a grid-based partitioning that uses careful duplication of feature objects in selected neighboring cells that allows independent processing of subsets of input data in parallel, thus establishing the foundations We further boost the performance of our algorithm by introducing an early termination mechanism for each independent work unit, thereby reducing the processing cost.
- We demonstrate the efficiency of our algorithms by means of experimental evaluation using both real and synthetic datasets in a medium-sized cluster.
- We examine further possible improvements that can be implemented without significant overhead, that take into account the distribution of the objects in the dataset. Those improvements incur quantified performance gains that are independent of the proposed algorithm, thus reusable to other similar works.

CHAPTER 2. RELATED WORK

2.1. Spatial preference queries
2.2. Top-k spatial preference queries
2.3. Set similarity joins
2.4. Spatio-textual search

In this section we will briefly present some existing work on the subject of spatial queries. Most of the existing work studies spatial queries with or without textual information based on a centric approach. That is that the solutions and algorithms proposed work on a single device (personal computer, mobile phone, etc).

2.1. Spatial preference queries

Spatial preference queries [71] rank objects based on the qualities of features in their spatial neighborhood.

For example, think a real estate agency database of flats for lease; a customer may want to rank the flats with respect to the appropriateness of their location, defined after aggregating the qualities of other features (e.g., schools, cafes, hospital, market, etc.) within their spatial neighborhood.

Such a neighborhood concept can be specified by the user via different functions. It can be an explicit circular region within a given distance from the flat. Another intuitive definition is to consider the whole spatial domain and assign higher weights to the features based on their proximity to the flat.

Given a set *D* of objects of interest (e.g., candidate locations), a top-*k* spatial preference query retrieves the *k* objects in *D* with the highest scores. The score of an object is defined by the quality of features (e.g., facilities or services) in its spatial neighborhood.

2.2. Top-k spatial preference queries

Top-*k* spatial preference queries return a ranked set of the *k* best data objects based on the scores of feature objects in their spatial neighborhood. Despite the wide range of location-based applications that rely on spatial preference queries, existing algorithms incur non-negligible processing cost resulting in high response time. The reason is that computing the score of a data object requires examining its spatial neighborhood to find the feature object with highest score.

One technique to speed up the performance of top-*k* spatial preference queries is the mapping of pairs of data and feature objects to a distance-score space, which in turn allows identifying and materializing the minimal subset of pairs that is sufficient to answer any spatial preference query [67].

In [67] the authors also present a novel algorithm that improves query processing performance by avoiding examining the spatial neighborhood of the data objects during query execution. In addition, an efficient algorithm for materialization is proposed and the useful properties that reduce the cost of maintenance are described.

2.3. Set similarity joins

Recently, the set-similarity join has attracted significant interest. Given a collection *D* of set-valued data, the problem is to find pairs (x, y) of sets in *D*, such that $sim_t(x, y) \geq \theta$, where $sim_t(x, y)$ is a similarity function and θ is a threshold. The main application of set-similarity joins is near-duplicate object detection [38] (e.g., identify plagiarism, record linkage in data integration, duplicate data cleansing, etc.). Set-similarity joins can also be used to facilitate string matching; [40] showed that the edit distance between two strings can be bounded by set-similarity measures defined on two sets of q-grams, which approximate the strings.

Computing set-similarity joins based on inverted files [55] was first proposed in [49]: for each object *x*, the inverted lists that correspond to *x*'s elements are scanned to accumulate the similarity between *x* and all other objects. Several optimizations over this baseline approach are proposed, including scanning only a smaller subset of *x*'s lists and performing a single pass over the data that constructs the inverted index and computes the join result at the same time.

Chaudhuri et al. [10] suggested an efficient filter-refinement framework for set-similarity joins, based on the observation that for two sets *x*, *y* to satisfy $sim(x, y) \geq t$, a necessary condition is that prefixes of *x* and *y* should have at least some minimum overlap.

Arasu et al. [26] showed that this prefix-based filtering is just one of the possible summary schemes that one could use as necessary conditions and provided alternative schemes with theoretical bounds on their effectiveness.

Bayardo et al., [28] proposed an efficient framework for evaluating set-similarity joins, which minimizes the necessary elements to add in the inverted file, during join evaluation, based on pre-computed bounds on the element weights in the sets and appropriate orderings for the domain of set elements and the database *D*.

This method is further optimized by Xiao et al. [52], by enhancing prefix-filtering using positional information of elements in the prefixes and partially-seen suffixes of the joined objects.

2.4. Spatio-textual search

In the past decade, there has been increasing interest on extracting spatial information from web pages such as addresses, phone numbers, zip codes, and then assigning geographic tags to the pages, a process known as geo-tagging [1, 13, 21].

Documents are given a geographic footprint, i.e., a set of locations; the footprint is often approximated by an MBR. Geo-tagging facilitates multi-criteria search, such as searching documents by textual content and spatial location; this type of search has already been considered by commercial search engines like Google Maps. SPIRIT [50] is a search engine that supports spatio-textual selection queries; the user inputs a set of keywords and a set of spatial predicates and the engine returns the documents, which contain the keywords and their spatial footprint satisfies the spatial predicates (e.g., "find all documents about children hospitals within 10km from the city center").

Several indexing approaches for the efficient support of spatio-textual selections have been proposed [35, 42, 50, 54]. Some of these methods propose extensions of the R-tree, which associate nodes or entries of the tree with inverted files for the contents of the corresponding subtrees [42]; other approaches primarily index the data using an inverted file and then spatially index each inverted list by an R-tree [51] or a space-filling curve [35].

De Felipe et al. [39] extend the R-tree to support containment nearest neighbor queries. Given a query location q, a set of keywords *K*, and an integer *k*, the objective is to find the *k* nearest objects to *q* which include all keywords in *K*. Each entry *e* of the tree, apart from its MBR, stores a bitmap, which encodes the set of keywords included in every document in the subtree indexed by *e*.

The algorithm of [43] is used to retrieve the nearest neighbors of *q* incrementally; entries that violate the keyword containment constraint of the query are pruned during search. Cong et al. [36] and Li et al. [44] independently proposed an IR-tree index, which primarily indexes the data using an R-tree, but creates an inverted file for each node of the tree. The inverted file of a leaf node indexes all documents in the node, while in the inverted file of a non-leaf node, each id corresponds to a child (i.e., subtree) of the non-leaf node. The inverted list for a term contains the children which include that term and the maximum weight of the term in any object of the corresponding subtree.

By extending the nearest neighbor algorithm of [43], the IR-tree can be used to answer spatio-textual proximity queries, where the user provides a location q and a set of keywords *K* and asks for the best object on a map with respect to both distance from *q* and similarity with *K*.

An alternative, Spatial Inverted Index for spatio-textual proximity queries was recently proposed by [48]. This method generates one inverted list per term and indexes each long inverted list using an aggregate R-tree [47]; given a query, the lists of the query terms are joined, by accessing from each tree the objects in increasing order of relevance and merging them, until the *k* best objects are guaranteed to be found.

Several, more complex queries have also been defined and studied in the context of spatio-textual search, like prestige-based spatio-textual similarity [31] and finding spatially close groups of objects that match the query keywords [32, 53].

CHAPTER 3. PARALLEL AND DISTRIBUTED PROCESSING OF SPATIAL PREFERENCE QUERIES USING KEYWORDS

3.1. Preliminaries
3.2. Problem statement and overview of solution
3.3. Grid-based partitioning and initial algorithm
3.4. Algorithms with early termination
3.5. Theoretical results
3.6. Experimental evaluation
3.7. Performance improvement techniques
3.8. Emulator

3.1. Preliminaries

In this section we give a brief overview of MapReduce and HDFS, and define the type of queries we will focus on.

3.1.1 MapReduce and HDFS

Hadoop is an open-source implementation of MapReduce [59], providing an environment for large-scale, fault-tolerant data processing. Hadoop consists of two main parts: the HDFS distributed file system and MapReduce for distributed processing.

Files in HDFS are split into a number of large blocks which are stored on DataNodes, and one file is typically distributed over a number of DataNodes in order to facilitate high bandwidth during parallel processing. In addition, blocks can be replicated to multiple DataNodes (by default three replicas), in order to ensure fault-tolerance. A separate NameNode is responsible for keeping track of the location of files, blocks, and replicas thereof. HDFS is designed for use-cases where large datasets are loaded (“write-once”) and processed by many different queries that perform various data analysis tasks (“read-many”).

A task to be performed using the MapReduce framework has to be specified as two steps. The *Map* step as specified by a map function takes some input (typically from HDFS files), possibly performs some computation on this input, and redistributes it to worker nodes (a process known as “shuffle”). An important aspect of MapReduce is that both the input and output of the Map step is represented as key-value pairs, and that pairs with same key will be processed as one group by a Reducer. As such, the *Reduce* step receives all values associated with a given key, from multiple map functions, as a result of the redistribution process, and typically performs some aggregation on the values, as specified by a reduce function.

It is important to note that one can customize the redistribution of data to Reducers by implementing a *Partitioner* that operates on the output key of the map function, thus practically enforcing an application-specific grouping of data in the Reduce phase. Also, the ordering of values in the reduce function can be specified, by implementing a customized Comparator. In our work, we employ such customizations to obtain a scalable and efficient solution to our problem.

3.1.2 Spatial Preference Queries

The spatial preference query belongs to a class of queries that rank objects based on the quality of other (feature) objects in their spatial neighborhood [67, 70, 71]. Inherently a spatial preference query assumes that two types of objects exist: the data objects, which will be ranked and returned by the query, and the feature objects, which are responsible for ranking the data objects. As such, the feature objects determine the score of each data object according to a user-specified metric. Spatial preference queries find more applications in the case of textually annotated feature objects [68], where the score of data objects is determined by a textual similarity function applied on query keywords and textual annotations of feature objects. This query is known as top-*k* spatio-textual preference query [68]. In this thesis, we study a distributed variant of this query.

Table 1: Overview of symbols

Symbol	Description
O	Set of data Objects
p	Objects in O, $p \in O$
F	Set of feature Objects
f	Feature object in F, $f \in F$
$f.W$	Keywords associated with feature object f
$q(k, r, W)$	Query for top-k data objects
$w(f, q)$	Textual relevance of feature f to query q
$\overline{w}(f, q)$	Upper bound of $w(f, q)$
$dist(p,f)$	Spatial distance between p and f
$\tau(p)$	Score of data object p
$\overline{\tau}$	Score of the k-th best data object
R	Number of reduce tasks
$C = \{C_1, \ldots, C_R\}$	Grid cells

3.2. Problem statement and overview of solution

3.2.1 Problem Formulation

Consider an *object* dataset O of spatial objects $p \in O$, which are described by their coordinates $p.x$ and $p.y$. Also, consider a *feature* dataset $F\langle i \rangle$ of spatio-textual objects $f \in F$, which are represented by spatial coordinates $f.x$ and $f.y$, and a set of keywords $f.W$.

The spatial preference query using keywords returns the k data objects $\{p_1, \ldots, p_k\}$ from O with the highest score. The score of a data object $p \in O$ is defined by the scores of feature objects $f \in F$ in its spatial neighborhood. As already mentioned, each feature object f is associated with a set of keywords $f.W$.

A query q consists of a neighborhood distance threshold r, a set of query keywords $q.W$ for the feature set F, and the value k that determines how many data objects need to be retrieved. For a quick overview of the basic symbols used in this thesis, we refer to Table 1.

Given a query $q(k, r, W)$ and a feature object $f \in F$, we define the *non- spatial score* $w(f, q)$ that indicates the goodness (quality) of f as the similarity of sets $q.W$ and $t.W$. In this work, we employ Jaccard similarity for this purpose. Obviously, the domain of values of $w(f, q)$ is the range [0, 1].

Definition 1: (Non-spatial score $w(f, q)$)*: Given a query* q *and a feature object $f \in F$, the non-spatial score $w(f, q)$ determines the textual relevance between the set of query keywords $q.W$ and the keywords $f.W$ of f using Jaccard similarity:*

$$w(f, q) = \frac{|q.W \cap f.W|}{|q.W \cup f.W|}$$

The score $\tau(p)$ of a data object p is determined by the feature objects that are within distance r from p. More specifically, $\tau(p)$ is defined by the maximum non-spatial score $w(f, q)$ of any feature object f in the r-neighborhood of p. This range based neighborhood condition is typically used in the related work [67, 68, 70, 71].

Formally:

Definition 2: *The score $\tau(p)$ of p based on feature dataset F, given the range-based neighborhood condition r is defined as:*

$$\tau(p) = max\{w(f, q) \mid f \in F : d(p, f) \leq r\}$$

Example 1: Figure 1 depicts an example: The spatial area contains both data objects (denoted as p_i) and feature objects (denoted as f_i). The data objects represent hotels and the feature objects represent restaurants.

Assume a user issues the following query: Find the best (top-k) hotels that have an Italian restaurant nearby. Let us assume that $k = 1$ and "nearby" is translated to $r = 1.5$ units of distance. Then, the query is expressed as: Find the top-1 data object for which there exists a highly ranked feature object based on the keyword "Italian" at a distance at most 1.5 units.

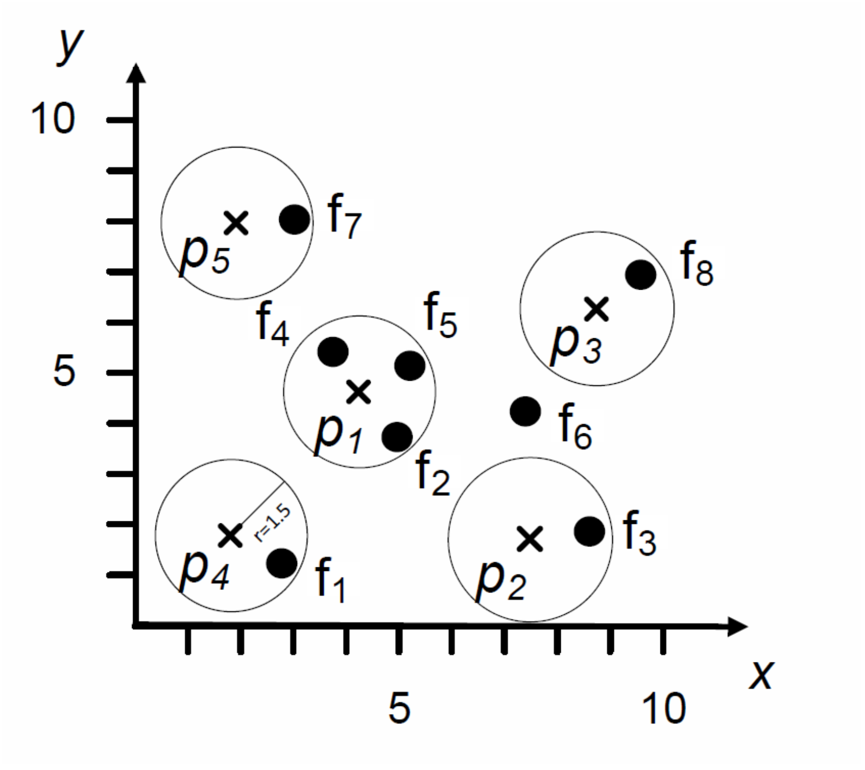

Figure 1: Example of spatial preference query using keywords (*SPQ*).

Table 2 lists the locations and descriptions of both data and feature objects. Only feature objects f_1, f_4 and f_7 have a common term with the user specified query (the keyword “italian”). Thus, only f_1, f_4 and f_7 will have a Jaccard score other than 0.

In the last column of Table 2 the Jaccard score for all feature objects is shown. The score of each data object is influenced only by the feature objects within a distance at most 1.5 units.

In Figure 1 the circles with radius 1.5 range units and center each data object include the feature objects that are nearby each data object and influence its score. The actual score of a data object is the highest score of all nearby feature objects.

Data object p_4 has a score of 0.5 due to feature object f_1, data object p_1 has a score of 1 because of feature object f_4 and data object p_5 has a score of 0.5 due to feature object f_7. Hence, the top-1 result is object p_1.

Table 2: Example of datasets and scores for query $q.W$ = {italian}

Object	X	Y	Keywords	Jaccard
p1	4.6	4.8	-	-
p2	7.5	1.7	-	-
p3	8.9	5.2	-	-
p4	1.8	1.8	-	-
p5	1.9	9.0	-	-
f1	2.8	1.2	italian,gourmet	0.5
f2	5.0	3.8	chinese,cheap	0
f3	8.7	1.9	sushi,wine	0
f4	3.8	5.5	italian	1
f5	5.2	5.1	mexican,exotic	0
f6	7.4	5.4	greek,traditional	notInRange
f7	3.0	8.1	italian,spaghetti	0.5
f8	9.5	7.0	indian	0

In the parallel and distributed setting that is targeted in this thesis, datasets O and F are horizontally partitioned and distributed to different machines (servers), which means that each server stores only a fraction (partition) of the entire datasets.

In other words, there exists a number of partitions $O_i \in O$ and $F_i \in F$ of datasets O and F respectively, such that $\bigcup O_i = O$, $O_i \cap O_j = \emptyset$ for $i \neq j$, and $\bigcup F_i = F$, $F_i \cap F_j = \emptyset$ for $i \neq j$. Due to horizontal partitioning, any (data or feature) object belongs to a single partition (*Oi* or *Fi* respectively). We make no assumption on the number of such partitions nor on having equal number of data and feature object partitions. Also, no assumptions are made on the specific partitioning method used; in fact, our proposed solution is independent of the actual partitioning method employed, which makes it applicable in the most generic case.

Problem 1. (Parallel/Distributed Spatial Preference Query using Keywords (*SPQ*)) *Given an object dataset* O *and a feature dataset* F*, which are horizontally partitioned and distributed to a set of servers, the parallel/distributed spatial preference query using keywords returns the k data objects* $\{p_1, \ldots, p_k\}$ *from O with the highest* $\tau(p_i)$ *scores.*

3.2.2 Design Rationale

The spatial preference query using keywords (*SPQ*) targeted in this thesis is a complex query operator, since any data object p may belong to the result set and the spatial range cannot be used for pruning the data space.

As a result, the computation becomes more challenging and efficient query processing mechanisms are required that can exploit parallelism and the availability of hardware resources. Parallelizing this query is also challenging because any given data object p and all feature objects within the query range r from p must be assigned to the same server to ensure the correct computation of the score $\tau(p)$ of p.

