Indexing, Query Processing, and Clustering of Spatio-Temporal Text Objects

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Anders Skovsgaard
July 31, 2014
Abstract

With the increasing mobile use of the web from geo-positioned devices, the Internet is increasingly acquiring a spatial aspect, with still more types of content being geo-tagged. As a result of this development, a wide range of location-aware queries and applications have emerged. The large amounts of data available coupled with the increasing number of location-aware queries calls for efficient indexing and query processing techniques.

This dissertation investigates how to manage geo-tagged text content to support these workloads in three specific areas: (i) grouping of spatio-textual objects, (ii) spatio-temporal aggregates, and (iii) spatio-textual region querying without special purpose index structures.

First, two novel techniques to perform grouping of spatio-textual objects are presented. In the first technique, top-$k$ groups of objects are returned while taking into account aspects such as group density, distance to the query, and relevance to the query keywords. The nodes of an R-tree are extended with compressed histograms that represent the objects contained in their sub-tree. Results of empirical studies show that the approach is viable in practical settings. In the second technique, the grouping of spatio-textual objects is done without considering query locations, and a clustering approach is proposed that takes into account both the spatial and textual attributes of the objects. The technique expands clusters based on a proposed quality function that enables clusters of arbitrary shape and density. Empirical studies show that the approach is effective at discovering real-world points of interest.

Second, an extension of static frequent item counting techniques is proposed to enable the processing of vocabularies that change considerably over time. The proposed techniques adaptively maintain the most frequent items with exact counts rather than approximations at varying spatial and temporal granularities to support top-$k$ spatio-temporal term queries. Studies show that the proposed techniques excel under update and query-intensive loads.

Finally, this dissertation investigates a technique to perform spatio-textual region queries without the use of special purpose index structures. Spatio-textual objects are encoded into bit strings with a spatial and textual part that may be indexed using any standard DBMS. A query processing algorithm is proposed that provides an exact top-$k$ result by merging partial results. The results shows excellent indexing and query execution performance on a standard DBMS.
Resumé


Afhandlingen vedrører effektiv håndtering af geo-tagget indhold inden for tre områder: (i) gruppering af spatio-tekstuelle objekter, (ii) spatio-temporal aggregering samt (iii) spatio-tekstuelle område forespørgsler uden brug af specielle indiceringsteknikker.


Dernæst foreslås en udvidelse af såkaldte ”frequent item counting” opsummeringer, så disse kan håndtere ordforråd, der ændres betragteligt over tid. De foreslåede teknikker er tilpasningsdygtige og kan vedligeholde de hyggelige elementer med nøjagtige optællinger i stedet for kun tilnærmelser. Der vedligeholdes optællinger for forskellige spatiale og temporale granulariteter for at kunne levere de top-k mest benyttede ord. Resultaterne viser, at de foreslåede teknikker er velegnede, når der er en intensiv belastning med opdateringer og forespørgsler.

Slutteligt præsenteres teknikker til at udføre spatio-tekstuelle forespørgsler uden brug af specielle indiceringsteknikker. Spatio-tekstuelle objekter repræsenteres som bit-strenge med en spatial og en tekstuel del, der kan indiceres af et standard DBMS. Der foreslås en algoritme, der er i stand til at levere top-k resultater ved at kombinere del resultater. Teknikkerne giver fremragende indicering- og forespørgsels-udførelsestider på et standard DBMS.
Acknowledgments

I would like to thank the people who helped and supported me during my Ph.D. studies. First and foremost, I want to thank my advisor Christian S. Jensen for providing me with advices, comments, and guidance.

I appreciate the many good discussions and fruitful collaborations with Darius Sidlauskas. Also, thanks to Kenneth Sejdenfaden for the work done together.

I want to thank the present and former members of the Data-Intensive Systems Group and MADALGO for providing a motivating, fun, and positive working environment.

Thanks to Junghoo Cho for support and guidance during my stay abroad at the University of California, Los Angeles. Also, thanks to the people in the UCLA Web Mining Lab for the kind welcome, their help, and the many chats.

I appreciated the meetings with Anders Møller and Olav W. Bertelsen who gave me useful comments and guidance.

Lastly, thanks to my supportive, encouraging, and patient girlfriend Christine.

Anders Skovsgaard
Aarhus, July 31, 2014
# Contents

Abstract i  
Resumé iii  
Acknowledgments v  
Contents vii

I Overview 1

1 Introduction 3  
1.1 Location-Aware Grouping of Spatio-Textual Objects 4  
1.2 Clustering of Spatio-Textual Objects 8  
1.3 Spatio-Temporal Aggregates 11  
1.4 Spatio-Textual Region Querying Without Special Purpose Index Structures 14  
1.5 Dissertation Structure 16

II Publications 19

2 Finding Top-k Relevant Groups of Spatial Web Objects 21  
2.1 Introduction 21  
2.2 Problem Design 24  
2.3 Proposed Solution 28  
2.4 Experimental Evaluation 44  
2.5 Related Work 53  
2.6 Conclusions and Future Work 54

3 GroupFinder: A New Approach to Top-K Point-of-Interest Retrieval 57  
3.1 Introduction 58  
3.2 Preliminaries 59
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3 System Architecture</td>
<td>59</td>
</tr>
<tr>
<td>3.4 The GroupFinder Web Service</td>
<td>63</td>
</tr>
<tr>
<td>3.5 Demonstration</td>
<td>66</td>
</tr>
<tr>
<td>4 A Clustering Approach to the Discovery of Points of Interest from Geo-Tagged Microblog Posts</td>
<td>69</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>69</td>
</tr>
<tr>
<td>4.2 Quality Based Clustering</td>
<td>72</td>
</tr>
<tr>
<td>4.3 Proposed Solution</td>
<td>77</td>
</tr>
<tr>
<td>4.4 Experimental Evaluation</td>
<td>84</td>
</tr>
<tr>
<td>4.5 Related Work</td>
<td>92</td>
</tr>
<tr>
<td>4.6 Conclusions and Future Work</td>
<td>93</td>
</tr>
<tr>
<td>5 Scalable Top-k Spatio-Temporal Term Querying</td>
<td>95</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>95</td>
</tr>
<tr>
<td>5.2 Preliminaries and Related Work</td>
<td>97</td>
</tr>
<tr>
<td>5.3 Proposed Solution</td>
<td>101</td>
</tr>
<tr>
<td>5.4 Experimental Evaluation</td>
<td>114</td>
</tr>
<tr>
<td>5.5 Conclusions</td>
<td>122</td>
</tr>
<tr>
<td>6 Scalable Spatio-Textual Region Querying in Key-Value Stores</td>
<td>125</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>125</td>
</tr>
<tr>
<td>6.2 Related Work</td>
<td>127</td>
</tr>
<tr>
<td>6.3 Preliminaries</td>
<td>128</td>
</tr>
<tr>
<td>6.4 Proposed Solution</td>
<td>130</td>
</tr>
<tr>
<td>6.5 Experimental Evaluation</td>
<td>141</td>
</tr>
<tr>
<td>6.6 Conclusions</td>
<td>150</td>
</tr>
<tr>
<td>Bibliography</td>
<td>153</td>
</tr>
</tbody>
</table>
Part I

Overview
Chapter 1

Introduction

The digital universe is currently expanding exponentially, and the amount of text content available on the web, including news items, web pages, and microblog posts, grows rapidly. Today, everyone can generate such content. In particular, microblogging services such as Twitter, Facebook, and Foursquare make it easy for all to contribute. Coupled with the increasing mobile use of the web from geo-positioned devices such as smartphones, the information available on the Internet is increasingly acquiring a spatial aspect, with still more types of content being geo-tagged.

The collection of the large amounts of geo-tagged text objects enables a wide range of location-aware services. The data may be used by map providers and microblogging services to provide users with more relevant and up-to-date information. A large number of new queries and indexing techniques targeting this type of data have emerged. Existing techniques for processing spatial objects cannot be applied directly to support queries that consider attributes such as text and time. Rather, the rapidly increasing amounts of data requires new techniques for storage, indexing, and query processing. Multiple new index structures have been proposed to enable efficient retrieval of spatio-textual objects. They are either R-tree based [16, 23, 27, 41, 53, 76, 90, 91, 98], grid based [51, 87], or space-filling-curve based [20, 22]. The efficient retrieval of spatio-textual objects has been studied in the context of a number of new queries. These studies utilize the new index structures and propose novel query processing algorithms to efficiently compute the targeted queries. The proposals either support Boolean queries, or they support top-$k$ queries with a ranking function.

Simultaneously, new data mining proposals have emerged that leverage the available data. These proposals include, event detection [67, 68, 78], news extraction [80], recommendation [40] and decision making [11] systems. The rapidly growing amounts of data also contains temporal information which is considered in recent work proposing aggregation R-trees [63, 74].

Motivated by the new content available on the web and the increasing
mobile use of the web, this dissertation studies scalable techniques for the indexing, query processing, and clustering of spatio-temporal text objects. Specifically, we study three subjects: (i) grouping of spatio-textual objects, (ii) spatio-temporal aggregates, and (iii) spatio-textual region querying without special purpose index structures.