As such, a repartitioning mechanism is required in order to assign (data and feature) objects to servers in a deliberate way that allows local processing at each server. To achieve the desired independent score computation at each server, duplication of feature objects to multiple servers is typically necessary. Based on this, we set the following objectives for achieving parallel, scalable and efficient query processing:

- Objective #1: parallelize processing by breaking the work into independent parts, while minimizing feature object duplication. In addition, the union of the results in each part should suffice to produce the final result set.
- Objective #2: avoid processing the input data in its entirety, by providing early termination mechanisms for query processing.

To meet the above objectives, we design our solution by using the following two techniques:

- a grid-partitioning of the spatial data space that uses careful duplication of feature objects in selected neighboring cells, in order to create independent work units (Section 3.3), and
- sorted access to the feature objects in a deliberate order along with a thresholding mechanism that allows early termination of query processing that guarantees the correctness of the result (Section 3.4).

3.3. Grid-based partitioning and initial algorithm

In this section, we present an algorithm for solving the parallel/distributed spatial preference query using keywords, 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. To ensure correctness of the result computed in parallel, we repartition the input data to grid cells and deliberately duplicate some feature objects in neighboring grid cells. As a result, this technique lays the foundation for parallelizing the query processing and leads to the first scalable solution.

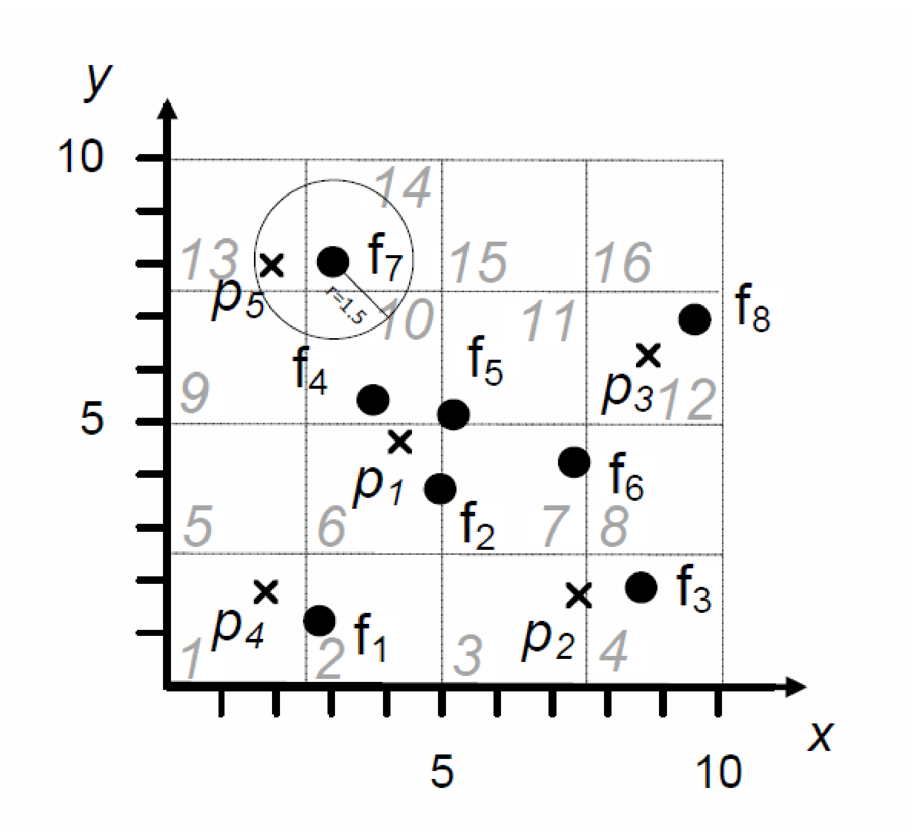

Figure 2: Example of grid partitioning

3.3.1 Grid-based Partitioning

Consider a regular, uniform grid in the 2-dimensional dataspace that consists of R cells: $C = \{C_1, . . . ,C_R\}$. Our approach assigns all data and feature objects to cells of this grid, and expects each cell to be processed independently of the other cells. In MapReduce terms, we assign each cell to a single processing task (Reducer), thus all data that affect this cell need to be sent to the assigned Reducer. The repartitioning mechanism operates in the following way: Based on the spatial location of an object (data or feature object), this object is assigned to the cell that encloses it in a straightforward way. However, some feature objects must be additionally assigned to other cells (i.e., duplicated), in order to ensure that the data in each cell can indeed be processed independently of the other cells and produce the correct result. More specifically, given a feature object $f \in C_j$ and any grid cell C_i ($C_i \neq C_j$), we denote by MINDIST(f, C_i) the minimum distance between feature object f and C_i. This distance is defined as the distance of f to the nearest edge of Ci, since f is outside C_i. When this minimum distance is smaller than the query radius r, i.e., MINDIST(f, C_i) $\leq$ r, then it is possible that f is within distance r from a data object $p \in C_i$. Therefore, f needs to be assigned (duplicated) also to cell C_i. The following lemma guarantees the correctness of the afore-described technique.

Lemma 1. *(Correctness) 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) $\leq r$.*

Figure 2 illustrates the same dataset as in Figure 1 and a 4x4 grid (the numbering of the cells is shown in the figure). Consider a query with radius r = 1.5, and let us examine feature object f_7 as an example. Assuming that f_7 has at least one common term in its keyword set ($f_7.W$) with the user specified query ($q.W$), then f_7 may affect neighboring cells located near cell with identifier C_{14}. It is fairly easy to see that f_7 needs to be duplicated to cells C_9, C_{10}, and C_{13}, for which MINDIST(f_7,C_i) $\leq$ r, thus the score of data objects located in these cells may be determined by f_7.

Before presenting the algorithm that exploits this grid partitioning, we make a note on how to select an appropriate grid size, as this affects the amount of duplication required. It should also be noted that in our approach the grid is defined at query time, after the value of r is known. Let α denote the length of the edge of a grid cell. For now, we should ensure that $\alpha \geq r$, otherwise excessive replication to neighboring cells would be performed. Later, in Section 6, we provide a thorough analysis on the effect of the grid cell size to the amount of duplicated data. Moreover we will show experimental results on this relationship in the results section.

3.3.2 Parallel Algorithm

We design a parallel algorithm, termed *pSPQ*, that solves the problem in MapReduce. The Map phase is responsible for re-partitioning the input data based on the grid introduced earlier. Then, in the Reduce phase, the problem of reporting the top-*k* data objects is solved in each cell independently of the rest. This is the part of the query that dominates the processing time; the final result is produced by merging the *k* results of each of the *R* cells and returning the top-*k* with the highest score. However, this last step can be performed in a centralized way without significant overhead, given that the number of these results is small because *k* is typically small.

In more detail, in the Map phase, each Map task (Mapper) receives as input some data objects and some feature objects, without any assumptions on their location. Each Mapper is responsible for assigning data and feature objects to grid cells, including duplicating feature objects. Each grid cell corresponds to a single Reduce task, which will take as input all objects assigned to the respective grid cell. Then, the Reducer can accurately compute the score of any data object located in the particular grid cell and report the top-*k*.

```
1:  **Input**: $q(k, r, W)$, grid cells $C = \{C_1, \dots, C_R\}$
2:  **function** $MAP(x: input\ object)$
3:  $C_i \leftarrow \{C_i : C_i \in C$ and $x$ enclosed in $C_i\}$
4:  **if** $x$ is a data object **then**
5:    $x.tag \leftarrow 0$
6:    output $\langle(i, x.tag), x\rangle$
7:  **else**
8:    $x.tag \leftarrow 1$
9:    **if** $(x.W \cap q.W = \emptyset)$ **then**
10:     output $\langle(i, x.tag), x\rangle$
11:     **for** $(C_j \in C$, such that $MINDIST(x, C_j) \leq r$ **do**
12:       output $\langle(j, x.tag), x)\rangle$
13:     **end for**
14:   **end if**
15: **end if**
16: **end function**
```

Figure 3: Algorithm 1: *pSPQ*: Map function

3.3.2.1 Map Phase

Algorithm 1 shows the pseudo-code of the Map phase, where each call of the Map function processes a single object denoted by *x*, which can be a data object or a feature object.

First, in line 3, the cell C_i that encloses object *x* is determined.

Then, if *x* is a data object, it is tagged (*x.tag*) with the value 0, otherwise with the value 1. In case of a data object, *x* is output using a *composite key* that consists of the cell id *i* and the tag as key, and as value the entire data object *x*. In case of a feature object, we apply a simple pruning rule (line 9) to eliminate feature objects that do not affect the result of the query. This rule practically eliminates from further processing any feature object that has no common keyword with the query keywords, i.e., $q.W \cap f.W = \emptyset$. The reason is that such feature objects cannot contribute to the score of any data object, based on the definition of our query. This pruning rule can significantly limit the number of feature objects that need to be i) duplicated and ii) be sent to the Reduce phase. For the remaining feature objects that have at least one common keyword with the query, they are first output with the same composite key as above, and value the entire feature object *x*.

In addition, we identify neighboring cells C_j that comply with Lemma 1, and replicate the feature object in those cells too. In this way, we have partitioned the initial data to grid cells and have performed the necessary duplication of feature objects.

The output key-values of the Map phase are grouped by cell id and assigned to Reduce tasks using a customized Partitioner. Also, in each Reduce task, we order the objects within each group by their tag, so that data objects precede feature objects. This is achieved through the use of the composite keys for sorting. As a result, it is guaranteed that each Reducer accesses any feature object after it has accessed all data objects.

3.3.2.2 Reduce Phase

As already mentioned, a Reduce task processes all the data assigned to a single cell and reports the top-k data objects within the respective cell. The pseudo-code of the Reduce function is depicted in Algorithm 2.

First, all data objects are accessed one-by-one and loaded in memory (O_i). Moreover, a sorted list L_k of the k data objects pi with higher scores τ (p_i) is maintained. Let $\overline{\tau}$ denote the k-th best score of any data object so far.

Then, for each feature object x accessed, its non-spatial score $w(x, q)$ (i.e., textual similarity to the query terms) is compared to $\overline{\tau}$. Only if the non-spatial score $w(x, q)$ is larger than $\overline{\tau}$ (line 9), may the top-k list of data objects be updated. Therefore, in this case we test all combinations of x with the data objects p kept in memory O_i. If such a combination (x, p) is within distance r (line 11), then we check if the temporary score of p denoted by $score(p)$ can be improved based on x (i.e., $w(x, q)$), and if that is the case we check whether p has obtained a score that places it in the current top-k list of data objects (L_k).

Line 12 shows how the score can be improved, however we omit from the pseudo-code the check of score improvement of p for sake of simplicity. Then, in line 13, the list L_k is updated. As explained, this update is needed only if the score of p is improved. In this case, if p already exists in L_k we only update its score; otherwise p is inserted into L_k. After all feature objects have been processed, L_k contains the top-k data objects of this cell.

3.3.2.3 Limitations

The above algorithm provides a correct solution to the problem in a parallel manner, thus achieving Objective #1. However, in each Reducer, it needs to process the entire set of feature objects in order to produce the correct result. In the following section, we present techniques that overcome this limitation, thereby achieving significant performance gains.

```
1: **Input**: $q(k,r,W)$
2: **function** $REDUCE(key, V$: objects assigned to cell with id $key)$
3: $L_k \leftarrow \emptyset$
4: **for** $(x \in V)$ **do**
5:   **if** $x$ is a data object **then**
6:   Load $x$ in memory $O_i$
7:   $score(x) \leftarrow 0$ // $initial\ score$
8:   **else**
9:     **if** $w(x,q) > \overline{\tau}$ **then**
10:       **for** $(p \in O_i)$ **do**
11:         **if** $d(p,x) \leq r$ **then**
12:           $score(p) \leftarrow \max\{score(p), w(x,q)\}$
13:           update list $L_k$ of top-$k$ data objects and $\overline{\tau}$
14:         **end if**
15:       **end for**
16:     **end if**
17:   **end if**
18: **end for**
19: **for** $p \in L_k$ **do**
20:   output $\langle p, score(p)\rangle$ // $at\ this\ point$: $score(p) = \tau(p)$
21: **end for**
22: **end function**
```

Figure 4: Algorithm 2: *pSPQ*: Reduce function

3.4. Algorithms with early termination

Even though the technique outlined in the previous section enables parallel processing of independent partitions to solve the problem, it cannot guarantee good performance since it requires processing both data partitions in a cell in their entirety (including duplicated feature objects). To alleviate this shortcoming, we introduce two alternative techniques that achieve *early termination*, i.e., report the correct result after accessing all data objects but only few feature objects. This is achieved by imposing a deliberate order for accessing feature objects in each cell, which in turn allows determining an upper bound for the score of any unseen feature object. When this upper bound cannot improve the score of the current top-*k* object, we can safely terminate processing of a given Reducer.

3.4.1 Accessing Feature Objects by Increasing Keyword Length

The first algorithm that employs early termination, termed *eSPQlen*, is based on the intuition that feature objects f with long textual descriptions that consist of many keywords ($|f.W|$) are expected to produce low scores $w(f, q)$. This is due to the Jaccard similarity used in the definition of $w(f, q)$ (Defn. 1), which has $|q.W \cup f.W|$ in the denominator. Based on this, we impose an ordering of feature objects in each Reducer by increasing keyword length, aiming at examining first feature objects that will produce high score values $w(f, q)$ with higher probability.

In more technical details, given the keywords $q.W$ of a query q, and a feature object f with keywords $f.W$, we define a bound for the best possible Jaccard score that this feature object can achieve as:

$$\overline{w}(f,q) = \begin{cases} 1 & ,|f.W| < |q.W| \\ \frac{|q.W|}{|f.W|} & ,|f.W| \geq |q.W| \end{cases} \quad (1)$$

Given that feature objects are accessed by increased keyword length, this bound is derived as follows. As long as feature objects f are accessed that satisfy $|f.W| < |q.W|$, it is not possible to terminate processing, thus the bound takes the value of 1. The reason is that $|f.W| < |q.W|$ | holds, it is possible that a subsequent feature object f' with more keywords than f may have higher Jaccard score than f. However, as soon as it holds that $|f.W| \geq |q.W|$ the bound (best possible score) equals:

$$\frac{\min\{|q.W|,|f.W|\}}{\min\{|q.W|,|f.W|\} + |f.W| - |q.W\}} = \frac{|q.W|}{|f.W|}$$

because in the best case the intersection of sets $q.W$ and $f.W$ will be equal to: $\min\{|q.W|, |f.W|\}$, while their union will be equal to: $\min\{|q.W|, |f.W|\} + |f.W| - |q.W|$. Recall that $\overline{\tau}$ denotes the k-th best score of any data object so far. Then, the condition for early termination during processing of feature objects by increasing keyword length, can be stated as follows:

Lemma 2. *(Correctness of Early Termination* eSPQlen*) Given a query q and an ordering of feature objects based on increasing number of keywords, it is safe to stop accessing more feature objects as soon as a feature object* f *is accessed with:*

$$\bar{\tau} \geq w(f,q)$$

Based on this analysis, we introduce a new algorithm that follows the paradigm of Section 4, but imposes the desired access order to feature objects and is able to terminate early in the Reduce phase.

3.4.1.1 Map Phase

Algorithm 3 describes the Map phase of the new algorithm. The main difference to the algorithm described in Section 4 is in the use of the composite key when objects are output by the Map function (lines 8 and 10).

The composite key contains two parts. The first part is the cell id, as previously, but the second part is a number. The second part corresponds to the value zero in the case of data objects, while it corresponds to the length $|f.W|$ of the keyword description in the case of a feature object *f*.

The rationale behind the use of this composite key is that the cell id is going to be used to group objects to Reducers, while the second part of the key is going to be used to establish the ordering in the Reduce phase in increased order of the number used. In this sorted order, data objects again precede feature objects, due to the use of the zero value.

Between two feature objects, the one with the smallest length of keyword description (i.e., fewer keywords) precedes the other in the sorted order.

```
 1: Input: $q(k, r, W)$, grid cells $C = \{C_1, \dots, C_R\}$
 2: function $MAP(x{:}\ input\ object)$
 3: $C_i \leftarrow \{C_i : C_i \in C$ and $x$ enclosed in $C_i\}$
 4: if $x$ is a data object then
 5:   output $\langle(i, 0), x\rangle$
 6: else
 7:   if $(x.W \cap q.W = \emptyset)$ then
 8:     output $\langle(i, |x.W|), x\rangle$
 9:     for $(C_j \in C$, such that $MINDIST(x, C_j) \le r$ do
10:       output $\langle(j, |x.W|), x\rangle$
11:     end for
12:   end if
13: end if
14: end function
```

Figure 5: Algorithm 3: *eSPQlen*: Map function

3.4.1.2 Reduce Phase

As already mentioned, feature objects with long keyword lists are expected to result in decreased textual similarity (in terms of Jaccard value). Thus, our hope is that after accessing feature objects with few keywords, we will find a feature object that has so many keywords that all remaining feature objects in the ordering cannot surpass the score of *k*-th best data object thus far.

Algorithm 4 explains the details of our approach. Again, only the set of data objects assigned to this Reducer is maintained in memory, along with a sorted list L_k of the *k* data objects with best scores found thus far in the algorithm.

The condition for early termination is based on the score $\overline{\tau}$ of the *k*-th object in list L_k and the best potential score *w*(*f*, *q*) of the current feature object *f* (line 9).

```
1: **Input**: $q(k, r, W)$
2: **function** $REDUCE(key, V$: objects assigned to cell with id $key)$
3: $L_k \leftarrow \emptyset$
4: **for** $(x \in V)$ **do**
5:   **if** $x$ is a data object **then**
6:   Load $x$ in memory $O_i$
7:   $score(x) \leftarrow 0$ // $initial\ score$
8:   **else**
9:     **if** $\overline{\tau} \geq \overline{w}(x, q)$ **then**
10:       break
11:     **end if**
12:     **if** $w(x, q) > \overline{\tau}$ **then**
13:       **for** $(p \in O_i)$ **do**
14:         **if** $d(p, x) \leq r$ **then**
15:           $score(p) \leftarrow \max\{score(p), w(x, q)\}$
16:           update list $L_k$ of top-$k$ data objects and $\overline{\tau}$
17:         **end if**
18:       **end for**
19:     **end if**
20:   **end if**
21: **end for**
22: **for** $p \in L_k$ **do**
23:   output $\langle p, score(p)\rangle$ // $at\ this\ point$: $score(p) = \tau(p)$
24: **end for**
25: **end function**
```

Figure 6: Algorithm 4: *eSPQlen*: Reduce function with Early Termination

3.4.2 Accessing Feature Objects by Decreasing Score

In this section, we introduce an even better early termination algorithm, termed *eSPQsco*.