We group, or cluster, spatio-textual objects in two separate proposals. The first forms the top-\(k\) most relevant groups that are in the vicinity of a query location. The relevancy of a group is calculated as a combination of textual relevancy and spatial proximity. An existing spatio-textual indexing technique is extended with histograms to enable efficient processing of the query. The next approach forms clusters of spatio-textual objects without considering the query location. A novel quality based clustering solution is proposed that considers textual descriptions. It is based on the reciprocal nearest neighbor algorithm, and the result is a set of points of interest, where each point of interest corresponds to a cluster.

The proposal for spatio-temporal aggregation can efficiently process top-\(k\) spatio-temporal term queries. It involve improving existing techniques for frequent item counting. The existing techniques require a fixed number of counters, which may be difficult to set with large amounts of data. It is improved to support a dynamic number of counters that vary with the data covered by spatio-temporal grid cells.

Much recent research has been done to enable the efficient processing of queries against spatio-textual objects. Multiple special index structures have been proposed to answer queries performed by users of geo-positioned devices such as smartphones. In contrast, we suggest to use techniques already available in any standard DBMS to answer such queries. We propose an approach that involves encoding spatio-textual objects into a bit string. A novel query processing algorithm provides an exact result using a standard DBMS.

The remainder of this chapter is structured as follows. Section 1.1 describes our two techniques for the grouping of spatio-textual objects. Our work on spatio-temporal aggregation is covered in Section 1.3. Section 1.4 describes how we perform spatio-textual region querying without special index structures. Finally, Section 1.5 covers the structure of the remaining chapters of the dissertation.

1.1 Location-Aware Grouping of Spatio-Textual Objects

Motivation and Background

Information on the Internet is increasingly acquiring a spatial aspect, with still more types of content being geo-tagged, such as points of interest, status updates from social-network users, and web pages [3, 29, 69]. With the
1.1. GROUPING OF SPATIO-TEXTUAL OBJECTS

increasing mobile use of the web from geo-positioned devices such as smartphones, it has become relevant to explore new kinds of spatial web queries that take into account both a user location and user-supplied keywords when returning geo-tagged web content. Today, many of the queries performed by users already concern local objects. Google reports that 20% of all searches have local intent, while Microsoft finds that 53% of all mobile searches have local intent.

Many existing studies propose novel index structures to enable efficient computation of spatial keyword queries that return a list of \( k \) individual objects \([14, 23, 27, 58, 76, 90, 98, 99]\). In these queries, each result object by itself satisfies the query. Thus, the user has to choose which single object to visit. This object may not be relevant to the user and the remaining objects in the result set may be relatively far from the chosen. This makes it a potentially tedious exercise to explore multiple objects.

Other studies return an aggregate result instead of single single objects \([15, 61, 95, 97]\), namely a result consisting of a single set of objects that collectively cover the query keywords. With this approach, a user may find a number of objects in close proximity of each other that together satisfy the query keywords. The result set is minimal such that if a single object can satisfy the query, it is returned.

The existing works all provide the user with single objects or a minimal set. Users may wish to explore co-located nearby places that match the supplied query keywords. This browsing behavior typically occurs when people do shopping or when tourists visit new cities. The shopper may know the type of item of interest, e.g., shoes, but not the specific place to find such items. Most likely, the shopper will browse a number of shoe shops before deciding which shoes to buy. A tourist visiting a new city may not know any interesting places to eat. Being able to browse a number of places of interest is important to get an impression of the possible options.

**Our Proposed Solution**

We propose a new type of query that targets the type of situation where a user wishes to explore co-located nearby places that match supplied query keywords. This query returns \( k \) groups of objects where the objects in a group are close to each other and to the query location, and where each object is relevant to the query keywords.

Given a set of spatial web objects with spatial coordinates and text descriptions, the proposed top-\( k \) groups spatial keyword query takes a location and a set of query keywords as arguments and returns \( k \) groups of objects such that the objects in a group are close to each other, the group is close to

\[\text{1https://sites.google.com/a/pressatgoogle.com/google places/metrics}\]
\[\text{2http://searchengineland.com/microsoft-53-percent-of-mobile-searches-have-local-intent-55556}\]
CHAPTER 1. INTRODUCTION

the query location, and the objects in the group are textually relevant to the query keywords. To balance the spatial and textual relevancy, we introduce a parameter $\alpha$. Setting $\alpha$ close to 1 means that groups in close proximity of the query location and with small inter-objects distances are preferred; and with $\alpha$ close to 0, textual relevance is given priority. To further be able to control the importance of the spatial distance to a query result versus the spatial diameter, we introduce another parameter $\beta$. Setting $\beta$ close to 0 favors dense groups, whereas setting it close to 1 assigns importance instead to the distance between the query and the group.

We propose a solution that follows the same idea as the spatially inverted index \[76\] with a list of terms that maps to a modified R-tree. The proposed index, the Group Extended R-tree (GER-tree), is an R-tree where all node entries maintain compressed histograms that represent the objects contained in their subtree, as illustrated in Figure 1.1. On the one hand, the compressed histogram must “describe” all objects in the subtree to enable pruning. On the other hand, it must be compact to enable a large node fanout.

A compressed histogram has entries of the form $(\text{minDist}, n_d, \text{avg} tf)$, where distance $\text{minDist}$ and count $n_d$ state a density property of the objects in the subtree that the entry refers to. Specifically, they state that no set of objects exists in the subtree with more than $n_d$ objects such that all pairs of objects in the set are no further apart than $\text{minDist}$. The example in Figure 1.1 shows a node $R_4$ with three subtrees. The entry of each subtree in $R_4$ has a compressed histogram. For example, the entry for subtree $R_1$ states that 2 objects are within the distance of 1.4 and have an average term frequency of 1.0.

![Figure 1.1: Example of a GER-Tree](image)

The compressed histograms are used by a proposed query processing algorithm to prune the search space while providing an exact result. To process single-keyword queries, we propose a query processing algorithm that uses best-first traversal of the GER-tree. Since the resulting groups may each be distributed across several leaf nodes, the path to the result may include several
1.1. GROUPING OF SPATIO-TEXTUAL OBJECTS

non-leaf nodes. Therefore, when calculating group relevance, all combinations of non-leaf entries should be considered. For example, the most relevant group may be formed by combining $R_1$ and $R_2$ in Figure [1.1]. By using the values in the compressed histogram, a best case relevance can be computed without fetching the leaf level objects. Thus, parts of the tree may be pruned, which makes the query processing algorithm more efficient. The processing of multiple keyword queries employs the same approach as the single-keyword query processing algorithm. We maintain a GER-tree per term, so in order to perform a multiple-keyword query, we work with several GER-trees simultaneously.

We propose a number of optimizations and evaluate them separately. Inter-object distances are used in the compressed histograms and are maintained for each object insertion. The first optimization stores the inter-object distances in the GER-tree. The second optimization addresses the problem of small fanout by allowing non-leaf nodes to share leaf nodes. The final optimization allows the insertion algorithm to perform delayed compressed histogram maintenance such that the compressed histograms are only created once all objects have been inserted. This approach may be applied to static datasets.

Further, we propose a demonstration of the proposed solution that includes a web-based service that take into account the user’s current location and user-supplied keywords. Predefined transportation modes captures how the user intends to travel to a group. Thereby, the user does not have to set any parameter values.

Contributions

- We propose a novel type of query that exploits the locations of objects to return aggregate results, namely groups of relevant spatial web objects.
- We introduce the idea of compressed histograms to efficiently provide a best-case group relevancy without fetching the actual objects.
- A new indexing technique is given that uses the compressed histograms.
- Query processing algorithms are proposed for queries with single and multiple keywords. They exploit the proposed compressed histograms to prune the search space.
- We present an extensive empirical performance study of the proposed techniques that suggests that these are applicable in realistic, practical settings.
- We give a proof-of-concept demonstration of the proposed solution using a web-based service and a real-world dataset.
Experimental Findings

The experiments are performed on real data obtained from Google Places. We study the efficiency of our proposed solution and its optimized versions when varying the size of the compressed histogram and the parameters $\alpha$ and $\beta$. The main findings are as follows:

- The insertion time increases as the size of the compressed histogram decreases because of the greater fanout.
- The numbers of objects in the groups increases when $\alpha$ approaches 0 since the spatial dimension becomes less influential.
- The runtime increases for small values of $\alpha$ due to the extended spatial search range.
- The diameter of result groups increases as expected with the value of $\beta$.
- The query runtime increases with the number of keywords and $k$.
- For reasonable settings of the parameters, the query runtime is less than 1 second for top-3 results.
- The optimizations improve the query runtime and insertion time for static datasets.

Future Work

There are a number of promising future research directions. First, it may be of interest to consider the use of other types of indexes such as quad-trees or the IR-tree. Second, the compressed histograms may be used to help speed up evaluations of clustering algorithms since information about inter-object distances and other attributes are maintained. Third, the processing of multiple-keyword queries may be optimized with a strategy similar to that of using composite indexes.

1.2 Clustering of Spatio-Textual Objects

Motivation and Background

Increasingly large volumes of geo-tagged content are becoming available. Specifically, users of GPS-enabled mobile devices are creating large amounts of geotagged microblog posts describing their daily activities. Some users may report information relating to their current location, while others in the same region may report information unrelated to their location. Intuitively, users are likely to use specific textual descriptions more often in some regions than
1.2. CLUSTERING OF SPATIO-TEXTUAL OBJECTS

in others. For example, posts mentioning “hotel” might be more frequent in
a region containing a hotel than in a region containing a supermarket.

Many existing studies have explored research directions related to mi-
icroblogs. This includes indexing of microblogs \cite{10, 18, 93, 95}, event detection
from microblogs \cite{67, 68, 78}, news extraction from microblogs \cite{80}, microblog
post based recommendation \cite{40} and decision making \cite{11} systems, microblog
ranking \cite{30, 86}, visualization \cite{66}, and spatio-temporal aggregation \cite{81}. Spa-
tial data mining is a popular research topic, where the most related work in-
cludes density-based clustering algorithms \cite{31, 44, 50, 54, 79}. However, the
existing mining techniques fail to consider the textual descriptions of objects.

Points of Interest (PoIs) represent geographical entities (e.g., restaurants,
museums, hotels) as geo-referenced points with attached semantics (e.g., names
or addresses). PoIs are important for a wide variety of services, e.g., Google /
Yahoo / Bing Maps, Yelp, TripAdvisor. Also, they have high value for compa-
nies like Factual and Google, that sell PoIs. PoIs may be collected in different
ways that each has its shortcomings. Manual PoI collection and maintenance
is expensive and affordable only to big corporations. It may be possible for
PoI owners to contribute information themselves; however, it is likely that PoI
owners only contribute to the most popular services and only if it is beneficial
to them.

Our Proposed Solution

We propose a new clustering approach that considers both the spatial and
the textual description of objects. Combined with the increasing availability
of geo-tagged microblog posts, we enable a new way of obtaining PoIs. A set
of spatio-textual objects has a spatial shape that may be represented by any
enclosing technique such as a convex hull, an orthogonal convex hull, or a
minimum bounding rectangle. This shape may be of arbitrary size and may
overlap other shapes. We propose a quality function that evaluates the quality
of a set of spatio-textual objects. The quality function considers all objects
inside an enclosing shape independent of their textual description. This is
illustrated in Figure 1.2 where the object symbols represent the textual de-
scriptions and the objects are enclosed by a convex hull. The quality function
provides a maximum score for sets of objects that all share a given textual
description, termed an annotation. This annotation is used as a description
of the cluster. Sets of objects that include objects inside their enclosing shape
that do not contain the annotation in their textual description result in a lower
quality score. For example, the sample dataset in Figure 1.2a has a higher
score than the sample dataset in Figure 1.2b.

A new clustering technique is proposed that employs the quality function
as a criterion for expansion. The technique is based on reciprocal nearest
neighbor expansion. Therefore, it does not require a range parameter that
may result in over-expansion or failure to detect sparse clusters. The cluster-
Figure 1.2: Sample Datasets where Shapes Represent Different Textual Descriptions

The clustering technique continuously finds the reciprocal nearest neighbor and merges objects or clusters. The merging depends on the score and a proposed quality threshold. If the merging of two clusters provides a score that exceeds the quality threshold, the clusters are merged. When the merging fails to meet the quality threshold, the expansion is stopped, and the existing clusters are returned as a result.

**Contributions**

- We contribute a new type of clustering method that takes into account the textual descriptions of the objects.
- We propose an extension of the reciprocal nearest neighbor algorithm that incorporates a quality-based clustering threshold.
- A formal definition of the quality function along with a proof of the correctness of the algorithm are given.
- We give an extensive empirical study using a real-word dataset that indicates that the proposed solution is able to successfully identify most existing PoIs from an abundance of geo-tagged messages.

**Experimental Findings**

Experiments are performed on real data obtained from Twitter, and the PoIs found are compared with places obtained via the Google Places API. The coverage and properties of the clustering algorithm are studied when varying the quality threshold. Also, the proposed solution is compared with the existing clustering algorithm DBSCAN. The study results in the following findings:
1.3. SPATIO-TEMPORAL AGGREGATES

- The clustering approach identifies 80-85% of the PoI that Google Places describes.

- The spatial extents of the clusters and the number of clusters discovered decrease as the quality threshold increases since less noise is tolerated.

- The annotations of the clusters contain multiple terms for all quality threshold values, making them highly descriptive.

- The comparison with DBSCAN shows that DBSCAN over-expands clusters and finds noisy clusters, while the propose solution is very accurate for all the studied annotations.

Future Work

The work opens to a number of interesting research directions. First, the clustering approach may be extended to consider the temporal attribute of the objects. Thereby, temporal meta-data of the PoIs may be inferred such as opening hours. Second, more advanced natural-language processing techniques may be applied to produce better annotations. Finally, the clustering algorithm may be extended to support dynamic datasets such that new objects may be inserted and existing ones removed.

1.3 Spatio-Temporal Aggregates

Motivation and Background

Users of social media services often generate similar content in response to events that catch their attention. For example, when a natural disaster occurs, multiple users are likely to report on this independently. Some users may discuss evacuations and the traffic situation, while other users may simultaneously discuss unrelated topics such as food recipes or a sporting event.

Several recent works target such social media streams. The existing solutions support the discovery and tracking of real-world entities (stories, events, etc.) [4, 5], the monitoring of most popular keywords (“trends”) [65], and the monitoring of temporal and/or spatial bursts [67].

The monitoring of most popular keywords is useful to track current trends and events. However, the querying of the past is not supported by existing works. Therefore, it is not possible to find the “word in the street” or “talk of the town” during some time period of interest.

Index structures, e.g., based on the R-tree, have been proposed to support spatio-temporal queries. However, these are not well suited for streams with rapid content and spatio-temporal aggregate queries [89]. Also, an aggregation R-tree have been proposed for spatio-temporal queries [63, 74]. It augments
each node with an aggregate value (e.g., count) and works only for non-holistic aggregation functions.

Other existing work is capable of counting frequent items in streams with rapid content \[28, 49, 63, 70, 71\]. These techniques require a fixed number of counters to be set. However, the textual properties of the vast amounts of available data vary both spatially and temporally, making it difficult or impossible to choose an appropriate, fixed number of counters.

Our Proposed Solution

We propose to extend recent work by supporting queries that retrieve the top-\(k\) most frequent terms of content within user-specified spatio-temporal regions. Specifically, we design a spatio-temporal grid index that maintains frequent item counting summaries. The existing work for static summaries is extended to support dynamic, variable-sized summaries so it can adapt to streams of objects that have varying textual descriptions.

We employ a static grid-based approach that uses uniform grids with pre-defined and fixed cell sizes. By introducing multiple layers of the grid, we partition space at multiple granularities. To accommodate the temporal extent of queries, the spatial grid is extended with a temporal dimension. The finest temporal granularity is the smallest query-enabled time interval, e.g., an hour. We create a new instance of every spatial grid cell for every new such time interval. Each spatial and temporal granularity maintains a summary that contains precomputed information about the most frequent terms in each grid cell. With this approach, the querying of summaries can be supported efficiently for spatial regions and temporal ranges of any size larger than the finest-granularity cell.

Each finest-granularity cell covers a small part of the world, and different cells may receive very different amounts of objects. To store, for each cell, each term in list, e.g., ordered by frequency, requires linear space and is infeasible for large-scale processing. Instead, we keep track only of the most frequent terms in each cell. This is achieved by extending the counter-based \textit{SpaceSaving} algorithm \[70\]. As a result, only a fraction of the terms is stored, while we maintain guarantees about the results of queries. The existing work on static summaries falls short when the vocabulary changes considerably. Instead, we propose a new technique that dynamically adapts the number of counters in a summary to accommodate the changes in the data. When a summary cannot accommodate a new object, we perform a so-called \textit{Aggressive Increment} of the number of counters in the summary. In this approach, we double the number of counters when needed. Thus, the summary can accommodate the new object while maintaining guarantees about query results. However, a situation may occur where guarantees cannot be maintained by performing \textit{Aggressive Increment}. In this case, we perform so-called \textit{checkpointing}, by which we prevent an inconsistent state. Checkpointing involves
1.3. SPATIO-TEMPORAL AGGREGATES

rendering the currently active summary inactive and replacing it with a new, active summary.

Since the vocabulary varies considerably, the dynamic summary should not only expand. Therefore, we perform a so-called *Relaxed Decrement* when the summary is larger than required. In this procedure, the number of counters is halved after any given time granule has passed.

A merging algorithm is provided to merge summaries across multiple cells and to merge the inactive and active summaries when the queries are processed. The guarantees of the top-$k$ result is maintained by the merging algorithm.