The rationale of this algorithm is to compute the Jaccard score in the Map phase and use this score as second part of the composite key.

In essence, this can enforce a sorted order in the Reduce phase where feature objects are accessed from the highest scoring feature object to the lowest scoring one.

To explain the intuition of the algorithm, consider the feature object with highest score. By drawing a circle in the dataset space with center that feature object and range *r* and checking witch data objects reside in that circle we get the top-*k* results. This observation leads to a more efficient algorithm that can (in principle) produce results when accessing even a single feature object.

As a result, the algorithm is expected to terminate earlier, by accessing only a handful of feature objects. This approach incurs additional processing cost at the Map phase (i.e., computation of the Jaccard score), but the imposed overhead to the overall execution time is minimal.

Lemma 3. (Correctness of Early Termination *eSPQsco*) *Given a query q and an ordering of feature objects* {*fi*} *based on decreasing score w*(*fi*, *q*)*, it is safe to stop accessing more feature objects as soon as k data objects are retrieved within distance r from any already accessed feature object.*

3.4.2.1 Map Phase

Algorithm 5 describes the Map function, where the only modifications are related to the second part of the composite key (lines 5, 8, and 10).

In the case of data objects, this must be set to a value strictly higher than any potential Jaccard value, i.e., it can be set equal to 2, since the Jaccard score is bounded in the range [0, 1]. Thus, data objects will be accessed before any feature object.

In the case of a feature object *f*, it is set to the Jaccard score *w*(*f*, *q*) of *f* with respect to the query *q*. Obviously, the customized Comparator must also be changed in order to enforce the ordering, from highest scores to lowest scores.

```
 1: **Input**: $q(k, r, W)$, grid cells $C = \{C_1, \dots, C_R\}$
 2: **function** $MAP(x: input\ object)$
 3: $C_i \leftarrow \{C_i : C_i \in C$ and $x$ enclosed in $C_i\}$
 4: **if** $x$ is a data object **then**
 5:   output $\langle (i, 2), x \rangle$
 6: **else**
 7:   **if** $(x.W \cap q.W = \emptyset)$ **then**
 8:     output $\langle (i, w(x, q)), x \rangle$
 9:     **for** $(C_j \in C$, such that $MINDIST(x, C_j) \le r$ **do**
10:       output $\langle (j, w(x, q)), x \rangle$
11:     **end for**
12:   **end if**
13: **end if**
14: **end function**
```

Figure 7: Algorithm 5: *eSPQsco*: Map function

3.4.2.2 Reduce Phase

Algorithm 6 details the operation of the Reduce phase. After all data objects are loaded in memory, the feature objects are accessed in decreasing order of their Jaccard score to the query. For each such feature object *f*, the algorithm draws a circle in the dataspace of the current reducer with center the feature object and range *r*. It then accesses all data objects and checks if they reside in that circle. If so (if dist ≤ *r*) that data object is part of the results.

If *k* data objects are found in the circle of the best feature object, the procedure stops. Since this feature object has the best possible score (in the scope of the current reducer), it is guaranteed that no other feature objects will be able to give a better score.

Here we have to note that if the best feature object does not have any data objects in its circle, it is discarded – it cannot be part of the results. In that case the next feature object with the second best score is accessed etc. until the *k* data objects are found.

In the highly unlikely case that no feature object has any data objects in its circle, then the algorithm will not produce a result for this specific reducer, and its performance will degrade to that of the *pSPQ* algorithm.

```
 1: **Input**: $q(k, r, W)$
 2: **function** $REDUCE(key, V$: objects assigned to cell with id $key)$
 3: **for** $(x \in V)$ **do**
 4:   **if** $x$ is a data object **then**
 5:     Load $x$ in memory $O_i$
 6:   **else**
 7:     **if** $\exists p \in O_i : d(p, x) \leq r$ **then**
 8:       output $\langle p, w(x, q)\rangle$ // $here$: $w(x, q) = \tau(p)$
 9:       $cnt++$
10:       **if** $(cnt = k)$ **then**
11:         break
12:       **end if**
13:     **end if**
14:   **end if**
15: **end for**
16: **end function**
```

Figure 8: Algorithm 6: *eSPQsco*: Reduce function

3.5. Theoretical results

In this section, we analyze the space and time complexity in the Reduce phase, which relate to the number of cells and the number of duplicate objects.

3.5.1 Complexity Analysis

Let *R* denote the number of reducers, which is also equivalent to the number of grid cells. Further, let O_i and F_i denote the subset of the data and feature object s assigned to the *i*-th reducer respectively.

Notice that F_i contains both the feature objects enclosed in the cell corresponding to the *i*-th reducer, as well as the duplicated feature objects that are located in other neighboring cells. In other words, it holds that:

$$\bigcup_{i=1}^{R} |Fi| \geq |F|$$

In the case of the initial parallel algorithm that does not use early termination (Section 4), a Reducer needs to store in memory the data objects $|O_i|$ and the list of *k* data objects with best scores, leading to space complexity: $O(|O_i|)$ since $|O_i| >> k$.

On the other hand, the time complexity is: $O(|O_i| * |F_i|)$, since in worst case for all data objects and for each feature object the score is computed. In practice, when using the early termination the processing cost of each Reducer is significantly smaller, since only few feature objects need to be examined before reporting the correct result set.

If we make the simplistic assumption that the work is shared fairly in the *R* Reducers (e.g., assuming uniform distribution and a regular uniform grid), then we can replace in the above formulas:

$$|Oi| = \frac{|O|}{R}$$

Let us also consider the *duplication factor* d_f of the feature dataset *F*, which is a real number that is grid-dependent and data-dependent, such that:

$$\bigcup_{i=1}^{R} |Fi| \ = d_f * \ |F|$$

Then, we can also replace in the above formulas:

$$|Fi| \ = \frac{d_f * |F|}{R}$$

Thus, the processing cost of a Reducer is proportional to:

$$|Oi| * \ |Fi| = \frac{|O|}{R} * \frac{d_f * |F|}{R}$$

3.5.2 Estimation of the Duplication Factor

In the following, we assume that the size *a* of each side of a grid cell is larger than twice the query radius, i.e., $a \ \geq \ 2r$, or equivalently $r \ \leq = \ \frac{a}{2}$. This is reasonable, since we expect *r* to be smaller than the size of a grid cell.

Depending on the area where a feature object is positioned, different number of duplicates of this object will be created. Figure 9 shows an example of a grid cell.

Given a feature object enclosed into a cell, we identify four different cases. If the feature object has a distance smaller than or equal to *r* from any cell corner then the feature object is enclosed in the area A_1 that is depicted as the dotted area. In this case, the feature object must be duplicated to all three neighboring cells to the corner of the cell.

If the feature object has a distance smaller than or equal to *r* from two cell borders but not from a cell corner then the feature object is enclosed in area A_3. This area is depicted with solid blue color defined by the four rectangles, but does not include the circles.

If located in A_3, then only 2 duplicates will be created (not on the diagonal cell). The third area is A_2 depicted as dashed area and corresponds to the feature objects that have a distance smaller than or equal to r from only one border of the cell. In this case, only one duplicate is needed. Finally, if the feature object is enclosed in the remaining area of the cell (white area, called area A_4), no duplication is needed.

Obviously, since it holds $r \leq = \frac{a}{2}$, any feature object that belongs to a cell is located in only one of these four areas. Let $|A_i|$ denote the surface of area A_i, and A denote the area of the complete cell. Then:

- $|A_1| = 4 \cdot \frac{\pi r^2}{4} = \pi r^2$
- $|A_2| = 4 \cdot r^2 - |A_1| = (4 - \pi)\, r^2$
- $|A_3| = 4 \cdot (\alpha - 2r)r$
- $|A_4| = \alpha^2 - |A_1| - |A_2| - |A_3| = (\alpha - 2r)^2$
- $|A| = \alpha^2$

Let $P(A_i)$ denote the probability that a feature object belongs to area A_i. Then, if we assume uniform distribution of feature objects in the space, we obtain the following probabilities: $(Ai) = \frac{|Ai|}{|A|}$. Based on this, given n feature objects enclosed in a cell, we can calculate the total number of feature objects (including duplicates), denoted by $\hat{n}$, of the n feature points:

$$\hat{n} = 3 \cdot n \cdot P(A_1) + 2 \cdot n \cdot P(A_2) + n \cdot P(A_3) + n$$

and we can calculate the duplication factor *df* for this cell:

$$\begin{aligned} df &= \frac{\hat{n}}{n} \\ &= 3 \cdot P(A_1) + 2 \cdot P(A_2) + P(A_3) + 1 \\ &= 3\,\frac{\pi r^2}{\alpha^2} + 2\frac{(4-\pi)r^2}{\alpha^2} + 4\frac{(a-2r)r}{\alpha^2} + 1 \\ &= \frac{\pi r^2}{\alpha^2} + \frac{4r}{a} + 1 \end{aligned}$$

Based on the above formula, we conclude that the worst value of d_f is $3 + \frac{\pi}{4}$ for the case of $a = 2r$ and it holds that $1 \leq d_f \leq 3 + \frac{\pi}{4}$ for any query range r such that $a \leq 2r$.

Also, the duplication factor depends only on the ratio of the cell size to the query range, under the assumption of uniform distribution. Moreover, the formula shows that smaller cell size α (compared to the query range r) increases the number of duplicated feature objects. Put differently, a larger cell size α reduces the duplication of feature objects.

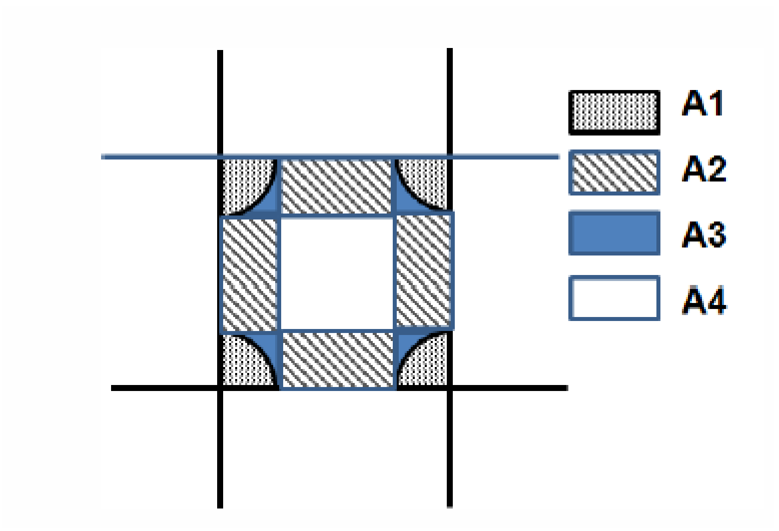

Figure 9: Breaking a cell in areas based on the number of duplicates.

3.5.3 Analysis of the Cell Size

Even though using a larger cell size α reduces the total number of feature objects, it also has significant disadvantages.

- First, it results in fewer cells thus reducing parallelism.
- Second, the probability of obtaining imbalanced partitions in the case of skewed data is increased.

Let us assume that all R cells can be processed in a single round, i.e., the hardware resources are sufficient to launch R Reduce tasks in parallel.

In this case, the total processing time depends on the performance of one Reducer, which as mentioned before depends on:

$$|Oi| \cdot |Fi| = \frac{|O|}{R} * \frac{d_f * |F|}{R} = |O| * |F| * \frac{d_f}{R^2}$$

If we normalize the dataset in [0, 1] × [0, 1], then $\alpha \leq 1$ and $= \frac{1}{\alpha}$.

Then, $|Oi| * |Fi| = |O| * |F| * d_f * a^4$.

In order to study the performance of one Reducer while varying α, it is sufficient to study $d_f * a^4$ since the remaining factors are constant. Based on the estimation of d_f in the previous section,

$$d_f * a^4 = \left(\frac{\pi r^2}{a^2} + \frac{4r}{a} + 1\right) * a^4 = \pi * r^2 * a^2 + 4 * r * a^3 + a^4$$

If we consider r as a constant, then for increasing positive values of a, the value of the previous equation increases, which means that the complexity of the algorithm increases. Thus, a smaller cell size α increases the number of cells and parallelism, and also reduces the processing cost of each Reducer.

3.5.4 Worst case performance of *eSPQlen* algorithm

The *eSPQlen* algorithm, as mentioned in section 5.1, accesses the feature objects by the number of features the have in |*f.W*| in increasing order. That means that the first feature objects that will be evaluated will be those that have at least a word in their |*f.W*|. We say at least because of the filter applied during the map phase (line 9 at Algorithm 3).

The early termination filter for the reduce function of the *eSPQlen* algorithm states:

$$\text{upper bound} \ = \ \overline{w}(f,q) = \begin{cases} 1 & ,|f.W| < |q.W| \\ \dfrac{|q.W|}{|f.W|} & ,|f.W| \geq |q.W| \end{cases}$$

Let’s assume the following scenario: It holds that $|f.W| \geq |q.W|$ but there are no feature objects that have score *w*|*f*,*q*| = $\frac{|q.W|}{|f.W|}$.

In that case the early termination condition will not be satisfied by the feature objects that have the minimum number of features which we denote as $|f.W|'$ from now on, nor will it be satisfied by any other feature objects that have number of features > $|f.W|'$,since the denominator on the Jaccard score index formula will be greater. For this specific reason the algorithm needs to update the upper bound value each time a new feature object with $|f.W|_{current}$ > $|f.W|'$

Lemma 4. (Worst case performance of *eSPQlen* algorithm) At the worst case, the *eSPQlen* algorithm will need to evaluate all the feature objects that have:

$$|f.W|_{current} \ \geq \ \frac{|q.W| * (\,|q.W| \cup |f.W|')}{|q.W| \cap |f.W|'}$$

where $|f.W|'$ stands for the last feature object that the top-*k* memory has.

Proof: First we assume that for a specific reducer we receive both some data objects and some feature objects. If this condition is not satisfied then that specific reducer cannot produce (and will not produce) any results. Now let’s assume that $|f.W| \geq |q.W|$ (The alternate possibility $|f.W| < |q.W|$will be examined later). Since the reducer has some feature objects, that means that all of those feature objects share at least one common term with the user specified query (otherwise they would be cut from further processing by the filter applied during the map phase of the algorithm). Since the feature objects are ranked (in increasing order) by the number of feature they have, the first feature object to be processed will have the $min|f.W| = |f.W|'$

If this specific feature object has one or more data objects in its vicinity (inside the circle with center the f.o and radius *r*) then that/those data objects will be part of the results list (the top-*k* memory will hold that/those data objects). In case that feature object is not near a data object, it will be discarded and the second feature object will be evaluated. If the second f.o doesn't have any data objects in its vicinity, it is discared as well etc, until a f.o that has one or more data objects in its vicinity is found. (In the highly unlikely case that no feature object has any data objects in its vicinity then the reducer will not produce any result so the results memory will be empty)

Since a feature object was found that has one or more data objects in its vicinity, we are interested in the number of features it has. That number is $|f.W|'$ and is important to know. Now the top-*k* memory will be filled with the id of this specific data object(s).

Since we keep the top-*k* memory sorted (decreasing order so that the best score comes first), the last position of the memory will hold the id of the first data object that satisfied the range query. Also let's assume that this specific feature object does not have a score that is equal to the upper limit score = $\frac{|q.W|}{|f.W|\prime}$ or in other words, not all of its terms are the same with the user specified query. In that case the early termination criterion will not be satisfied by this specific feature object and the next one will have to be evaluated.

Now let's assume that all the remaining feature objects that the reducer received do not have a score that is equal to their possible maximum score. That means that

$$w(f,q) < \bar{\tau} => \frac{|q.W| \cap |f.W|_{current}}{|q.W| \cup |f.W|_{current}} < \frac{|q.W|}{|f.W|_{current}}$$

This is the absolute worst case for the algorithm. The algorithm will continue to evaluate the feature objects and compare the current worst top-*k* score with the current upper limit/early termination criterion.

Since all of the feature objects have scores lower than their maximum possible, the algorithm will continue to evaluate feature objects up to the point that the early termination criterion is satisfied:

That is, $\overline{\tau} \geq \overline{w}(f,q)$. But the score $\overline{\tau}$ is equal to the score of the last object in the top-*k* memory or:

$$\overline{\tau} = \frac{|q.W| \cap |f.W|'}{|q.W| \cup |f.W|'}$$

By replacing this equation to the previous we get:

$$\overline{\tau} \geq \overline{w}(f,q) => \frac{|q.W| \cap |f.W|'}{|q.W| \cup |f.W|'} \geq \frac{|q.W|}{|f.W|_{current}}$$

By solving this equation for $|f.W|_{current}$ we get that:

$$|f.W|_{current} * |q.W| \cap |f.W|' \geq |q.W| * |q.W| \cup |f.W|' =>$$

$$|f.W|_{current} \geq \frac{|q.W| * |q.W| \cup |f.W|'}{|q.W| \cap |f.W|'} \blacksquare$$

We now need to examine the opposite possibility, that $|f.W| < |q.W|$. All other conditions are the same as above (worst case for all f.o not having a score equal to the upper limit).

It is easy to see that in this case nothing changes. The upper limit score is still updated each time $|f.W|_{current}$ is increased. Regardless of the $|f.W|'$ the first f.o that makes it into the results memory has, the early termination criterion will check the $\overline{\tau} \geq \overline{w}(f,q)$ condition.

By following the same computation as above, we reach the exact same formula.

This proves the lemma.

3.6. Experimental evaluation

In this section, we report the results of our experimental study. All algorithms are implemented in Java.