Contributions

- We provide techniques capable of supporting a holistic aggregate function: the top-$k$ most frequent terms. We propose a variant of frequent item counting that maintains the top-$k$ most frequent terms error-free in each summary.

- We give a merging algorithm to combine aggregates as needed to support arbitrary spatio-temporal regions with low probability of introducing errors, and it allows us to provide correctness guarantees.

- We provide techniques that maintain and use spatio-temporal aggregates that grow and shrink according to the activity in the incoming data stream.

- We integrate the techniques into a framework that supports spatio-temporal top-$k$ frequent term queries.

- We study the framework empirically using a real-world data stream.

Experimental Findings

For large-scale experiments, we employ an Amazon EC2 High-Memory Cluster backed by a dual Intel Xeon E5-2670 processor with 32 hardware threads and 244 GB of memory. We collected all geo-tagged messages from the public Twitter FireHose during May, 2013. The dataset contains 110,426,053 tweets. The main results are as follows:

- The comparisons with static summaries and hash tables shows that our proposal is more accurate while using less space.

- The throughput is always more than 13,000 objects per second.

- The number of checkpoints decreases quickly over time since the number of counters adjusts to the intensity of the cell.
• The index with summaries requires approximately between 70 and 150 GB of memory depending on the settings.

• The query runtime is often less than 1 millisecond.

Future Work

Some improvements of the work may involve the study of reducing the memory consumption. In the proposed solution, summaries may describe terms that also exists in other spatial and temporal granularities. While this yields efficient query processing, it could be of interest to study whether this information can be represented more compressed, thereby reducing the memory consumption.

1.4 Spatio-Textual Region Querying Without Special Purpose Index Structures

Motivation and Background

Maps-based services allow users to retrieve Points of Interest (PoI) near them. For example, users may look for hotels or restaurants in neighborhoods where they are currently present or are going to be at a later time. Typical maps-based functionality supports the retrieval of relevant PoIs that belong to the part of space visible on the user’s screen. This functionality supports users who wish to browse or explore their surroundings and is not limited to the retrieval of PoIs that are closest to an exact user location.

Several proposals exist that are capable of finding the most relevant places that are closest to an exact location. They are either R-tree based [16, 23, 27, 41, 58, 76, 90, 91, 98], grid based [51, 87], or space-filling-curve based [20, 22]. The proposals either support Boolean queries, or they support top-$k$ queries with a ranking function.

The existing works involve the use of a special purpose index structure to efficiently process the queries. However, this makes it difficult to benefit from existing DBMS technologies. Also, many of the works focus on finding the top-$k$ most relevant objects, that are closest to an exact location instead of providing the top-$k$ most relevant objects in a region (on a user’s screen).

Our Proposed Solution

We aim at supporting queries that returns the top-$k$ most relevant PoIs in a spatial region using any standard DBMS or key-value store. These queries are used by all of the major providers of maps-based services, including Google Maps, Bing Maps, and Yahoo! Maps.
1.4. **SPATIO-TEXTUAL REGION QUERYING**

We propose a novel approach to the storing and indexing of spatio-textual objects in any standard DBMS or key-value store. The objects are first encoded into a bit string with a spatial and a textual part. The spatial part is encoded using a grid index, while the textual part employs a hash function. Collisions may occur both spatially and textually since objects may be located in the same cells and textual descriptions may be hashed to the same value. We propose to store only a predefined number of object references for each spatio textual bit string. This reduces the storage requirements and the insertion time while preserving exact results. The bit strings are stored as the key while the list of references to matching objects is stored as the value. This enables storage of spatio textual objects in any DBMS or key-value store.

We present a query processing algorithm that provides the exact result to the spatio textual region query. The algorithm queries the spatio textual objects using the indexed bit strings. The query region may overlap several grid cells, which are merged while guaranteeing the exact result.

![Figure 1.3: Framework Overview with the Spatial and Textual Encoding (two left-most figures) and a Standard DBMS (right-most figure)](image)

An overview of the framework is given in Figure 1.3 where the data first is preprocessed such that each term associated with an object is handled with the proposed spatial and textual encoding techniques. The result is a bit string for each term associated with an object. The bit string is then stored and indexed in a standard DBMS using a standard indexing technique. The query processing algorithm preprocesses the query region and keyword in a similar manner such that the spatial and textual parts of a query are encoded into a bit string. This bit string is then used to perform a lookup in a standard DBMS that returns a number of objects. The grid structure may have to be searched at several levels in order to produce an exact result. Therefore, the bit string is refined and used to perform additional lookups. When enough objects necessary to answer the query have been retrieved, the result is returned.

**Contributions**

- We propose and give a precise definition of a query used by major providers of maps-based services that finds the top-k most relevant objects in a spatial region with respect to their textual descriptions.
• We propose a novel approach to the storing and indexing of spatio-textual objects in any standard DBMS or key-value store.

• We present an query processing algorithm that provides the exact result to the proposed query.

• We report on a extensive experimental study of the proposed techniques using real-world data.

Experimental Findings
The experiments are performed on commodity hardware with real data collected from Twitter. Two datasets are created: one that is dense and one that is sparse. The main findings are as follows:

• The query runtime is approximate 4 milliseconds for the dense dataset while the same type of queries take less than 1 millisecond for the sparse dataset.

• The storage time and space consumption increase with the number of granularities and the predefined number of objects to maintain.

• The number of cells to fetch increases with the value of $k$ and the predefined number of objects to maintain.

• On average, no more than 7 cells are required to produce a result for all the parameter settings studied.

Future Work
It is of interest to extend the work to support other types of spatio-textual queries without the requirement of a special purpose index structure. It may also be of interest to study spatio-textual nearest neighbor queries using the proposed solution.

1.5 Dissertation Structure
The remaining chapters of the dissertation correspond to self-contained papers. The papers are unedited except for formatting changes.

• Chapter 2 corresponds to the paper:
1.5. DISSERTATION STRUCTURE

- Chapter 3 corresponds to the paper:

- Chapter 4 corresponds to the paper:

- Chapter 5 corresponds to the paper:

- Chapter 6 corresponds to the paper:

The following survey paper was submitted and accepted during my Ph.D. studies but is not included in the dissertation.

Part II

Publications
Chapter 2

Finding Top-k Relevant Groups of Spatial Web Objects

Abstract

The web is increasingly being accessed from geo-positioned devices such as smartphones, and rapidly increasing volumes of web content is geo-tagged. In addition, studies show that a substantial fraction of all web queries have local intent. This development motivates the study of advanced spatial keyword based querying of web content. Previous research has primarily focused on the retrieval of the top-$k$ individual spatial web objects that best satisfy a query specifying a location and a set of keywords.

This paper proposes a new type of query functionality that returns top-$k$ groups of objects while taking into account aspects such as group density, distance to the query, and relevance to the query keywords. To enable efficient processing, novel indexing and query processing techniques for single and multiple keyword queries are proposed. Empirical performance studies with an implementation of the techniques and real data suggest that the proposals are viable in practical settings.

2.1 Introduction

Information on the Internet is increasingly acquiring a spatial aspect, with still more types of content being geo-tagged, such as points of interest, status updates from social-network users, and web pages [3, 29, 69]. Coupled with the increasing mobile use of the web from geo-positioned devices such as smartphones, it has become relevant to explore a new kind of spatial web query that takes into account both a user location and user-supplied keywords when returning geo-tagged web content, called spatial web objects.
Existing spatial keyword queries return a list of \( k \) individual objects, where each object in isolation satisfies the query \([14, 23, 27, 58, 76, 90, 98, 99]\). Only four studies in the literature are known to the authors that do not return single objects, but rather return an aggregate result \([15, 61, 96, 97]\), namely a result consisting of a single set of objects that collectively cover the query keywords. Google Maps can display all objects that match a given keyword. But as only locations are displayed on the map, users cannot easily assess the textual relevancy of the objects. The user also cannot take into account objects outside the map borders.

This paper considers a new type of query that targets the type of situation where a user wishes to explore co-located nearby places that match the supplied query keywords. This browsing behavior typically occurs when people do shopping or when tourists visit new cities. The shopper may know the type of item of interest, e.g., shoes, but not the specific place to find such items. Most likely, the shopper will browse a number of shoe shops before deciding which shoes to buy. A tourist visiting a new city may not know any interesting places to eat. Being able to browse a number of places of interest is important to get an impression of the possible options.

Today, many of the queries performed by users already concern local objects. Google reports that 20%\(^1\) of all searches have local intent, while Microsoft finds that 53% of all mobile searches have local intent\(^2\). With approximately 1 billion tourists per year (2010 numbers) spending more than 900 billion US dollars\(^3\), this new type of query is highly relevant.

To support the browsing behavior, the query returns \( k \) groups of objects where the objects in a group are close to each other and the query location, and where each object is relevant to the query keywords.

Figure 5.1a illustrates a real-world example where a tourist located at a hotel in San Francisco issues the query “restaurant” in preparation for exploring the neighborhood for a good place to eat. Consider the case where a list of \( k \) individual restaurants is returned. The top-1 restaurant may be closed, the top-2 restaurant may be too expensive, and the food may look too spicy at the top-3 restaurant; and these restaurants may be located relatively far apart. In the figure, the three restaurants (dots) nearest to the query (the cross) are indeed located in different directions from the query. This makes it a potentially tedious exercise to explore multiple restaurants before making a dining decision.

The query we propose identifies groups of places that are relevant to the query keywords such that the places in a group are near each other and such that the groups are near the query location. The query returns a list of \( k \) such

---

\(^1\)https://sites.google.com/a/pressatgoogle.com/google-places/metrics

\(^2\)http://searchengineland.com/microsoft-53-percent-of-mobile-searches-have-local-intent-55556

\(^3\)http://www.bbc.co.uk/schools/gcsebitesize/geography/tourism/tourism_trends_rev1.shtml
groups. With $k$ set to 3, the query in the example returns the three dense groups of restaurants seen in the lower left part of the figure. By visiting one or more of these groups, the tourist can conveniently explore multiple restaurants before making a decision.

We offer a precise definition of a carefully designed top-$k$ relevant groups spatial keyword query and then propose indexing and query processing techniques that aim to enable efficient computation of the query. In doing so, we build on existing state-of-the-art indexing techniques for spatio-textual data. This involves mapping from terms to an R-tree. Specifically, we extend a spatial inverted index with compressed histograms in each node entry in order to obtain compact descriptions of the objects contained in subtrees that then enable effective search space pruning.

We present an algorithm capable of building and maintaining the compressed histograms in the index structure, and we present query processing algorithms that are able to prune the search space by utilizing the compressed histograms. The algorithms work without spatial ranges that limit the search space and guarantee correct results that take into account the entire data set. They calculate the spatial distance to a group and use the compressed histograms to calculate the diameter and textual relevance of a group.

The proposed techniques are evaluated on real data collected from Google Places [36]. Three datasets are used: one that covers San Francisco, one that covers Las Vegas, and one that covers a synthetic city where objects from four major US cities have been shifted to the same region. We study different parameter settings and offer insight into the impact on the properties of the resulting groups along with the performance in terms of runtime and page accesses. The study suggests that the proposed solution is capable of finding relevant groups while maintaining good performance.

The contributions are as follows: (i) A novel type of query that exploits the locations of objects to return aggregate results, namely groups of relevant
spatial web objects; (ii) a new indexing technique for spatial web objects; (iii) query processing algorithms that exploit the indexing technique; and (iv) an empirical performance study of the proposed techniques that suggests that these are applicable in realistic, practical settings.

The rest of the paper is organized as follows. Section 5.2 formally defines the problem. Section 2.3 presents the new index structure and query processing algorithms along with two enhancements. The empirical study is covered in Section 6.5. Finally, Section 2.5 gives an overview of related work, and conclusions and future work are covered in Section 6.6.

2.2 Problem Design

Intuitively, given a set of spatial web objects with spatial coordinates and text descriptions, the top-\(k\) groups spatial keyword query takes a location and a set of query keywords as arguments and returns \(k\) groups of objects such that the objects in a group are close to each other, the group is close to the query location, and the objects in the group are textually relevant to the query keywords. A number of definitions are needed to formalize this setting and query.

The spatial web objects being queried are objects with a geographical point location and a textual description. One type of such objects is web pages for places such as restaurants, attractions, and public offices that may be given meaningful locations. Another type is listings of business in online directories such as Google Places.

We let \(D\) be the set of spatial web objects \(o\), where each object is a pair \((\lambda, \phi)\) such that \(o.\lambda\) is \(o\)'s location and \(o.\phi\) is \(o\)'s textual description. Specifically, we assume that the location is a point location in two-dimensional Euclidean space.

Similarly, a top-\(k\) groups query \(q\) takes a triple of arguments \((\lambda, \phi, k)\), where \(q.\lambda\) is a point location, \(q.\phi\) is a set of keywords, and \(q.k\) is the number of groups in the result. Intuitively, the former is the querying user’s location, and the latter is keywords entered by the user.

The next step is to define the spatial and textual relevance notions used in the query. First, we define the distance between a query location \(\lambda\) and a group \(G\) of objects as the distance between the query location and the nearest object in the group: 
\[
d(\lambda, G) = \min_{o \in G} ||\lambda, o.\lambda||,
\]
where \(||·, ·||\) denotes the Euclidian distance.

Intuitively, a group of relevant objects that are close to each other is more attractive than a group of equally relevant objects that are further apart. We define the spatial compactness of a group \(G\) as the largest inter-object distance between any two objects in the group: 
\[
diameter(G) = \max_{o_1, o_2 \in G} ||o_1, o_2||.
\]

Another natural definition might be the area of the convex hull of the location of the objects in a group. However, if the objects tend to be aligned...
along a straight line, it is possible for objects in a group with small area to be far apart. Our definition of diameter avoids such groups.

Numerous measures exist for evaluating textual relevance, and the techniques presented in this paper are applicable to many of these. In this paper, we employ the state-of-the-art language models developed by Ponte et al. [75], which are also used in related work by Cong et al. [23]. Here, a text document is represented by a vector where each dimension corresponds to a distinct term in the document. The relevance of an object $o$ to a query term, $t \in q.\phi$, is defined as follows:

$$TR(t, o.\phi) = (1 - \gamma) \frac{tf(t, o.\phi)}{|o.\phi|} + \gamma \frac{tf(t, Coll)}{|Coll|},$$

(2.1)

where $tf(t, o.\phi)$ is the number of occurrences of term $t$ in $o.\phi$, $tf(t, Coll)$ is the count of term $t$ in the collection $Coll$ of $D$, and $\gamma$ is a smoothing parameter.

The above equation defines the relevance of a single object to a query term. The following definition of group proximity extends the definition to apply to all terms in a query and groups of objects. The underlying idea is to give more weight to larger groups of relevant objects and to give more weight to groups with several objects for each query term.

$$GP(q.\phi, G) =$$

(2.2)

$$\prod_{t \in q.\phi} \left( \left( \sum_{o \in G^t} TR(t, o.\phi) \right) + 1 \right) \cdot |G^t|^{-1},$$

where $G^t$ is the objects in $G$ containing the term $t$. Note that the lower the score, the better the proximity.

The equation sums up the textual relevance of each object and adds 1 to ensure that the final result is a value between 0 and 1. The sum yields the total textual relevance of the group without taking into account the number of objects.

Next, we wish to give preference to groups where the query terms are distributed relatively evenly among objects (rather than a single term being contained in many objects and other terms being contained in few objects). This is achieved by multiplying the sum for each query terms by the number of objects involved—the more evenly terms are distributed, the larger this product becomes. An example is given in Example 1.

If a group contains no objects for some query term, the score is undefined because of division by zero. We will only compute score when this does not occur. Also, we note that the following proposals remain valid when using alternative definitions of group proximity. The definition given here provides good results on real datasets, to be shown in Section 2.4.

**Example 1.** We consider two datasets each with 12 objects, that each
CHAPTER 2.

contain one of three different terms, \( t_1 \), \( t_2 \), and \( t_3 \). An object containing term \( t_i \) is denoted \( O_{t_i}, i \in [1,3] \). As seen from Table 1, the group consisting of objects with terms that are evenly distributed has the best (i.e., lowest) score.

\[
\begin{array}{cccc}
O_{t_1} & O_{t_2} & O_{t_3} & GP(q, \phi, G) \\
10 & 1 & 1 & 0.041 \\
4 & 4 & 4 & 0.006 \\
\end{array}
\]

Table 2.1: Scores for Groups with Different Term Distributions

The score, or cost, \( Cost(q, G) \) of a group \( G \) with respect to a query \( q \) takes into account both the spatial and textual characteristics of the objects in the group.

We introduce a parameter \( \alpha \) that enables us to control the importance of the spatial properties versus the textual relevance. To further be able to control the importance of the spatial distance versus the diameter, we introduce a parameter \( \beta \). For example, this parameter can be used for the modeling of situations where a user wants to drive by car to reach a group, but then wants to visit the objects by foot. In this case, a relatively long distance to the group is acceptable, while the diameter should be small. Our prototype offers driving, bicycling, and walking settings. The 3 options can be selected both for travel to the groups and for the movement within them. This way, a user can specify different means of transportation, e.g., traveling by car to the group and then walking between the objects in the group. Thus, the users need not see parameters \( \alpha \) and \( \beta \).

As a result of these design considerations, we get the following ranking function for groups.

\[
Cost(q, G) = \alpha \beta d(q, \lambda, G) + (1 - \beta) \frac{diameter(G)}{maxD} + (1 - \alpha) GP(q, \phi, G),
\]

(2.3)

where \( maxD \) is the largest Euclidean distance the space may have.

Parameter \( \alpha \in [0, 1] \) enables balancing the spatial proximity and the group proximity. An \( \alpha \) close to 1 means that groups in close proximity and with small inter-objects distances are preferred, while with \( \alpha \) close to 0, textual relevance is given priority. In fact, \( \alpha = 0 \) gives the lowest cost to the group of all objects in \( D \) that contain at least one of the query keywords. Setting parameter \( \beta \in [0, 1] \) close to 0 favors dense groups, whereas setting it close to 1 assigns importance instead to the distance between the query and the group.

\[\text{http://cs.au.dk/~anderssk/groupfinder/}\]
With the above definitions in place, we can define the result of the top-k groups spatial keyword query.