Table 3: Experimental parameters (default values in bold)

Parameter	Values

Datasets	Real: {TW, FL}
	Synthetic: {UN, CL}

Query Keywords (|q.*W*|)	1, **3**, 5, 10
Query Radius (*r*) (% of grid cell side *α*)	5%, **10**%, 25%, 50%
Top-*k*	5, **10**, 50, 100
Grid size (FL, TW)	35x35, **50x50,** 75x75, 100x100
Grid size (UN, CL)	10x10, **15x15**, 50x50, 100x100

3.6.1 Experimental Setup

Platform: We deployed our algorithms in an in-house CDH cluster consisting of sixteen (16) server nodes. Each of the nodes d1-d8 have 32GB of RAM, 2 disks for HDFS (5TB in total) and 2 CPUs with a total of 8 cores running at 2.6 GHz. The nodes d9-d12 have 128GB of RAM, 4 disks for HDFS (8TB in total) and 2 CPUs with a total of 12 cores (24 hyper threads) running at 2.6 GHz. Finally, each of the nodes d13-d16 is equipped 128GB RAM, 4 disks for HDFS (8TB in total), and 2 CPUs with a total of 16 cores running at 2.6GHz.

Each of the servers in the cluster functions as DataNode and NodeManager, while one of them in addition functions as NameNode and ResourceManager. Each node runs Ubuntu 12.04. We use the CDH 5.4.8.1 version of Cloudera and Oracle Java 1.7.

The JVM heap size is set to 1GB for Map and Reduce tasks. We configure HDFS with 128MB block size and use a default replication factor of 3.

Datasets: In order to evaluate the performance of our algorithms we used four different large-scale datasets.

Two real datasets are included, a dataset of tweets obtained from Twitter and a dataset of images obtained from Flickr.

The Twitter dataset (TW) was created by extracting approximately 80 million tweets which requires 5.7GB on disk. Besides a spatial location, each tweet contains several keywords extracted from its text, with 9.8 keywords on average per tweet, while the size of the dictionary is 88,706 keywords.

The Flickr dataset (FL) contains metadata of approximately 40 million images, thus capturing 3.5GB on disk. The average number of keywords per image is 7.9 and the dictionary contains 34,716 unique keywords.

In addition, we created two synthetic datasets in order to test the scalability of our algorithms with even larger datasets.

The first synthetic dataset consists of 512 million spatial (data and feature) objects that follow a uniform (UN) distribution in the data space. Each feature object is assigned with a random number of keywords between 10 and 100, and these keywords are selected from a vocabulary of size 1,000. The total size of the file is 160GB.

The second synthetic dataset follows a clustered (CL) distribution. We generate 16 clusters whose position in space is selected at random. All other parameters are the same. The total size of the generated dataset is 160GB, as in the case of UN. Figures 9,10,11 depict the spatial distribution of the datasets employed in our experimental study. In all cases, we randomly select half of the objects to act as data objects and the other half as feature objects.

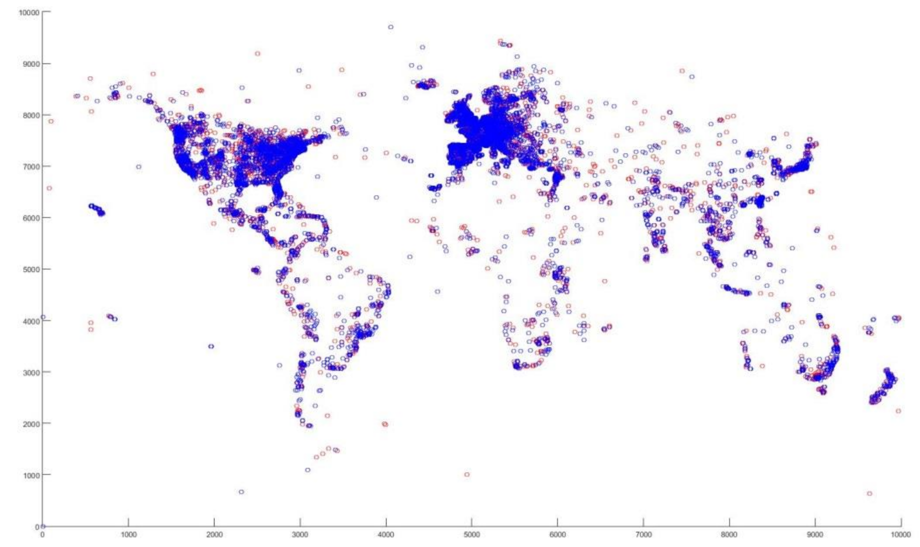

Figure 9: Spatial distribution of flickr dataset

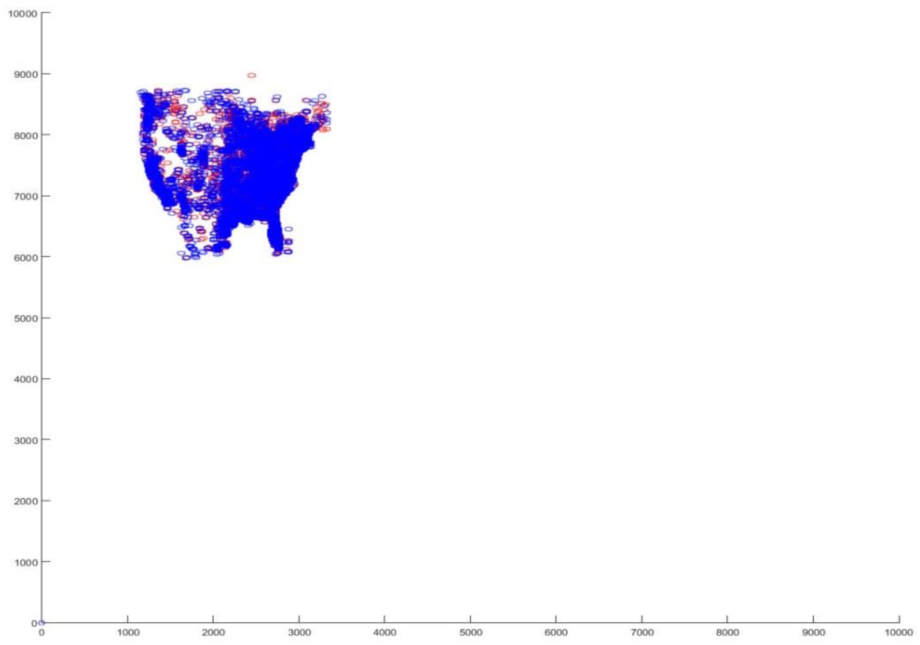

Figure 10: Spatial distribution of twitter dataset

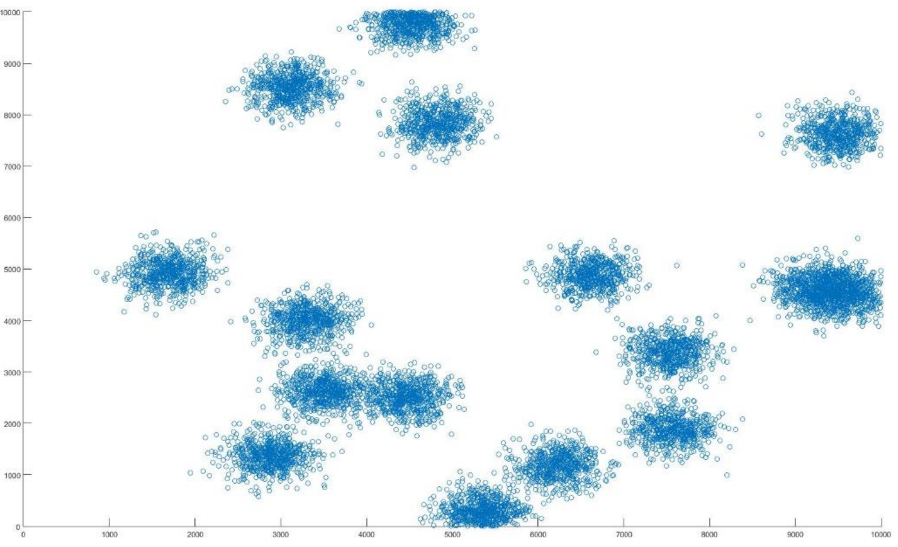

Figure 11: Spatial distribution of clustered dataset

Algorithms: We compare the performance of the following algorithms that are used to compute the spatial preference query using keywords in a distributed and parallel way in Hadoop:

- *pSPQ*: the parallel grid-based algorithm without early termination ,
- *eSPQlen*: the parallel algorithm that uses early termination by accessing feature objects based on increasing keyword length, and
- *eSPQsco*: the parallel algorithm that uses early termination by accessing feature objects based on decreasing.

For clarification purposes, we note that centralized processing of this query type is infeasible in practice, due to the size of the underlying datasets and the time necessary to build centralized index structures.

Query generation: Queries are generated by selecting various values for the query radius *r* and a number of random query keywords *q.W* from the vocabulary of the respective dataset. We also explored alternative methods for keyword selection instead of random selection, such as selecting from the most frequent words or the least frequent words, but the execution time of our algorithms was not significantly affected. This is shown in figures 30 and 31.

Parameters: During the experimental evaluation a number of parameters were varied in order to observe their effect on each algorithm’s runtime. These parameters, reported in Table 3, are:

- the radius of the query,
- the number of keywords of the query,
- the size of the grid that we use to partition the data space,
- the number of the *k* results that the algorithm returns, and
- the size of the dataset.

In all cases, the number of Reducers is set equal to the number of cells in the spatial grid.

Metrics: The algorithms are evaluated by the time required for the MapReduce job to complete, i.e., the job execution time. Another metric that was considered is the duplication factor of the feature objects during the MapReduce job. This was done in order to get a better insight of the dependence of the execution time and the grid size. The duplication of the feature objects was measured and plotted. The results can be seen at figure 12.

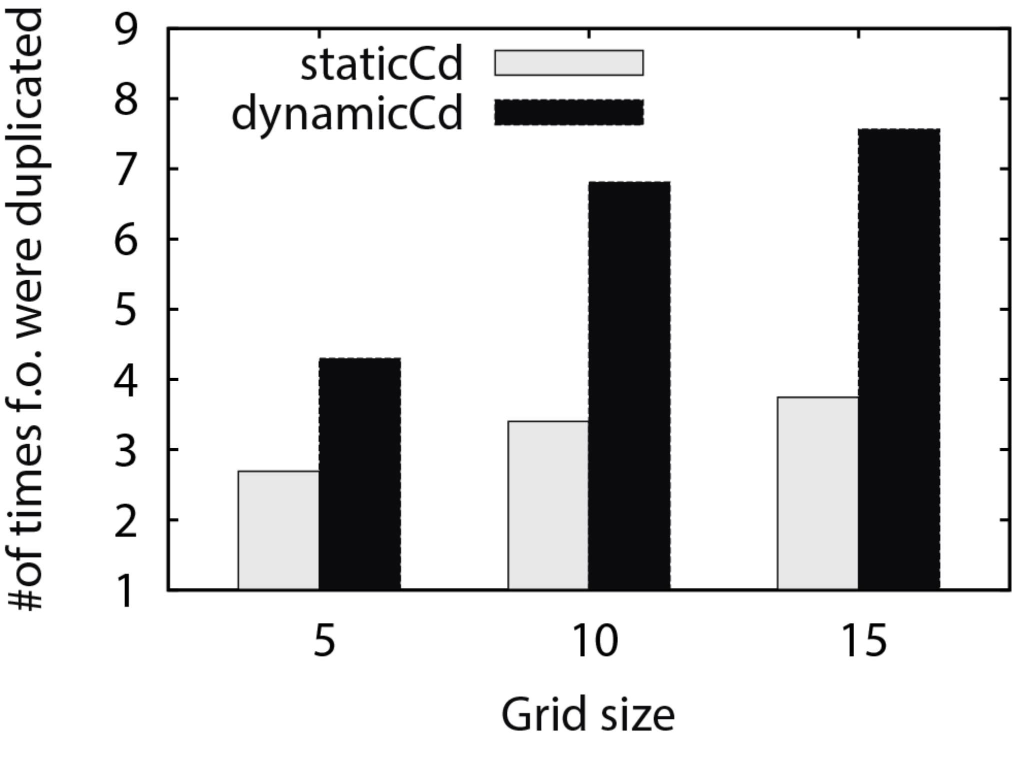

Figure 12: Duplication of feature objects

3.6.2 Experimental Results

3.6.2.1 Experiments with Real Data: Flickr

Figures 13, 14, 15 and 16 present the results obtained for the Flickr (FL) dataset.

First, in Figure 13, we study the effect of grid size to the performance of our algorithms. The first observation is that using more grid cells (i.e., Reduce tasks) improves the performance, since more, yet smaller, parts of the problem need to be computed.

The algorithms that employ early termination (*eSPQlen*, *eSPQsco*) are consistently much faster than the grid-based algorithm *pSPQ*. In particular, *eSPQsco* improves *pSPQ* up to a factor of 6x. Between the early termination algorithms, *eSPQsco* is consistently faster due to the sorting based on score, which typically needs to access only a handful of feature objects before reporting the correct result.

Figure 14 shows the effect of varying the number of query keywords ($|q.W|$). In general, when more keywords are used in the query more feature objects are passed to the Reduce phase, since the probability of having non-zero Jaccard similarity increases.

This is more evident in *pSPQ*, whose cost increases substantially with increased query keyword length. Instead, *eSPQsco* is not significantly affected by the increased number of keywords, because it still manages to find the correct result after examining only few feature objects.

This experiment demonstrates the value of the early termination criterion employed in *eSPQsco*.

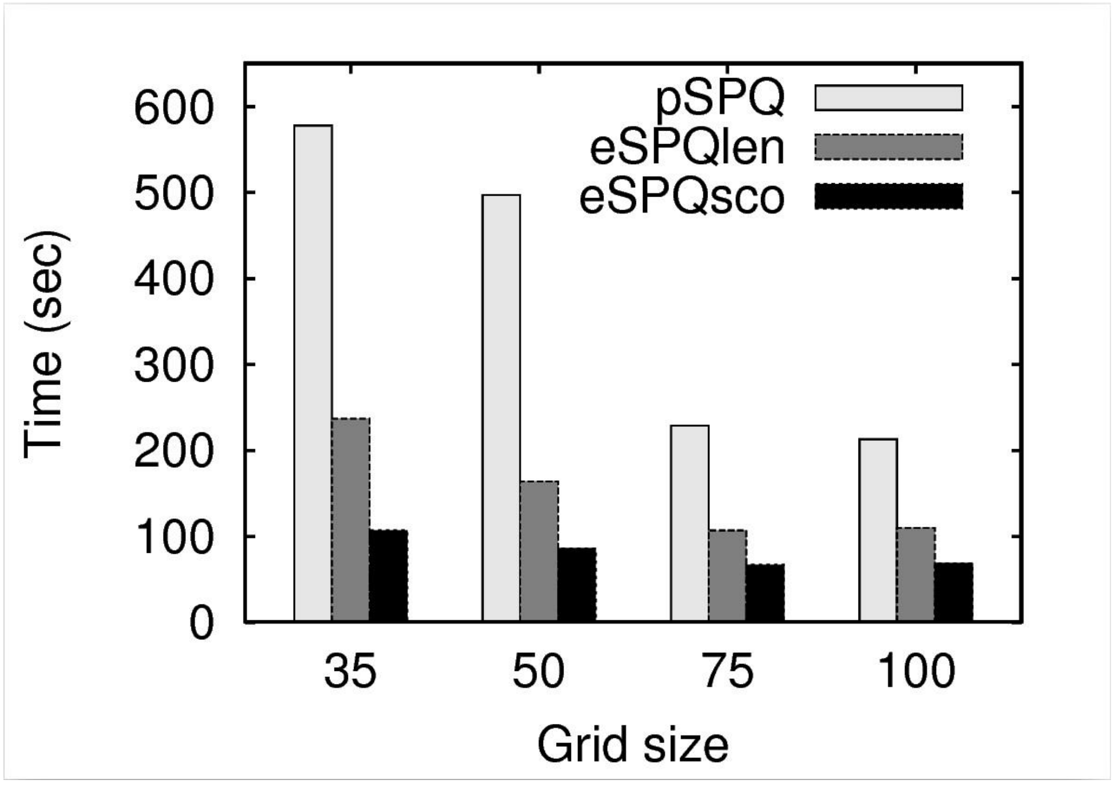

Figure 13: Flickr dataset grid size experiment results

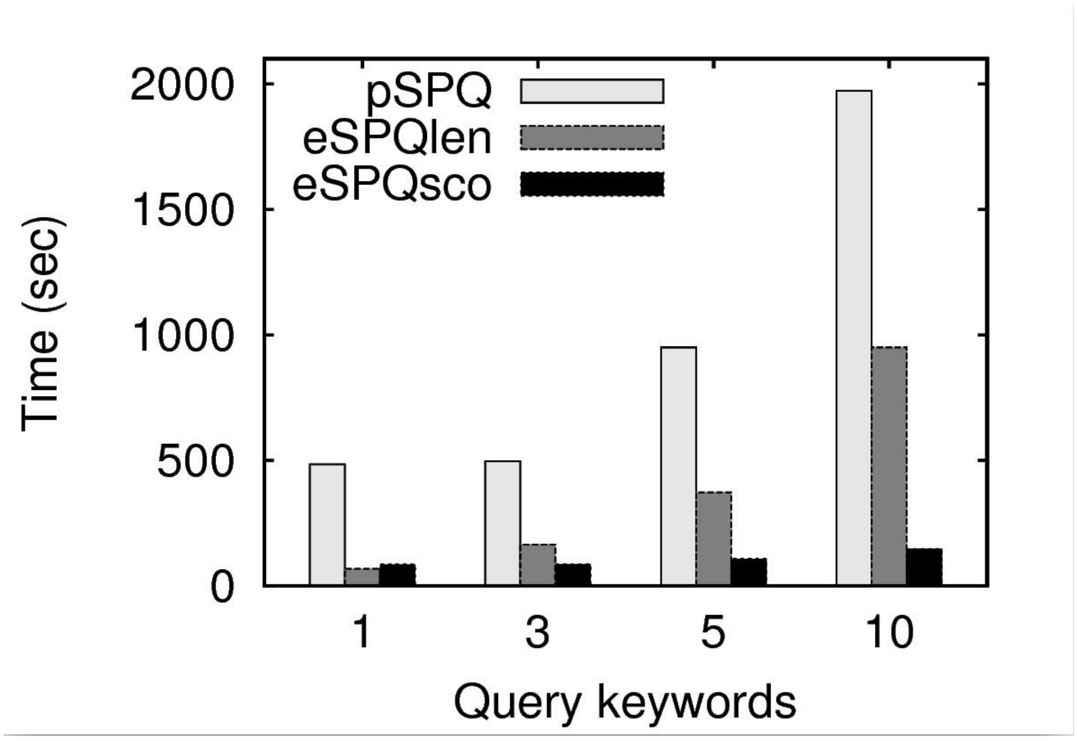

Figure 14: Flickr dataset keywords experiment results

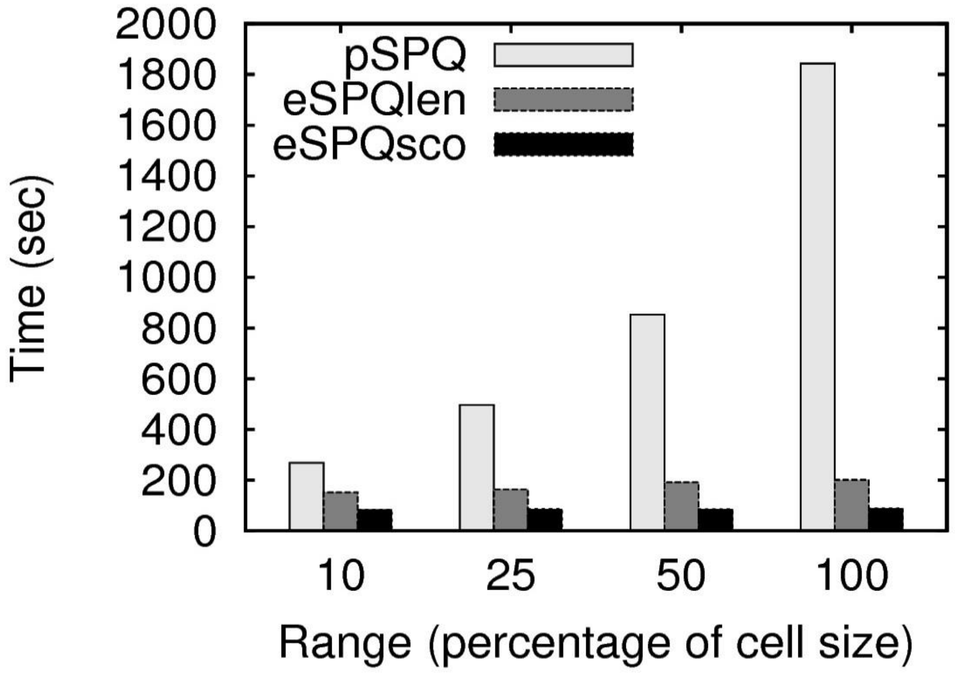

Figure 15: Flickr dataset range experiment results

In Figure 15, we gradually increase the radius of the query. In principle, this makes query processing costlier as again more feature objects become candidates for determining the score of any data object.

However, the early termination algorithms are not significantly affected by increased values of radius, as they can still report the correct result after examining few feature objects only.

Finally, in Figure 16, we study the effect on increased values of top-*k*. The chart shows that all algorithms are not particularly sensitive to increased values of *k*, because the cost of reporting a few more results is marginal compared to the work needed to report the first result.

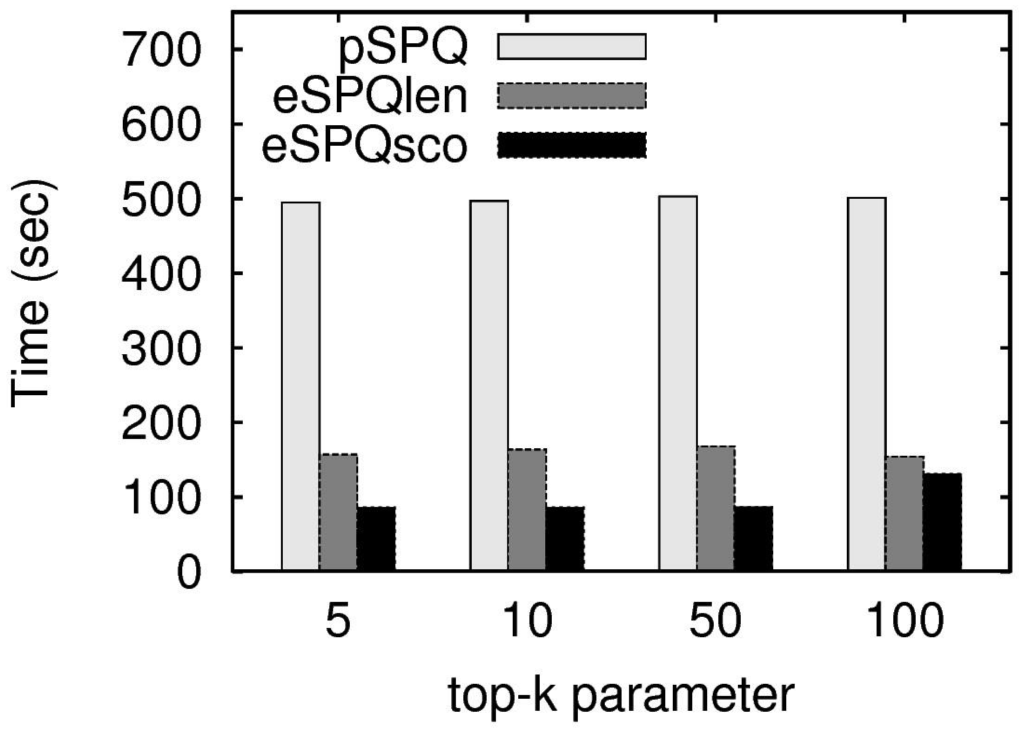

Figure 16: Flickr dataset top-*k* parameter experiment results

3.6.2.2 Experiments with Real Data: Twitter

Figures 17, 18, 19 and 20 depict the results obtained in the case of the Twitter (TW) dataset. In general, the conclusions drawn are quite similar to the case of the FL dataset. Algorithms that employ early termination, and in particular *eS-PQsco*, scale gracefully in every setup.

Even in the harder setups of many query keywords (Figure 18) and larger query radius (Figure 19), the performance of *eSPQsco* is not significantly affected. This is because as soon as the first few feature objects with highest scores are examined, the algorithm can safely report the top-*k* data objects in the cell. In other words, the vast majority of feature objects assigned to a cell are not actually processed, and exactly this characteristic makes the algorithm scalable both in the case of more demanding queries as well as in the case of larger datasets.

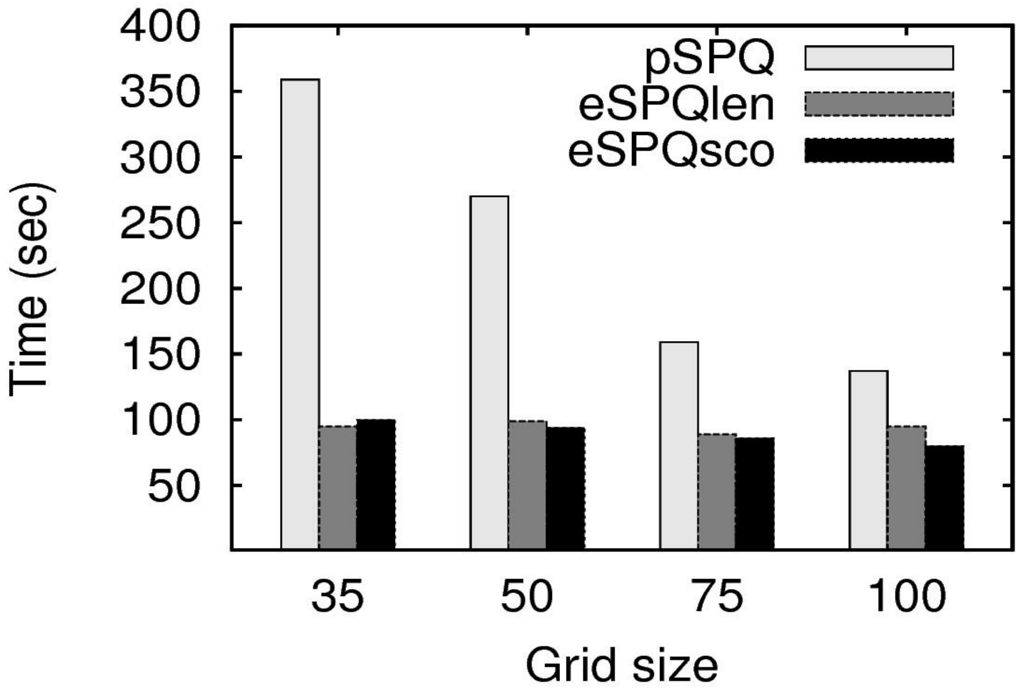

Figure 17: Twitter dataset grid size experiment results

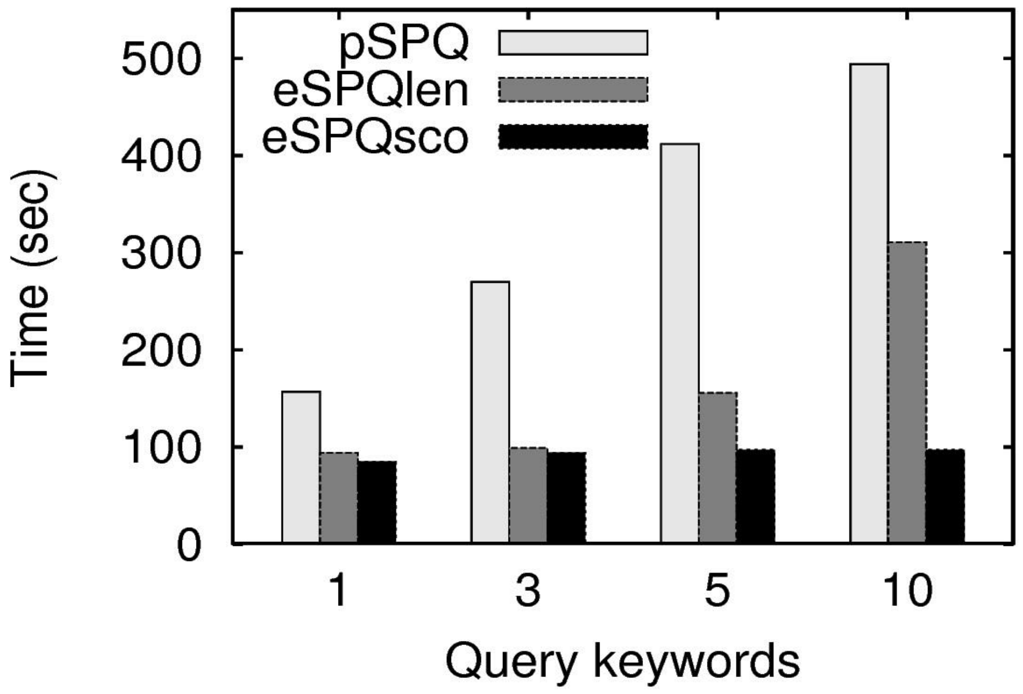

Figure 18: Twitter dataset keywords experiment results

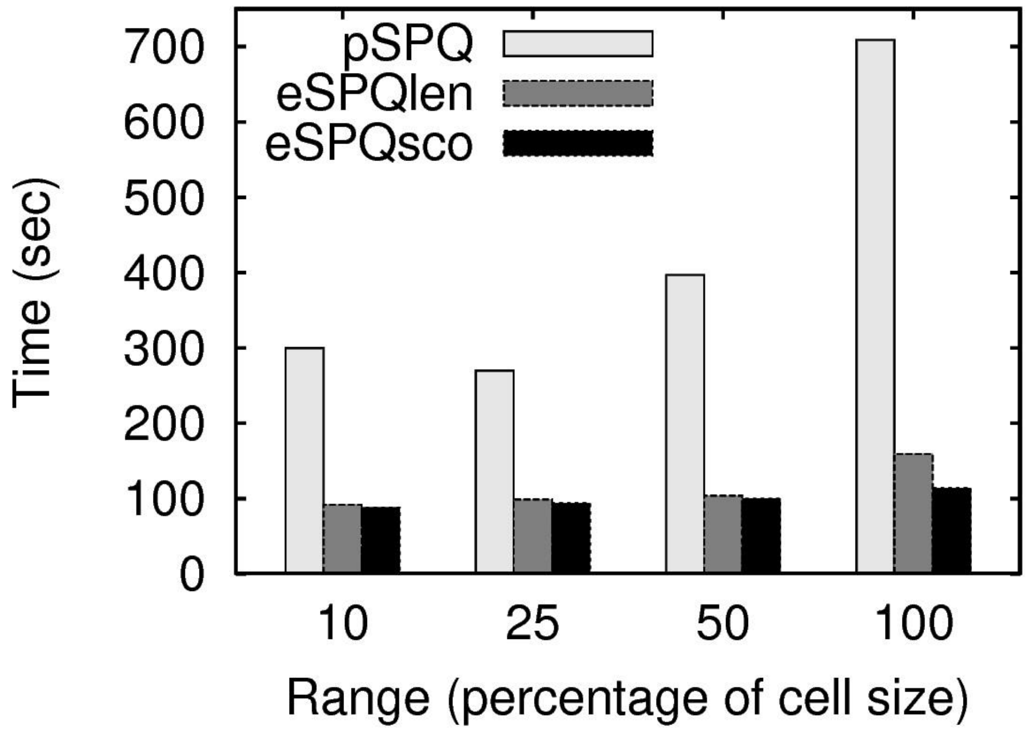

Figure 19: Twitter dataset range experiment results

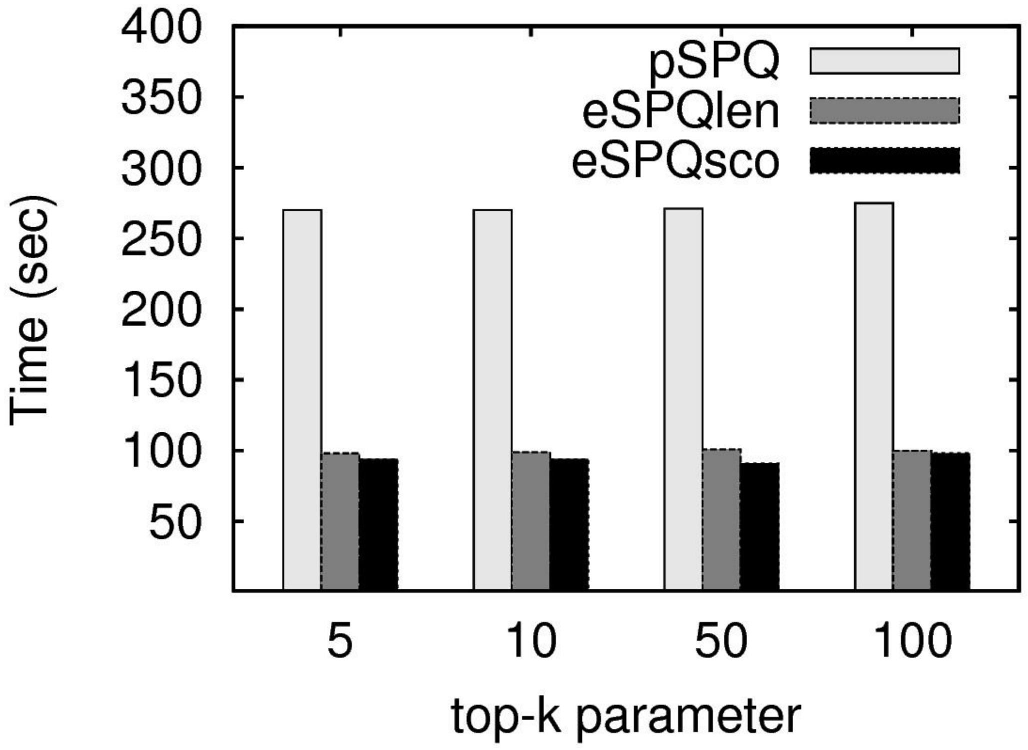

Figure 20: Twitter dataset top-*k* parameter experiment results

3.6.2.3 Experiments with Uniform Data

In order to study the performance of our algorithms for large-scale datasets, we employ in our study synthetic datasets. Figures 22, 23, 24 and 25 present the results obtained for the Uniform (UN) dataset. Notice the log-scale used in the y-axis. A general observation is that *eSPQsco* that uses early termination outperforms *pSPQ* by more than one order of magnitude. This is a strong indication in favor of the algorithms employing early termination, as their performance gains are more evident for larger datasets, such as the synthetic ones used in our study. Also, the general trends in the charts are in accordance with the conclusions derived from the real datasets. It is noteworthy that the performance of *eSPQsco* remains relatively stable in the harder setups consisting of many query keywords (Figure 23) and larger query radius (Figure 24).

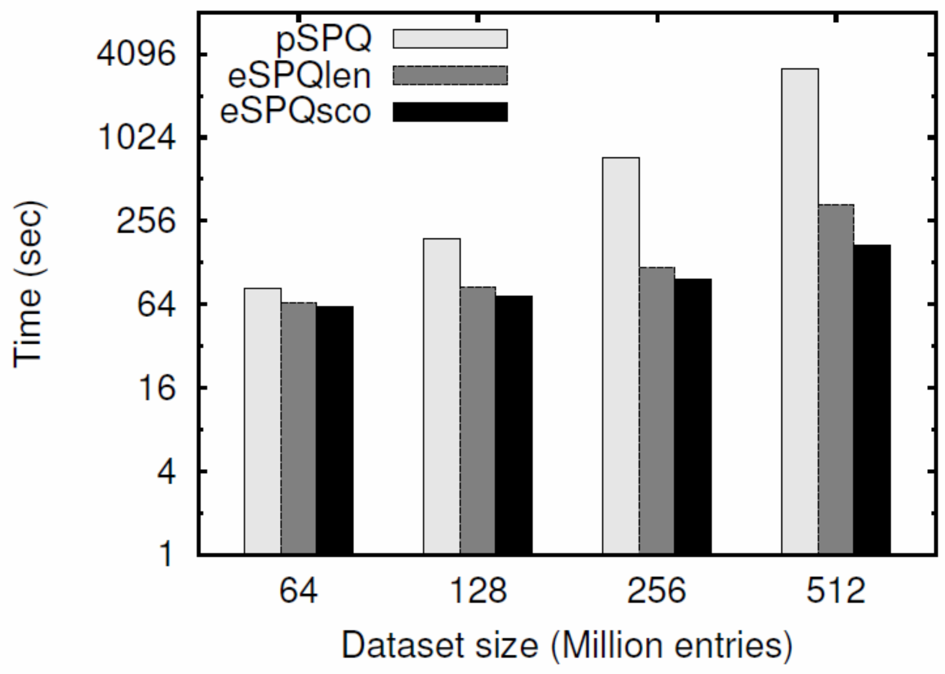

Figure 21: Scalability of our algorithms performance when varying the dataset size

Moreover, Figure 20 shows the results obtained when we vary the dataset size. This experiment aims at demonstrating the nice scaling properties of our algorithms. In particular, *pSPQ* scales linearly with increased dataset size, which is already a good result. However, the algorithms that employ early termination perform much better, since they only examine few feature objects regardless of the increase of dataset size. The experiment also shows that the gain of the algorithms that employ early termination compared to *pSPQ* increases for larger datasets.