**Definition 1. Top-k groups spatial keyword query** A top-k groups spatial keyword query \( q \) takes a set \( D \) of spatial web objects as an argument together with three parameters \((\lambda, \phi, k)\). It returns \( k \) subsets \( G_1, \ldots, G_k \) of \( D \) such that \( G_i \subseteq D_i, \ i = 1, \ldots, k \), where \( D_1 = D \) and \( D_i = D \setminus \bigcup_{j=1}^{i-1} G_j \), \( i = 2, \ldots, k \), and such that there does not exist a subset \( G \subseteq D_i \) for which \( \text{Cost}(q, G) < \text{Cost}(q, G_i), \ i = 1, \ldots, k \).

The definition ensures that the groups of objects in the results are disjoint. This avoids results where objects are repeated in multiple groups. The definition guarantees that when \( G_i \) is computed, no better group can be computed from the objects available.

**Example 2.** Figure 2.2 gives the locations of 8 spatial objects. We set \( \alpha = 0.4 \) and \( \beta = 0.4 \); and for simplicity, we assume that the textual description of each object only contains one single term. A number of different costs needed for computing top-k queries are presented in Table 2.2. For example, the cost for \( G = \{o_1\} \) is calculated as follows:

\[
\text{Cost}(q, G) = 0.4 \cdot \frac{0.4 \cdot 3 + (0.6) \cdot 0}{7} + 0.6 \cdot (1 + 1) \cdot 1^{-1} = 0.37
\]

The top-3 groups are \( G_1 = \{o_6, o_7, o_8\}, G_2 = \{o_4, o_5\}, \) and \( G_3 = \{o_1, o_2, o_3\} \). Notice that \( \{o_7, o_8\} \) has a lower cost than \( G_2 \) and \( G_3 \), but is excluded because it is a subset of \( G_1 \). Setting \( \beta = 0.2 \) changes the order of \( G_2 \) and \( G_3 \), while \( G_1 \) remains the top-1 result.

![Figure 2.2: Objects and MBRs](image)


2.3 Proposed Solution

Before presenting the proposed techniques, Section 2.3 covers two baseline algorithms. Section 2.3 describes the proposed index structure along with related definitions. Section 2.3 covers the compressed histograms used in the index, and index constructions is covered in Section 28. Finally, Section 17 presents the query processing algorithms for single and multiple keyword queries.

Baseline Approaches

While no algorithm exists for computing the top-k relevant groups of spatial web objects, we proceed to outline two baseline approaches based on different ideas. One performs a full enumeration, and the other adapts existing techniques for finding top-k single spatial web objects.

Baseline 1 (Full Enumeration): We create all possible subsets, \( G \subseteq D \), and calculate the cost of each. We then sort the subsets according to their scores and return the top-k most relevant, non-overlapping groups. However, the number of subsets and score computations are exponential in the dataset cardinality.

Baseline 2 (Index): For \( \alpha > 0 \) and \( \beta > 0 \), the groups are ranked according to both spatial proximity and textual relevance to the query. Thus, we can employ any of the existing spatio-textual index structures, e.g., the S2I index \[76\], to search for nearest neighbors.

We find the spatially nearest neighbor to the query location that matches one of the query keywords. Considering the object found as a starting object, we create all possible subsets, \( G \subseteq D \), that contains this starting object. For

<table>
<thead>
<tr>
<th>( G )</th>
<th>( d(q, \lambda, G) )</th>
<th>( \text{diameter}(G) )</th>
<th>( GP(q, \phi, G) )</th>
<th>( \text{Cost}(q, G) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( o_6, o_7, o_8 )</td>
<td>3.16</td>
<td>2.24</td>
<td>0.08</td>
<td>0.199</td>
</tr>
<tr>
<td>( o_7, o_8 )</td>
<td>3.16</td>
<td>1.0</td>
<td>0.17</td>
<td>0.207</td>
</tr>
<tr>
<td>( o_4, o_5 )</td>
<td>3.61</td>
<td>1.0</td>
<td>0.17</td>
<td>0.217</td>
</tr>
<tr>
<td>( o_1, o_2, o_3 )</td>
<td>3.0</td>
<td>3.16</td>
<td>0.08</td>
<td>0.227</td>
</tr>
<tr>
<td>( o_2, o_3 )</td>
<td>3.61</td>
<td>1.41</td>
<td>0.17</td>
<td>0.231</td>
</tr>
<tr>
<td>( o_1, o_2 )</td>
<td>3.0</td>
<td>2.0</td>
<td>0.17</td>
<td>0.24</td>
</tr>
<tr>
<td>( o_1, o_2, o_3, o_4 )</td>
<td>3.0</td>
<td>5.83</td>
<td>0.05</td>
<td>0.3</td>
</tr>
<tr>
<td>( o_1, o_6, o_7, o_8 )</td>
<td>3.0</td>
<td>6.32</td>
<td>0.05</td>
<td>0.32</td>
</tr>
<tr>
<td>( o_1 )</td>
<td>3.0</td>
<td>0.0</td>
<td>0.5</td>
<td>0.37</td>
</tr>
<tr>
<td>( o_8 )</td>
<td>3.16</td>
<td>0.0</td>
<td>0.5</td>
<td>0.37</td>
</tr>
<tr>
<td>( o_4 )</td>
<td>3.61</td>
<td>0.0</td>
<td>0.5</td>
<td>0.38</td>
</tr>
</tbody>
</table>

Table 2.2: Function Values
each of these subsets we calculate the cost.

We continue to find the next nearest neighbor and calculate the costs for the corresponding subsets. We do this until the distance to the nearest neighbor object becomes so large that no matter which objects in the dataset are added to the group, the group’s cost cannot be competitive. In doing this, we must assume that all inter-object distances for the group are 0. This conservative assumption is needed to ensure that the best possible cost is assumed.

The first baseline is inefficient because it materializes the entire search space of solutions and computes the cost of every possible result group. We thus disregard this solution. The second baseline has to search most parts of the dataset and will also not be considered in the paper’s empirical study. Instead, different variants of the proposed solution will be used for comparison.

**Group Extended R-Tree**

We proceed to present a new indexing technique that makes it possible to compute an exact query result while still being able to prune the search space substantially.

We map each term to a new index structure, the GER-tree, that extends the R-tree with compressed histograms.

Overall, the index consist three elements. A *vocabulary* maps each term to a GER-tree; a GER-tree exists for each term that indexes the objects that contain the term; and each entry in a non-leaf GER-tree node contains a *compressed histogram*, representing the child subtree.

Table 2.3 gives an example of the mapping from a term to a GER-tree. Parts used to compute the second term of Equation 2.1 along with the total number of objects are maintained together with the mapping to enable fast lookup.

| term | \( tf(t, Coll) \) | \( |Coll| \) | \( |D| \) | pointer |
|------|-----------------|-----------|---------|--------|
| bar  | 452             | 721       | 421     | GER\(_{bar}\) |
| hotel| 295             | 642       | 354     | GER\(_{hotel}\) |
| pizza| 124             | 524       | 245     | GER\(_{pizza}\) |
| shoes| 117             | 236       | 189     | GER\(_{shoes}\) |

Table 2.3: Mapping From Term to GER-Tree

Each tree used by the indexing technique resembles an R-tree. Leaf node entries are of the form \( (mbr, cp, tf(t, o,φ)|o,φ|) \), where \( mbr \) is the minimum bounding rectangle of the object and \( cp \) is the pointer to the objects in the database \( D \). The object information needed to calculate the textual relevance is also maintained in the leaf
node entry. Non-leaf node entries are of the form \((\text{histogram}, \text{mbr}, \text{cp})\), where \text{histogram} is a compressed histogram that represents the objects contained in the subtree. On the one hand, the histogram must “describe” all objects in the subtree to enable pruning. On the other hand, it must be compact to enable a large node fanout. The histograms are covered in detail shortly.

The \text{mbr} encloses all objects in the subtree, as in a standard R-tree. The child pointer, \(\text{cp}\), points to the root of the subtree. Figure 2.3 shows the GER-tree created from the objects in Figure 2.2. For simplicity, each object contains only one single term.

A compressed histogram has entries of the form \((\text{minDist}, n_d, \text{avg}\_\text{tf})\), where distance \text{minDist} and count \(n_d\) state a density property of the objects in the subtree that the entry refers to. Specifically, they state that no set of objects exists with more than \(n_d\) objects such that all pairs of objects in the set are no further apart than \text{minDist}. Put differently, for the set of objects, \(S\), contained in a given subtree the following holds for \text{minDist}:

\[
\forall O \subseteq S \ (|O| > n_d \Rightarrow \text{diameter}(O) > \text{minDist})
\]

Consider Figure 2.3. The first entry in the histogram that describes \(R_1\) states that any set with 2 or more objects has a diameter that exceeds 1.4. Likewise, the second entry states that any set with 3 or more objects has a diameter that exceeds 3.2. This entry then describes all objects in the subtree.