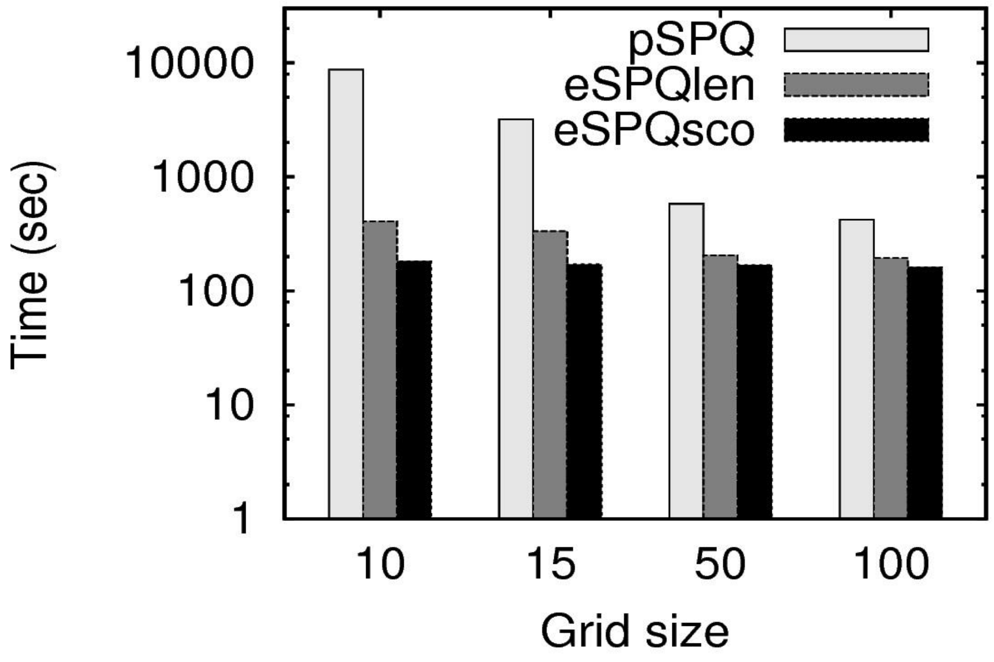

Figure 22: Uniform dataset grid size experiment results

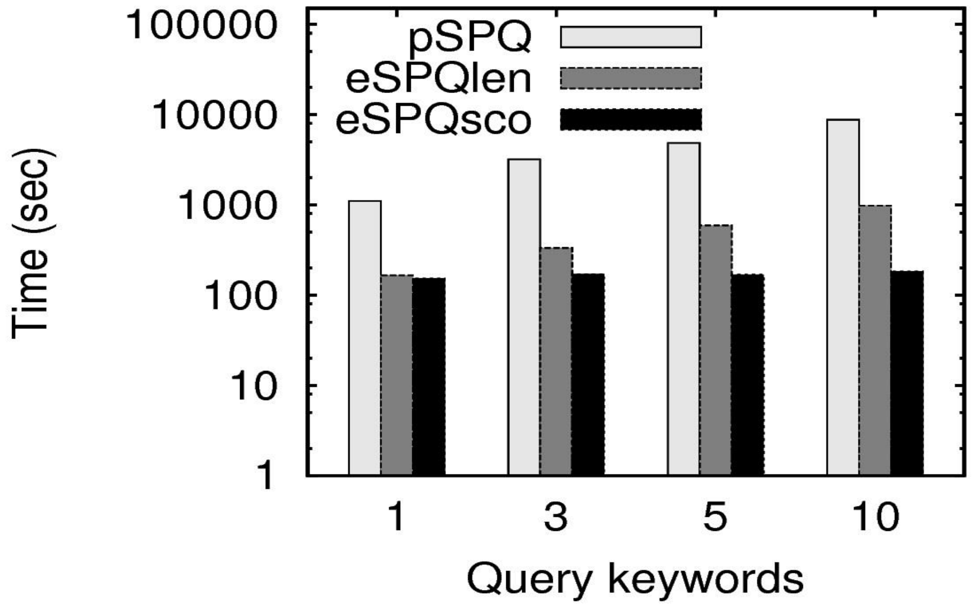

Figure 23: Uniform dataset keywords experiment results

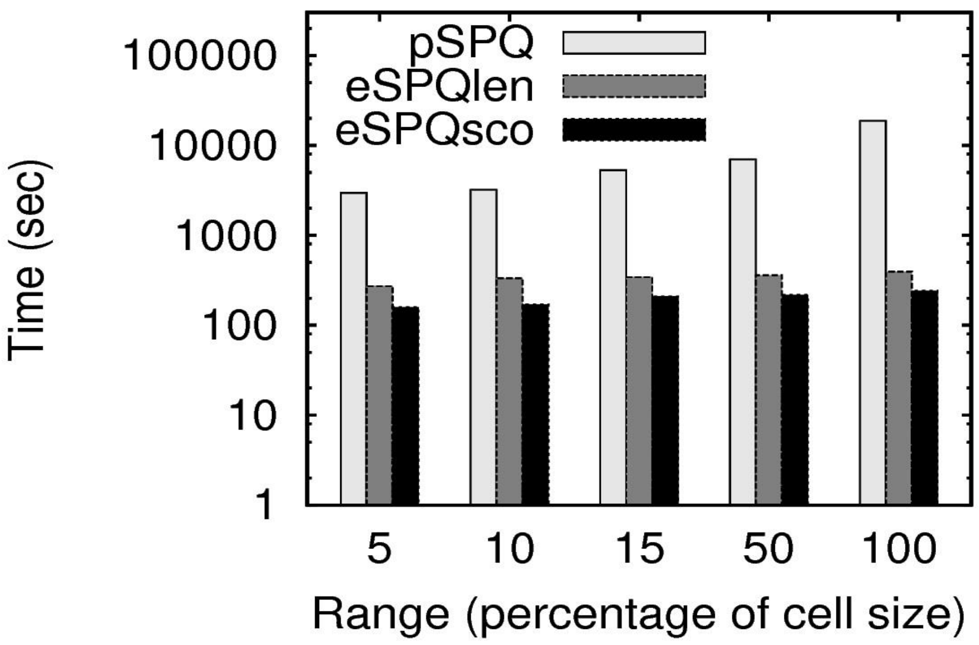

Figure 24: Uniform dataset range experiment results

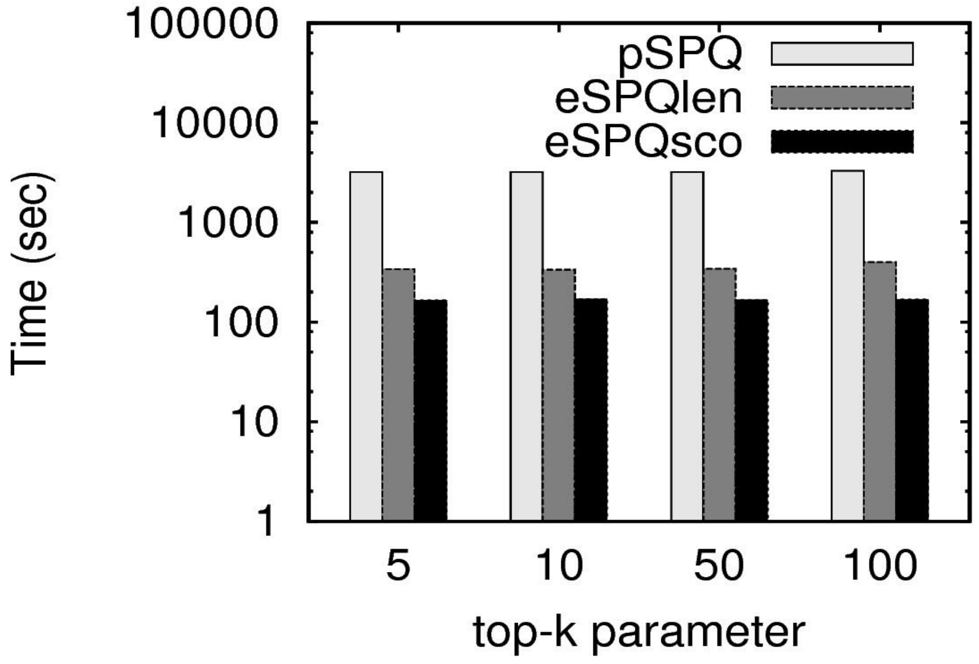

Figure 25: Uniform dataset top-*k* parameter experiment results

3.6.2.4 Experiments with Clustered Data

Figures 26, 27, 28 and 29 present the results obtained for the Clustered(CL) dataset. It should be noted that such a data distribution is particularly challenging as: (a) it is hard to fairly assign the objects to Reducers, thus typically some Reducers are overburdened, and (b) excessive object duplication can occur when a cluster is located on grid cell boundaries.

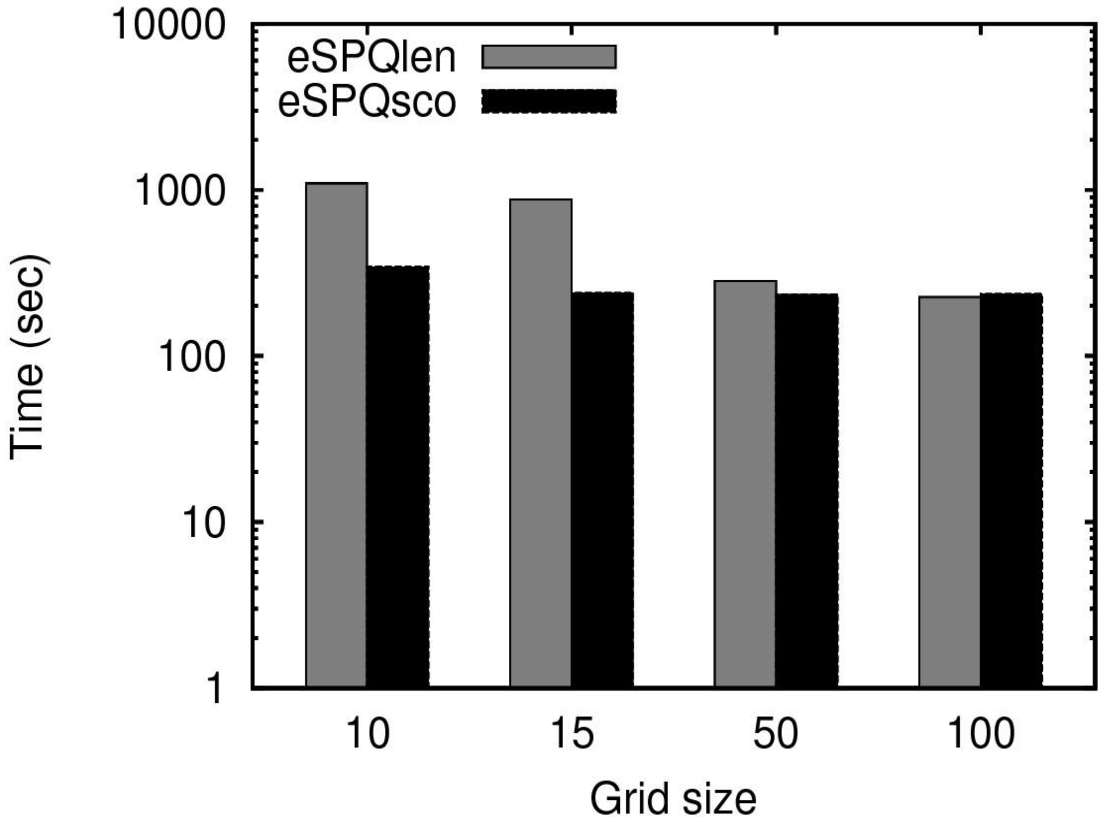

Figure 26: Clustered dataset grid size experiment results

For the CL dataset, we observed that *pSPQ* results in extremely high execution time, thus it is not depicted in the charts. For instance, for the default setup, it takes approximately 48 hours for *pSPQ* to complete. This is due to the fact that some Reducers are assigned with too many feature objects and *pSPQ* has to perform O(|*Oi*| • |*Fi*|) score computations before termination. Still, the algorithms employing early termination perform much better in all cases.

Again, when *eSPQsco* is considered, its performance is the best among all algorithms, and it remains quite stable even in the case of more demanding queries. This experiment verifies the nice properties of *eSPQsco*, even for the combination of large-scale dataset with a demanding data distribution.

By using an emulator that emulates the map/reduce job (see section 3.8), we can verify that at most cases, the *eSPQsco* only evaluates a handful of feature objects in order to produce the result (usually only 1). That explains its significantly better performance than *eSPQlen.*

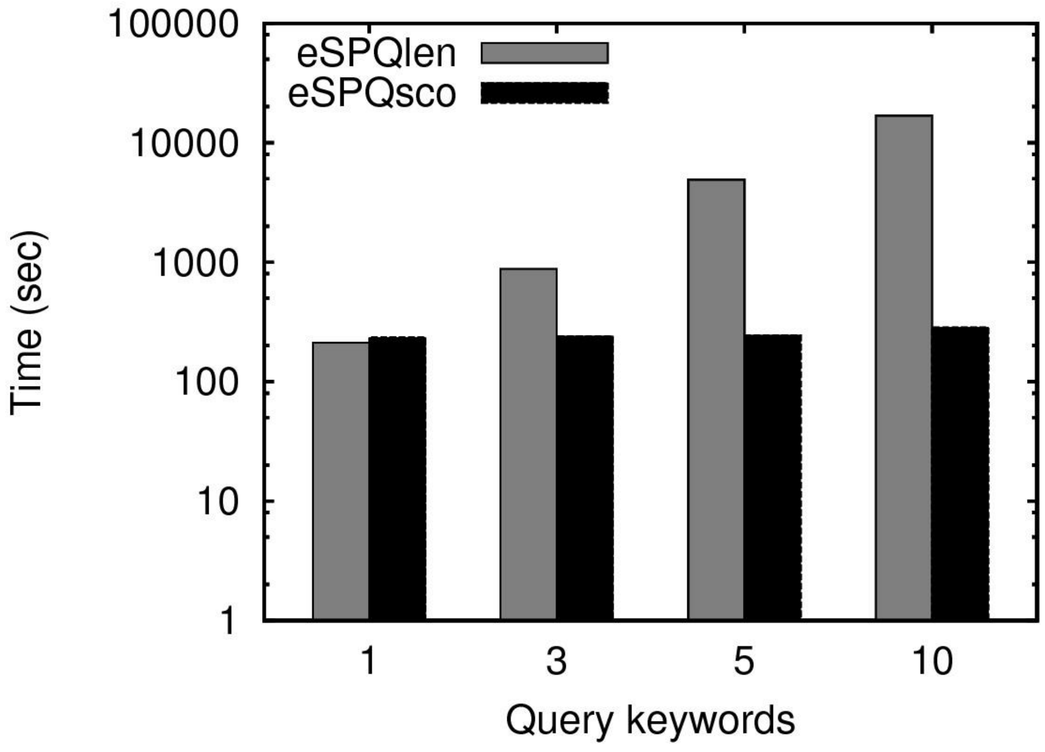

Figure 27: Clustered dataset keywords experiment results

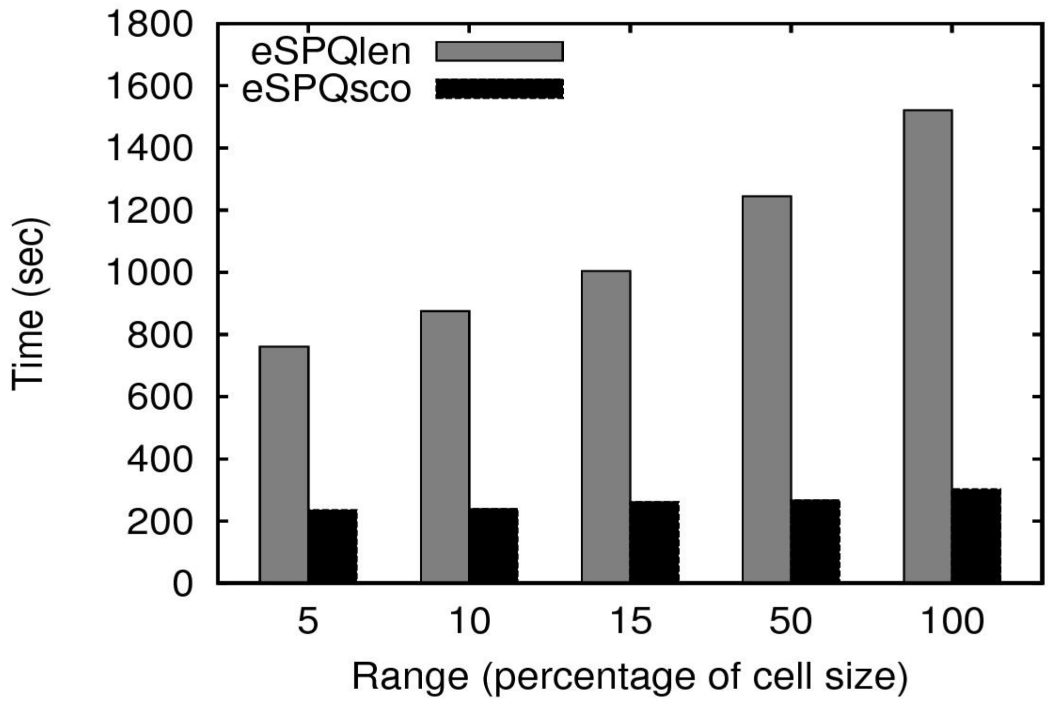

Figure 28: Clustered dataset range experiment results

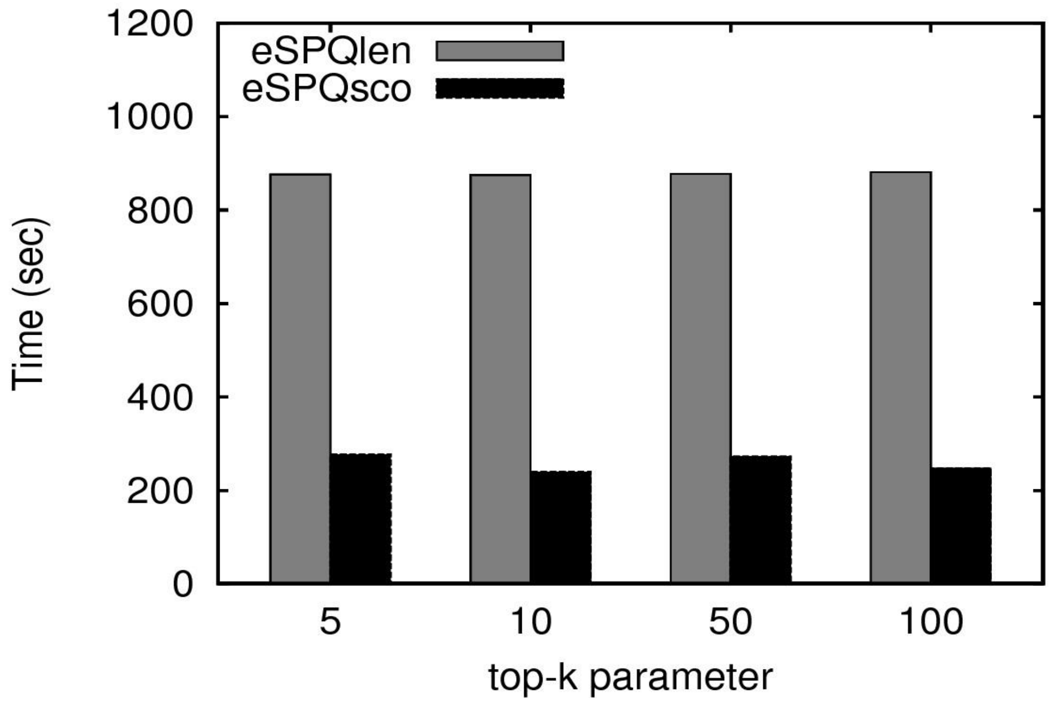

Figure 29: Clustered dataset top-*k* parameter experiment results

3.6.2.5 Query keyword selection and execution time

In this section we describe the effect the query keyword selection method has on the execution time. We designed and executed two experiments to show that dependence. We chose the eSPQsco algorithm as our basis and run several queries keeping all parameters the same, except for the query keywords.

In order to be as thorough as possible, we chose to run the queries with some of the most frequent words in the dataset, some of the least frequent and some random.

This way we cover all possibilities and if a relation between the method of selecting the keywords and execution time exists, it will be reflected on the results.

Also we run the experiment on all of our available datasets in order to verify that the results are not dataset dependent.

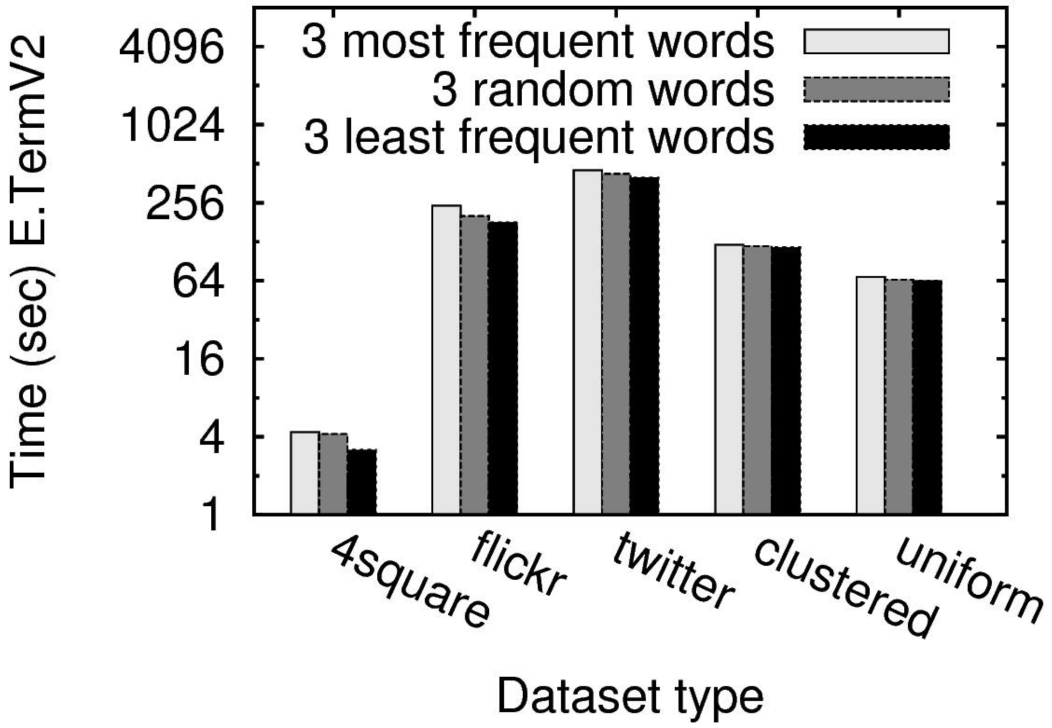

Figure 30: *eSPQsco* execution time based on keyword selection (1/2)

In figure 30 we can see the first experiment: We chose the 3 most common, the 3 least common and 3 random words as the query input. As we can see, for all the datasets, the execution time does not vary significantly.

In figure 31 we can see the second experiment. This time we chose 10 of the most common, least common and random words as the query input. Again the results are not affected in any significant way by the process of the query keyword selection.

This justifies the decision we made to use a random selection of keywords for each set of experiments on our datasets. (That selection remained the same during the run of all the experiments for that specific dataset)

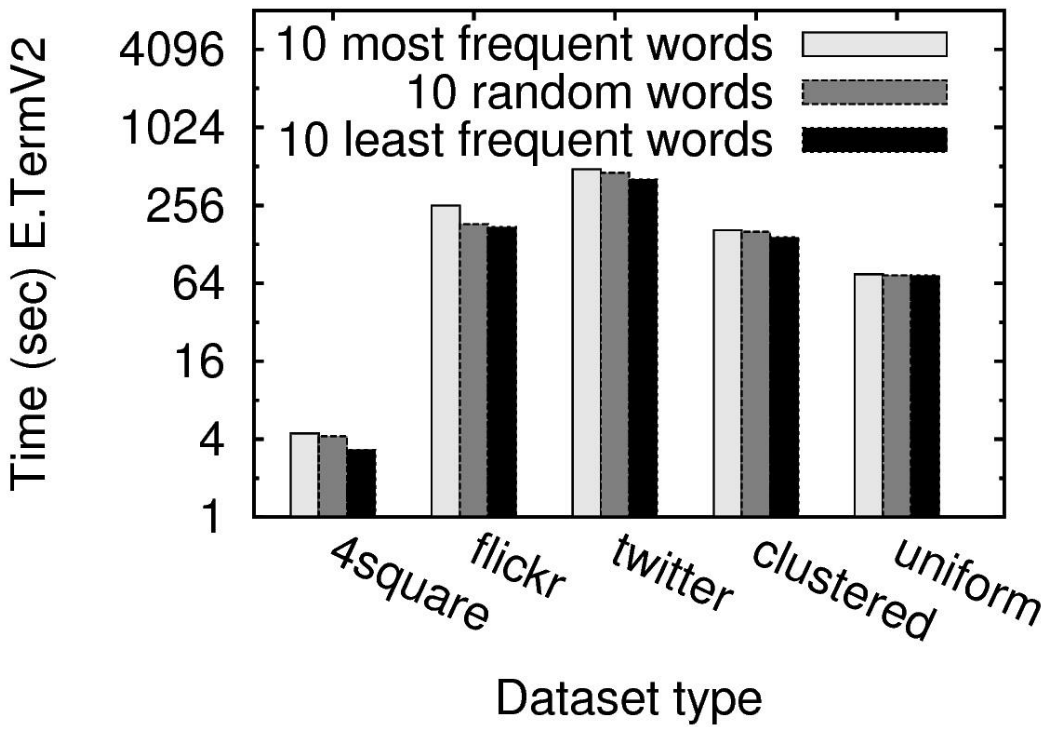

Figure 31: *eSPQsco* execution time based on keyword selection (2/2)

3.7. Performance improvement techniques

In this section we will examine possible improvements that can be achieved regarding the performance of the algorithms. Those improvements are not on the algorithms themselves – we showed that the *eSPQsco* algorithm is the optimal, but have to do with the grid partitioning method of the datasets.

3.7.1 Dynamic Grid

So far we only considered the use of a uniform grid. That is logical and of course yields the best possible results in case of datasets that their objects follow a uniform distribution. But as we saw in the case of the clustered dataset experiment, the performance of the algorithms is highly dependent on the distribution of the objects inside the dataset. This is expected. As we can see in figure 32 that depicts a clustered dataset, the objects (both data and feature) are in close proximity to each other and more importantly, do not take up much of the available dataset's spatial dimensions. That in turn will lead to a highly unbalanced load to the reducers using the uniform grid partitioning method. Some reducers will receive no input, while others will be called to carry out the bulk of the computation. In the same picture we can see an example 5x5 static grid. Notice how many of the cells are empty.

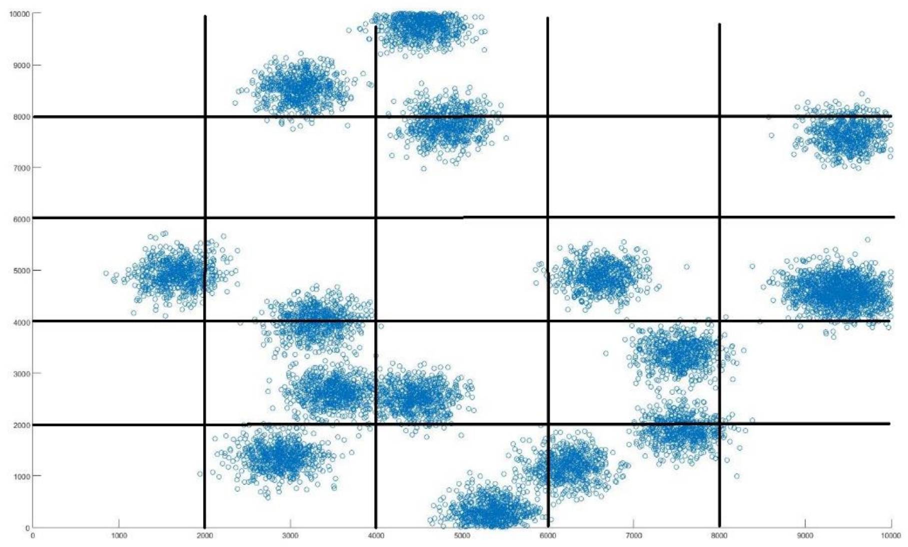

Figure 32: Clustered file with 5x5 static grid

To address this issue we consider a dynamic grid partitioning method. This method takes into account the spatial distribution of the data and finds an improved way of partitioning the dataset into cells. By achieving such a partitioning scheme, the processing load is distributed much more evenly to the reducers, thus significantly reducing the total processing time.

To achieve such a partitioning scheme, the dataset needs to be preprocessed. In other words, we need to introduce a new step before the execution of our algorithms. That step will need to find the best possible partitioning scheme without incurring a significant time penalty to the total execution of our MapReduce jobs. For example, if we wanted to find the centroids of a clustered dataset of 512m entries, the use of an algorithm like k-means would be inadvisable. It's time complexity is greater than O(n^2), (where n the dataset size) that our worst algorithm requires to solve the problem. So most of the time would be spent on trying to optimize the solution, than the actual solution of the problem itself.

In figure 33 we can see the same clustered dataset as in figure 32, but this time partition with a 5x5 dynamic grid. Notice the large variation between the cell sizes. Also far less cells are empty.

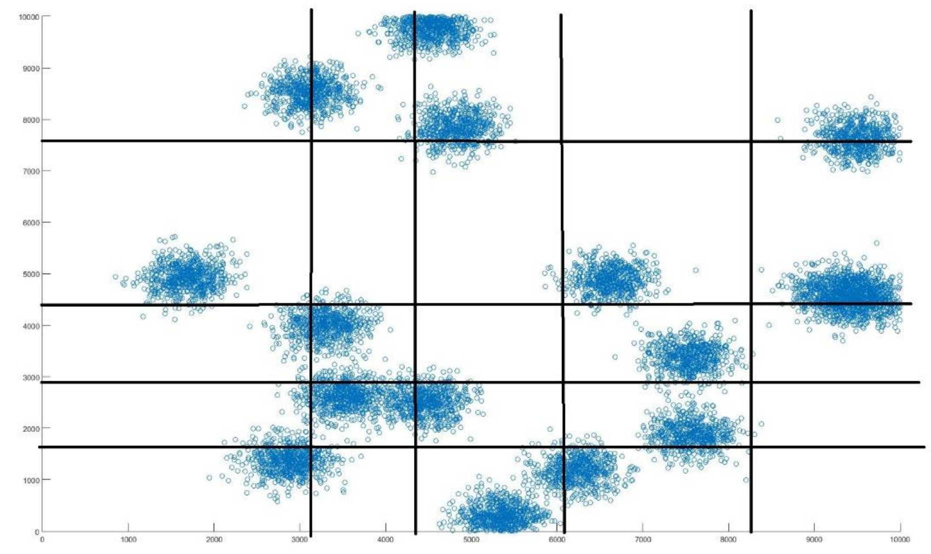

Figure 33: Clustered file with 5x5 dynamic grid

3.7.2 Reservoir Sampling

To address this issue we propose the use of a sampling method like reservoir sampling.

The reservoir sampling method operates in the following manger: An array with fixed size ,i, is specified – that is the reservoir. Then the dataset is processed linearly and the first values of the dataset (up to the size of the reservoir) are stored in the reservoir array (the coordinates of the objects). When the *i*+1 element of the dataset is processed it will replace a random element of the reservoir with probability $\frac{1}{i}$. This probability is gradually decreased as the dataset is processed.

By providing a reasonable size for the reservoir (eg 1% of the dataset size), at the end of the procedure the reservoir will hold a representative sample for the distribution of the objects.

The next step is to sort the reservoir in increasing order.

Since we use a two-dimensional space, we need to create 2 reservoirs, one for each axis. Now, instead of using the static grid partitioning method that would produce a cell at each of the:

$$datasetSize/\sqrt{numOfcells}$$

we split at:

$$reservoirSize[reservoirSize/\sqrt{numOfcells}]$$

(the element of the reservoir array at position $reservoirSize/\sqrt{numOfcells}$).

This method produces a much fairer grid in terms of data distribution in each of its cells. Instead of having some grid cells overloaded with points of interest and others almost empty, the load is balanced. As a result a significant reduction in the total processing time is achieved, especially for non-early termination algorithms (since they process all of the points of interest in the cell).

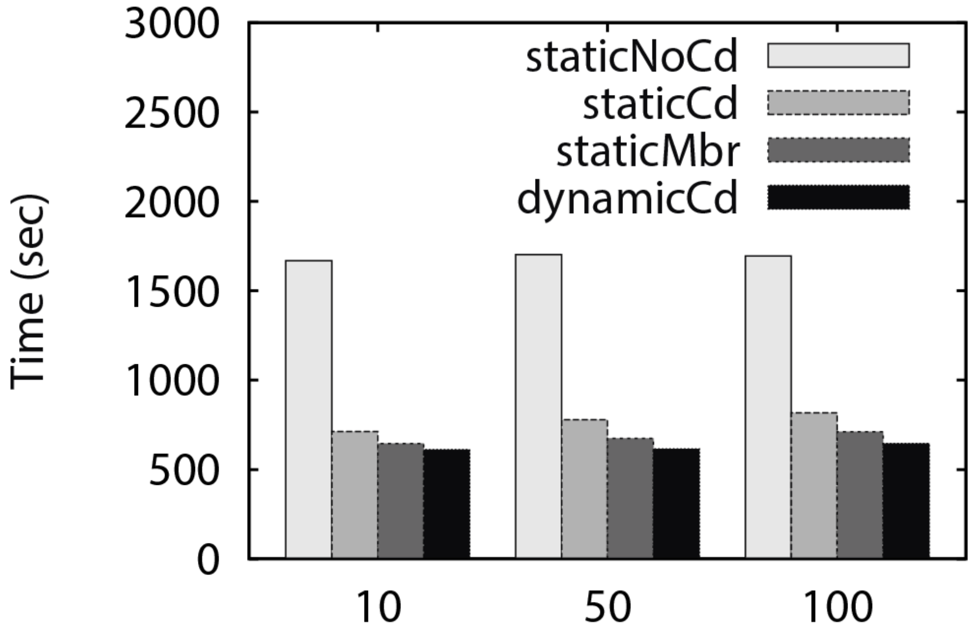

Figure 34: Gains by using mbrs or dynamic grid

Another possible improvement that has to do with the duplication factor of the feature objects, is the use of mbr's (maximum bounding rectangles) of the grid cells instead of their physical limits during the collision detection test. In cases of real datasets that do not follow uniform distribution, the maximum bounding rectangle of a cell usually has significantly lower area than the cell itself. If this is the case, the collision detection filter will produce significantly fewer duplicates of the feature objects. This in turn has an effect of the performance, especially in the *pSPQ* algorithm.

The experimental results for the aforementioned techniques can be seen in figure 34. In that figure we see how the performance of one of our algorithms (*pSPQ*) is affected by the use of a dynamic grid or mbrs. The most notable gains are given by the collision detection filter (which determines to which cells a f.o is duplicated) – if cd is off then each f.o is duplicated to all of its adjacent cells, thus significantly increasing the total number of objects each reducer receives. The use of mbrs shaves off some of the execution time while the dynamic grid offers slightly better results.

The gains are even more impressive if we take into account that the total execution time includes the time required by the map phase, which is constant (since the same piece of code runs for all the cases).