This information is used to prune the search space during query processing. According to the definition, exaggerations of the compactness are allowed. For example, an entry for \(R_1\) that states that any set with 5 or more than objects has a diameter that exceeds 3.2 is implied by the second entry and is correct. Similarly, an entry that states that any set with 3 or more objects has a diameter that exceeds 2.0 also is an exaggeration of the second entry and is correct. However, such entries provide lower pruning power. In contrast, an entry saying that any set with 2 or more objects has a diameter that exceeds 3.2 can lead to incorrect pruning and is wrong.
2.3. PROPOSED SOLUTION

The textual descriptions of the \( n_d \) objects that an entry states exist are not available at non-leaf levels. However, Equation 2.1 relies on information about both the term occurrences in the objects and in the complete collection. Thus, to be able to calculate the textual relevance when processing a query, the average value in the first term in Equation 2.1, \( \frac{tf(t, o.\phi)}{|o.\phi|} \), is stored in \( \text{avg}_t \). This is illustrated in Figure 2.3, where all objects have average textual relevance 1.0.

The average textual relevance of the objects referred to in a compressed histogram entry is calculated by the following function:

\[
TR_{\text{entry}}^a(t, e) = (1 - \gamma)e.\text{avg}_t + \gamma \frac{tf(t, \text{Coll})}{|\text{Coll}|},
\]

where \( t \) is a term and \( e \) is an entry in a compressed histogram. The collection information is available in the vocabulary as seen in Table 2.3. Having the average textual relevance, the group proximity for an entry for a specific term can be inferred from Equation 2.2 as shown below. We use the average textual relevance instead of the objects’ actual textual relevance. This allows us to propagate information up through the non-leaf nodes in the tree.

\[
GP_{\text{entry}}(t, e) = \left( \left( TR_{\text{entry}}^a(t, e) \cdot e.n_d \right) + 1 \right) \cdot e.n_d^{-1},
\]

where \( t \) is a term and \( e \) is a compressed histogram entry.

**Theorem 1.** Given a single term \( t \) and a compressed histogram entry \( e \) that describes a set of objects \( G \) the following holds:

\[
GP(t, G) = GP_{\text{entry}}(t, e)
\]

**Proof.** Since \( e.\text{avg}_t \) holds the average value, of the first term in Equation 2.1, we can compute the per-object average textual relevance using Equation 2.4. Thus, we can calculate the total textual relevance of the objects in \( G \):

\[
\sum_{o \in G} TR(t, o.\phi) = TR_{\text{entry}}^a(t, e) \iff \sum_{o \in G} TR(t, o.\phi) = TR_{\text{entry}}^a(t, e) \cdot e.n_d
\]

Substituting this into Equation 2.2 we get:

\[
\left( \left( \sum_{o \in G} TR(t, o.\phi) + 1 \right) \cdot |G^t| \right)^{-1} = \left( \left( TR_{\text{entry}}^a(t, e) \cdot e.n_d + 1 \right) \cdot e.n_d \right)^{-1}
\]

This completes the proof. \( \square \)
CHAPTER 2.

Building Compressed Histograms

So far, we have shown how the compressed histograms can be used in query processing. It is intuitively clear that the specific choices of \( \text{minDist} \) values to be used in the entries in a histogram affect the pruning power of the histogram. If the values are not chosen with care, several entries may cover almost the same objects, which wastes pruning power. Thus, we proceed to detail means of carefully selecting \( \text{minDist} \) values.

First of all, to guarantee correct query results, a histogram must describe all objects in its subtree. This is so because if objects exist that are not covered in the histogram, these are not taken into account during search, and the subtree can be pruned incorrectly.

We next consider how to select suitable \( \text{minDist} \) values that describe all objects in a leaf node. When processing a query, a compressed histogram with a good distribution of \( \text{minDist} \) values is important, since each query may form different candidate groups with different diameters and group proximity scores. The quality of a group is measured by the group proximity as defined in Equation 2.2. Therefore, a good distribution of \( \text{minDist} \) values should be according to the group proximity.

Adding an object to a small group has a bigger impact on the group’s proximity score than when a adding an object to a large group. If we were to select \( \text{minDist} \) values for entries according to the numbers of objects covered, the resulting histogram may have low pruning power since the group proximity may not be very different across different \( \text{minDist} \) values.

Instead, we select \( \text{minDist} \) values according to the group proximity. By choosing \( \text{minDist} \) values for entries based on group proximity, we ensure that the resulting compressed histogram is effective.

Compressed histogram construction is described in Algorithm 1.

First, all possible groups in the leaf node are calculated and inserted into a temporary histogram \( h_{\text{tmp}} \) with entries of the form \( (\text{minDist}, n_d, \text{GP}_{\text{entry}}, \text{avg}_{\text{tf}}) \), where \( n_d \) is the cardinality of a set \( O' \) of objects and \( \text{GP}_{\text{entry}} \) is the group proximity of the objects in \( O' \) to the query term, \( \text{term} \), of the tree. The entries in this histogram are a result of either (a) the grouping of leaf objects, (b) several compressed histogram entries being combined, or (c) the duplication of existing compressed histogram entries.

The temporary histogram for the objects in Figure 2.2 is shown in Table 2.4. For ease of understanding, sets are indicated that correspond to the \( n_d \) values. Irrelevant entries have been removed according to Line 4 in Algorithm 1.

The entry with lowest group proximity value is chosen if more entries exist with the same \( \text{minDist} \) value because of this filtering step. This is important since we always need to be conservative. For example, if we have two entries with the same \( \text{minDist} \) value, we must keep the one that provides the weakest pruning, i.e., the one that has the smallest group proximity value. If we do
Algorithm 1: BuildCompressedHistogram($term, O$)

1. $h_{tmp} \leftarrow$ new empty temporary histogram;
2. Calculate all possible groups and group proximity values and add them to $h_{tmp}$;
3. Order $h_{tmp}$ by $minDist$ ascendingly and group proximity value descendingly;
4. Remove entries from $h_{tmp}$ that do not provide lower group proximity values when comparing with the previous entry;
5. $h_r \leftarrow$ new empty compressed histogram;
6. $h_r$.Add($h_{tmp}$,peekFirst());
7. while $h_r$.Size() < $n$ do
   8. if $h_{tmp}$.Size() $\leq$ $h_r$.Size() then
      9. $h_r \leftarrow h_{tmp}$;
     10. break;
     11. mostDiscGP_entry $\leftarrow -\infty$;
     12. nextCandidate $\leftarrow$ empty;
     13. foreach Entry $e_{tmp}$ in $h_{tmp}$ do
        14. if $e_{tmp} \in h_r$ then
           15. continue;
        16. diff $\leftarrow e_{tmp}.GP_{entry} - h_{tmp}.peekNext().GP_{entry}$;
        17. if diff $> mostDiscGP_{entry}$ then
           18. mostDiscGP_{entry} $\leftarrow$ diff;
           19. nextCandidate $\leftarrow e_{tmp}$;
        20. $h_r$.Add(nextCandidate);
     21. foreach Entry $e$ in $h_r$ do
        22. foreach Entry $e_{tmp}$ in $h_{tmp}$ do
           23. if $e_{tmp}.minDist < h_r.peekNext().minDist \wedge e_{tmp}.GP_{entry} < GP_{entry}(term, e)$ then
              24. $e.n_d \leftarrow e_{tmp}.n_d$;
              25. $e.avg_{if} \leftarrow e_{tmp}.avg_{if}$;
           26. $h_r$.last.$n_d \leftarrow h_{tmp}.last.n_d$;
        27. $h_r$.last.$avg_{if} \leftarrow h_{tmp}.last.avg_{if}$;
     28. return $h_r$;

not discard the entries with higher group proximity, we do not describe the best case for the subtree, which may in turn yield incorrect pruning.

The next step is to select some of the entries from the temporary histogram for use in the size-constrained, compressed histogram, $h_r$. In doing so, we wish to select a set of entries that provides high pruning power.
Initially, the entry from $h_{\text{tmp}}$ with the smallest $\text{minDist}$ value is moved to the result histogram. This entry is important because if it is excluded, it is necessary to create an entry with $\text{minDist} = 0$, which adversely affects the pruning.

Then, starting in Line 13, we find the entry with the group proximity value, $\text{GP}_{\text{entry}}$, that differs the most from that of the following entry. For example, the second entry in Table 2.4 is chosen because $0.08 - 0.05 = 0.03$ is the largest difference in the group proximity. Note that the first entry has already been selected.

The only information available when later making a pruning decision is the subset of entries in the compressed histogram. Therefore, it is important that the entries describe the objects in the subtree correctly. In particular, when omitting an entry, a higher number of objects must be assumed to exist in the entry located before it in the temporary histogram.