To better quantify the results a new experiment was performed on a small dataset (cluster with 2 million entries). The map phase required approximately 10 seconds to complete. In the first case, where no performance improving technique is used, the total execution time was 42 seconds (that means that the reduce phase took 32 seconds).

Using the mbrs technique the total execution time dropped to 40 seconds, meaning that the reduce phase completed in 2 seconds less, or a 6.25% improvement.

Using the dynamic grid, the total execution time dropped further, to 37 seconds, meaning the reduce phase completed in 5 seconds less, or a 15.625% improvement.

Note that we can use both performance improvement techniques but that would require running two different preprocessing jobs on the dataset, one to determine the dynamic grid and one to determine the mbrs.

3.8. Emulator

For the needs of this thesis we also produced a java based emulator that mimics the execution of the map-reduce jobs described at the previous sessions. In contrast with the Hadoop cluster, the emulator runs on a single machine and executes the reduce jobs sequentially. Its benefits are:

- easy debugging of the source code for the three described algorithms
- more readable output of the map-reduce job using apache POI library to write the results to an excel file.

As expected, the emulator does not support processing datasets the size the Hadoop cluster does. This is due to the memory constraints imposed by running on a single machine. Also, the execution time is greatly increased since each reduce task is processed sequentially.

Below are some figures that show the detailed output of the emulator for a run of the *pSPQ* algorithm:

Table 4: Emulator output 1/4

Reducer Id	#of data elements	#of feature elements	#of feature elements from duplication	sum of feature elements (column C + column D)	#of feature elements that were evaluated	time taken
load for reducer #0:	11410	85072	0	85072	85072	40,495
load for reducer #3:	809	569	0	569	569	0,035
load for reducer #5:	329	246	57	303	303	0,004
load for reducer #6:	34631	26037	5033	31070	31070	4,516

In table 4 we can see the detailed results for each of the reduce tasks. The emulator keeps track for the following parameters: the data objects each reducer received, the feature objects it received, how many of the feature objects come from duplication, the sum of feature objects that were received, the number of feature objects that were evaluated in order to produce the result and finally the time taken for this reducer to complete.

Table 5: Emulator output 2/4

Vectors initialized in:	1,116
Map phase completed in:	1,875
Reducer input sorted in:	0,117
Reducer phase completed in:	103,879
Job completed in:	108,056
Total memory used during job execution (mb):	937

Table 5 shows some statistics for a query run on the emulator. The times to complete each step of the job are recorded, along with the total memory requirements. One interesting thing to note is that the intermediate step of sorting the reducer input, which is mandatory for all the algorithms, only amounts to a small fraction of the total processing time. This is due to the use of the highly optimized sorting function that Java includes.

In table 6 we can see some more detailed results for each of the reducers. This data is stored on a new sheet on the output file produced by the emulator. It documents (for each reducer) its best calculated score, its id, the feature object that gave that best score and its features. That helps a lot with the verification process.

Table 6: Emulator output 3/4

Reducer id:	Best score:	Data object id:	Feature object id:	Features of f.o :
#0:	#1: 0.0833	#:974	#:15	kv16|kv2|kv10|kv4|kv11|kv15 |kv6|kv13|kv1|kv17
#0:	#2: 0.0833	#:1012	#:15	kv16|kv2|kv10|kv4|kv11|kv15 |kv6|kv13|kv1|kv17
#0:	#3: 0.0833	#:1211	#:15	kv16|kv2|kv10|kv4|kv11|kv15 |kv6|kv13|kv1|kv17
#0:	#4: 0.0833	#:1579	#:15	kv16|kv2|kv10|kv4|kv11|kv15 |kv6|kv13|kv1|kv17
#0:	#5: 0.0833	#:1851	#:15	kv16|kv2|kv10|kv4|kv11|kv15 |kv6|kv13|kv1|kv17
#0:	#6: 0.0833	#:2141	#:15	kv16|kv2|kv10|kv4|kv11|kv15 |kv6|kv13|kv1|kv17
#0:	#7: 0.0833	#:2264	#:15	kv16|kv2|kv10|kv4|kv11|kv15 |kv6|kv13|kv1|kv17
#0:	#8: 0.0833	#:2408	#:15	kv16|kv2|kv10|kv4|kv11|kv15 |kv6|kv13|kv1|kv17
#0:	#9: 0.0833	#:2698	#:15	kv16|kv2|kv10|kv4|kv11|kv15 |kv6|kv13|kv1|kv17
#0:	#10: 0.0833	#:2821	#:15	kv16|kv2|kv10|kv4|kv11|kv15 |kv6|kv13|kv1|kv17

Finally in table 7 we can see the top-*k* results. The emulator creates a new sheet on the output file and there it stores the top-*k* results for the job. That is achieved by comparing the results each reducer produced and keeping the *k* best ones.

Table 7: Emulator output 4/4

Reducer id:	Best score:	Data object id:	Feature object that gave best score:
#0	0.0833	#:974	#:15
#0	0.0833	#:1012	#:15
#0	0.0833	#:1211	#:15
#0	0.0833	#:1579	#:15
#0	0.0833	#:1851	#:15
#0	0.0833	#:2141	#:15
#0	0.0833	#:2264	#:15
#0	0.0833	#:2408	#:15

CHAPTER 4. CONCLUSIONS – FURTHER WORK

4.1. Conclusions
4.2. Further Work

4.1. Conclusions

In this thesis we studied the problem of parallel/distributed processing of spatial preference queries using keywords.

We proposed scalable algorithms that rely on grid-based re-partitioning of input data in order to generate partitions that can be processed independently in parallel.

To boost the performance of query processing, we showed that it is possible to employ early termination, thus reporting the correct result set after examining only a handful of the input data.

Our experimental study using both real and synthetic datasets shows that our best algorithm consistently outperforms other implementations, whilst its performance is not significantly affected even in the case of demanding queries.

Moreover, we examined additional possible improvements i.e dynamic grid partitioning and the use of mbrs and showed that those two methods, despite requiring an extra preprocessing step, offer quantified improvement gains over the static grid partitioning.

4.2. Further work

One important contribution of this thesis is that the mechanism that deals with the datasets is reusable. If one has to deal with data that belongs to the “big-data” category (or in other words hundreds of gigabytes), he can then adopt the partitioning mechanism as described. By splitting one large problem into sub-problems, he can then design an algorithm based on the problem/query he has to solve.

The queries examined in this thesis used the non-spatial score as the only parameter that defines the quality of a feature object/point of interest. In real life applications it makes sense to use another score function, the *influence score* [71].

The influence score is a combination of both the quality of a feature object and its distance to the data objects. In other words, a traveler would likely be interested in renting a hotel room that has some quality restaurants/cafes nearby that are not necessarily the best ones available, just to save the trouble of getting to the best ones.

Such a query would then involve a second parameter that directly impacts the score of each feature object, its distance to the candidate data objects.

Formally we could write: $= a * w(f,q) + (1-a)dist_{score}(f,q)$, where $\alpha \in [0,1]$, is a balancing parameter that defines which factor will have a greater influence on the score: the textual similarity or the distance between the two objects and $dist_{score}(f,q)$ is the spatial score that is defined as: $dist_{score} = \frac{maxDatasetDimension}{dist(f,q)}$

The greater the distance between two points (*f*,*q*), the lower the $dist_{score}$ is, so the $InfluenceScore$ between f,g is reduced. The problem with this definition is that the $dist_{score}$ is by definition a continuous variable.

That poses a challenge if we are to implement algorithms with early termination that would use the *InfluenceScore,* since a feature object might have a really low Jaccard score, but being extremely close to a data object would give it a high overall score.

To tackle this issue we propose using a quantified $dist_{score}$. This means that the total possible values of $dist_{score}$ are already known and defined. For instance let's assume we have 3 values (representing "close", "average distance" and "far") for the distance between f,q. Each possible distance $dist(f, q)$ would then be assigned a quantified $dist_{score}$ with the following check:

- If $dist(f, q)$ < MaxDistance/3 then $dist_{score} = 1$
- If MaxDistance/3 < $dist(f, q)$ < 2*MaxDistance/3 then $dist_{score} = 0.5$
- If $dist(f, q)$ > MaxDistance/3 then $dist_{score} = 0.1$

Obviously the values 1, 0.5, 0.1 are not important and can be chosen arbitrarily. What is important is that the lowest possible distance category gets the highest score and the highest category gets the lowest score. Also the total possible distance categories can be as many as we like.

Using this technique, we would be able to modify our early termination algorithms to work with this query. We would use the same strategy of sending the feature objects to the reducers sorted by their Jaccard score, and during the influence score calculation we would calculate the distance as before, but we would also retrieve the $dist_{score}$ that corresponds to the calculated distance. The early termination criterion would be altered to the product of the best poi's Jaccard score with the best $dist_{score}$. Taking the poi with the best score and drawing the circle with radius MaxDistance/3, we then check if a data object resides in that area. If so we have an optimum result.

If this quantification of the $dist_{score}$ is acceptable, we have a technique to implement the best possible algorithm, based on our early termination work.

REFERENCES

[1] M. L. Yiu, X. Dai, N. Mamoulis, and M. Vaitis, “Top-k Spatial Preference Queries,” in ICDE, 2007.

[2] N. Bruno, L. Gravano, and A. Marian, “Evaluating Top-k Queries over Web-accessible Databases,” in ICDE, 2002.

[3] A. Guttman, “R-Trees: A Dynamic Index Structure for Spatial Searching,” in SIGMOD, 1984.

[4] G. R. Hjaltason and H. Samet, “Distance Browsing in Spatial Databases,” TODS, vol. 24(2), pp. 265–318, 1999.

[5] R. Weber, H.-J. Schek, and S. Blott, “A quantitative analysis and performance study for similarity-search methods in highdimensional spaces.” in VLDB, 1998.

[6] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, “When is “nearest neighbor” meaningful?” in ICDT, 1999.

[7] R. Fagin, A. Lotem, and M. Naor, “Optimal Aggregation Algorithms for Middleware,” in PODS, 2001.

[8] I. F. Ilyas, W. G. Aref, and A. Elmagarmid, “Supporting Top-k Join Queries in Relational Databases,” in VLDB, 2003.

[9] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung, “Efficient Top-k Aggregation of Ranked Inputs,” ACM TODS, vol. 32, no. 3, p. 19, 2007.

[10] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao, “Efficient OLAP Operations in Spatial Data Warehouses,” in SSTD, 2001.

[11] S. Hong, B. Moon, and S. Lee, “Efficient Execution of Range Top-k Queries in Aggregate R-Trees,” IEICE Transactions, vol. 88-D, no. 11, pp. 2544–2554, 2005.

[12] T. Xia, D. Zhang, E. Kanoulas, and Y. Du, “On Computing Top-t Most Influential Spatial Sites,” in VLDB, 2005.

[13] Y. Du, D. Zhang, and T. Xia, “The Optimal-Location Query,” in SSTD, 2005.

[14] D. Zhang, Y. Du, T. Xia, and Y. Tao, “Progessive Computation of The Min-Dist Optimal-Location Query,” in VLDB, 2006.

[15] Y. Chen and J. M. Patel, “Efficient Evaluation of All-Nearest-Neighbor Queries,” in ICDE, 2007.

[16] P. G. Yokesh Kumar, Ravi Janardan, “Efficient Algorithms for Reverse Proximity Query Problems,” in ACM GIS, 2008.

[17] M. L. Yiu, P. Karras, and N. Mamoulis, “Ring-Constrained Join: Deriving Fair Middleman Locations from Pointsets via a Geometric Constraint,” in EDBT, 2008.

[18] M. L. Yiu, N. Mamoulis, and P. Karras, “Common Influence Join: A Natural Join Operation for Spatial Pointsets,” in ICDE, 2008.

[19] Y.-Y. Chen, T. Suel, and A. Markowetz, “Efficient Query Processing in Geographic Web Search Engines,” in SIGMOD, 2006.

[20] V. S. Sengar, T. Joshi, J. Joy, S. Prakash, and K. Toyama, “Robust Location Search from Text Queries,” in ACM GIS, 2007.

[21] S. Berchtold, C. Boehm, D. Keim, and H. Kriegel, “A Cost Model for Nearest Neighbor Search in High-Dimensional Data Space,” in PODS, 1997.

[22] E. Dellis, B. Seeger, and A. Vlachou, “Nearest Neighbor Search on Vertically Partitioned High-Dimensional Data,” in DaWaK, 2005, pp. 243–253.

[23] N. Mamoulis and D. Papadias, “Multiway Spatial Joins,” TODS, vol. 26(4), pp. 424–475, 2001.

[24] A. Hinneburg and D. A. Keim, “An Efficient Approach to Clustering in Large Multimedia Databases with Noise,” in KDD, 1998.

[25] E. Amitay, N. Har’El, R. Sivan, and A. Soffer. Web-a-where: geotagging web content. In SIGIR, 2004.

[26] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In VLDB, 2006.

[27] J. Ballesteros, A. Cary, and N. Rishe. Spsjoin: parallel spatial similarity joins. In GIS, pages 481–484, 2011.

[28] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In WWW, 2007.

[29] C. B¨ohm, B. Braunm¨uller, F. Krebs, and H.-P. Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In SIGMOD Conference, 2001.

[30] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In SIGMOD Conference, 1993.

[31] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. PVLDB, 3(1):373–384, 2010.

[32] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In SIGMOD Conference, 2011.

[33] E. P. F. Chan. Buffer queries. IEEE Trans. Knowl. Data Eng., 15(4):895–910, 2003.

[34] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In ICDE, 2006.

[35] Y.-Y. Chen, T. Suel, and A. Markowetz.Efficient query processing in geographic web search engines. In SIGMOD Conference, 2006.

[36] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. PVLDB, 2(1):337–348, 2009.

[37] J. Ding, L. Gravano, and N. Shivakumar. Computing geographical scopes of web resources. In VLDB, 2000.

[38] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. IEEE Trans. Knowl. Data Eng., 19(1):1–16, 2007.

[39] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In ICDE, 2008.

[40] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In VLDB, 2001.

[41] A. Guttman. R-trees: A dynamic index structure for spatial searching. In SIGMOD Conference, 1984.

[42] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (sk) queries in geographic information retrieval (gir) systems. In SSDBM, 2007.

[43] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. ACM Trans. Database Syst., 24(2):265–318, 1999.

[44] Z. Li, K. C. K. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang. Ir-tree: An efficient index for geographic document search. IEEE Trans. Knowl. Data Eng., 23(4):585–599, 2011.

[45] M. D. Lieberman, H. Samet, and J. Sankaranarayanan. Geotagging with local lexicons to build indexes for textually-specified spatial data. In ICDE, 2010.

[46] J. A. Orenstein. Spatial query processing in an object-oriented database system. In SIGMOD, 1986.

[47] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In SSTD, 2001.

[48] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørv°ag. Efficient processing of top-k spatial keyword queries. In SSTD, 2011.

[49] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In SIGMOD Conference, 2004.

[50] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. In SSTD, 2005.

[51] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In ICDE, 2009.

[52] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. ACM Trans. Database Syst., 36(3):15, 2011.

[53] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In ICDE,2009.

[54] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In CIKM, 2005.

[55] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. ACM Trans. Database Syst., 23(4):453–490, 1998.

[56] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *PVLDB*, 6(1):1–12, 2012.

[57] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *Proc. Of SIGMOD*, pages 373–384, 2011.

[58] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[59] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of OSDI*, 2004.

[60] K. Deng, X. Li, J. Lu, and X. Zhou. Best keyword cover search. *IEEE Trans. Knowl. Data Eng.*, 27(1):61–73, 2015.

[61] C. Doulkeridis and K. Nørv°ag. A survey of analytical query processing in MapReduce. *VLDB Journal*, 2014.

[62] Y. Du, D. Zhang, and T. Xia. The optimal-location query. In *Proc. of SSTD*, pages 163–180, 2005.

[63] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. In *Proc. of ICDE*, pages 1352–1363, 2015.

[64] H. Gupta, B. Chawda, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. K. Mohania. Processing multi-way spatial joins on MapReduce. In *Proc. Of EDBT*, pages 113–124, 2013.

[65] H. Hu, G. Li, Z. Bao, J. Feng, Y. Wu, Z. Gong, and Y. Xu. Top-k spatio-textual similarity join. *IEEE Trans. Knowl. Data Eng.*, 28(2):551–565, 2016.

[66] J. Rao, J. J. Lin, and H. Samet. Partitioning strategies for spatio-textual similarity join. In *Proc. Of BigSpatial Workshop*, pages 40–49, 2014.

[67] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørv°ag. Efficient processing of top-k spatial preference queries. *PVLDB*, 4(2):93–104, 2010.

[68] G. Tsatsanifos and A. Vlachou. On processing top-k spatio-textual preference queries. In *Proc. of EDBT*, pages 433–444, 2015.

[69] T. Xia, D. Zhang, E. Kanoulas, and Y. Du. On computing top-t most influential spatial sites. In *Proc. of VLDB*, pages 946–957, 2005.

[70] M. L. Yiu, X. Dai, N. Mamoulis, and M. Vaitis. Top-k spatial preference queries. In *Proc. of ICDE*, pages1076–1085, 2007.

[71] M. L. Yiu, H. Lu, N. Mamoulis, and M. Vaitis. Ranking spatial data by quality preferences. *IEEE Trans. Knowl. Data Eng.*, 23(3):433–446, 2011.

[72] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. SJMR: parallelizing spatial join with MapReduce on clusters. In *Proc. of CLUSTER*, pages 1–8, 2009.

[73] Y. Zhang, Y. Ma, and X. Meng. Efficient spatio-textual similarity join using MapReduce. In *Proc. of IEEE Web Intelligence*, pages 52–59, 2014.

SHORT VITA

Mpestas Dimitrios was born in Romania in 1983. He moved do Greece in 1985. In 2001 he received his high school diploma. He continued his studies at the University of the Aegean where he received his Diploma in Computer Science in 2009. In 2010 he enlisted to the Greek army and fulfilled his military obligations. In 2011 he was accepted as a post-graduate student at the University of the Aegean, where he studied computer networks and technologies. In 2013 he received his Master from the University of the Aegean. In 2014 he got accepted as a post-graduate student at the University of Ioannina. Since 2015 he has worked as a java developer in two E.U co-funded projects (roadRunner at the University of Piraeus, economicLinkedOpenData at the University of Piraeus). In February 2017 he presented his thesis in order to complete the requirements for the Master’s degree set by the University of Ioannina.