For example, if we were to omit the third entry in the resulting compressed histogram in Table 2.5, we are missing information about a candidate group. In response to this, we adjust the number of objects, $n_d$, and average textual relevance, $\text{avg}_{\text{tf}}$, described by the second entry. The search algorithm could assume $n_d$ for this entry to be 8, since this is the $n_d$ value for the fourth entry, which is the next entry assuming that the third entry is omitted. While

| $\text{minDist}$ | $n_d = |O|$ | $\text{GP}_{\text{entry}}$ | $\text{avg}_{\text{tf}}$ |
|------------------|-----------------|-----------------|-----------------|
| 1.0              | $\{o_7, o_8\}$ | 0.17            | 1.0             |
| 2.24             | $\{o_7, o_6, o_8\}$ | 0.08 | 1.0 |
| 5.83             | $\{o_2, o_4, o_3, o_1\}$ | 0.05 | 1.0 |
| 6.32             | $\{o_7, o_6, o_4, o_8, o_1\}$ | 0.03 | 1.0 |
| 6.71             | $\{o_2, o_4, o_3, o_8, o_1, o_5\}$ | 0.02 | 1.0 |
| 7.21             | $\{o_7, o_2, o_6, o_1, o_3, o_8, o_1, o_5\}$ | 0.01 | 1.0 |

Table 2.4: Temporary Histogram of Leaf Objects

Table 2.5: Compressed Histogram

this assumption is safe, the pruning power is unnecessarily low. Instead, we replace the $n_d$ and $\text{avg}_{\text{tf}}$ of the second entry with the values from the omitted entry. By using these values of $n_d$ and $\text{avg}_{\text{tf}}$, we describe the best case for the subtree. This occurs starting in Line 21. In the example, we set the $n_d$ value
2.3. PROPOSED SOLUTION

of the second entry to 4 if we remove the third entry.

Notice that adding a relevant object to a group will always result in a lower (i.e., better) group proximity score according to Equation \ref{eq:group_proximity} independently of the textual relevance score of the object. Thus, we keep the best case information in order to prune correctly when searching.

The entries in the resulting compressed histogram in Table \ref{table:compressed_histogram} are selected from the temporary histogram in Table \ref{table:temporary_histogram} assuming that the compressed histogram can hold four entries. The first entry was added in Line 6. Next, the temporary histogram is iterated, and the second temporary histogram entry is added to the compressed histogram because it has the largest difference in group proximity value when compared with the third entry. This is done in Lines 7–20. This loop runs until the compressed histogram is full. The next two entries are added in the same manner. Finally, in Lines 26–27, the last entry in the compressed histogram is assigned all objects in the subset in order to provide the best case information.

Building the Index

Insertion of objects occurs generally as in the R-tree. The new aspect is that of updating the compressed histograms in the entries during insertions. The algorithm that accomplishes this takes two arguments, namely the MBR and text of the object to be inserted, is shown in Algorithm \ref{alg:insertion}. The operations ChooseLeaf and Split are those of the R-tree \cite{R-tree}.

When adding a new object to a leaf, the histograms in the parent non-leaf node entries are updated using the strategy described above. To have correct information in the complete tree, the parent nodes up to the root also have to be updated. This occurs in Lines 16 and 20 in Algorithm \ref{alg:insertion}. The compression approach is similar to the one described in Algorithm \ref{alg:compression} starting from Line 3.

Combining Entries

When propagating information up through non-leaf nodes, not only the information propagated from the leaf level is considered. Also, the compressed histograms in the non-leaf node entries are combined into new temporary histogram entries while taking into account the Euclidean distances between the MBRs of the subtrees. This ensures that the minDist values of the combined entries are consistent with the actual distances between the underlying objects.

Thus, the information about the combined non-leaf nodes are included in a temporary histogram so that accurate information can be used when creating the compressed histogram entries. Throughout, it must be ensured that the pruning is conservative. Therefore, the objects that correspond to the object count in a compressed histogram entry are assumed to be located on the MBR’s boundary nearest to the entry.

Example 3. If we consider Figures \ref{fig:2.2} and \ref{fig:2.3} and focus on the first
histogram entry for \( R_1 \) and the single histogram entry for \( R_2 \), we can see the shortest distance between the two MBRs of the two subtrees is 4.0 (this follows from Figure 2.2). The entry covering \( R_1 \) states that all groups with more than 2 objects have a diameter higher than 1.4. Likewise, the entry covering \( R_2 \) states that all groups with more than 2 objects have diameter higher than 1.

From this information, it is safe to compose a temporary combined entry with \( n_d = 4 \) and \( \text{minDist} = 4.0 \), which states that all groups with more than 4 objects have a diameter that exceeds 4.0. The group proximity value of the combined entry can be calculated using Equation 2.5. However, this entry is temporary and cannot be directly applied to the compressed histogram.

The second histogram entry for \( R_1 \) can be combined with the single entry describing \( R_2 \) to obtain a temporary entry with \( n_d = 5 \) and \( \text{minDist} = 4.0 \). This entry has the same \( \text{minDist} \) value as the previous temporary entry. Thus, when building the compressed histogram, in a similar manner as in Algorithm 1 starting from Line 3, the previous entry is discarded since it does not describe the best case for the subtree.

Figure 2.4 provides an overview of the best combinations of compressed histogram entries for the three subtrees. The first row describes the best case combination of the entries describing \( R_1 \) and \( R_2 \). The second row describes the best case for \( R_1 \) and \( R_3 \), the third row provides the best case for \( R_2 \) and \( R_3 \), and the fourth row provides the best case when considering all three subtrees.

\[ \begin{array}{ccc|c|c|c} \hline R_1 & Entries & R_2 & \text{Combined} \\ 3.2 & 3 & 1.0 & 1 & 2 & 1.0 & 4.0 & 5 & 1.0 \\ 3.2 & 3 & 1.0 & 2.2 & 3 & 1.0 & 3.6 & 6 & 1.0 \\ 1 & 2 & 1.0 & 2.2 & 3 & 1.0 & 5.0 & 5 & 1.0 \\ 3.2 & 3 & 1.0 & 1 & 2 & 1.0 & 2.2 & 3 & 1.0 & 5.0 & 8 & 1.0 \\ \hline \end{array} \]

Figure 2.4: Combining Compressed Histogram Entries

If two compressed histogram entries both have a larger \( \text{minDist} \) value than the distance between the MBRs, the \( \text{minDist} \) value of the combined histogram entry can be set to the smallest of the two \( \text{minDists} \). This is possible since the covered objects cannot be within a distance smaller than this distance. The objects with a distance smaller than this are described by a previous histogram entry and will therefore still be considered when creating the resulting compressed histogram. Since we do not describe single objects in the histograms, we will always assume a single object on the border MBRs. This is important in order to prune correctly.
2.3. PROPOSED SOLUTION

The approach described previously for controlling the size of a compressed histogram is used. Equation 2.5 is used to calculate the group proximity value having only the compressed histogram entries. Small histograms are faster to maintain and yield tree nodes with larger fanout. However, pruning benefits from large histograms. The experimental study covers the impact of different histogram sizes.

Updates

The objects we index are stationary and change infrequently, which is why we focus on query processing performance. However, updates can be handled by conservative deletions followed by insertions. When an object is deleted, the information in compressed histograms remains correct since it is conservative. However, some pruning power is generally lost. When inserting an updated object, the relevant histograms are updated to also describe this new object.
Query Processing

In query processing we need to be able to calculate a lower bound of the cost using the compressed histograms. We thus present a lower bound, $\text{MinCost}(q, e_E)$, that is the minimum spatio-textual group distance between the query $q$ and a single entry $e_E$ of an extended compressed histogram. This bound is used to prune the search space when querying.

Extended Compressed Histograms

An extended compressed histogram provides more information about the histogram entry. For each query term, the corresponding number of objects, $n_d$, and value of $\text{avg}_{tf}$ is stored. Several node entries may combine to form an intermediate result since a candidate group’s objects may be located in different MBRs. Therefore, the extended histogram includes a set $M$ that describes the MBRs that contain the objects.

Thus, entries in an extended compressed histogram are of the form $(\text{minDist}, M, \{(t, n_d, \text{avg}_{tf})_1, \ldots, (t, n_d, \text{avg}_{tf})_{q,\phi}\})$, where $t$ is a term, $n_d$ is a number of objects, and $\text{avg}_{tf}$ is an average textual relevance.

We define the lower bound cost function where the extended compressed histogram is used:

$$\text{MinCost}(q, e_E) = \alpha \left( \beta D_{qM}(q,\lambda, e_E.M) + (1 - \beta) e_E.\text{minDist} \right)$$

$$+ (1 - \alpha) \prod_{t \in q,\phi} GP(t, e_E^t),$$

where $q$ is the query and $e_E$ is an extended compressed histogram entry. Next, $e_E^t$ contains the same information as the normal histogram entry, but only for the query term $t$, and $GP(t, e)$ is defined in Equation 2.5. The smallest distance from the query location $q,\lambda$ to any MBR, $m$, in the set $M$ is defined by $D_{qM}(q,\lambda, M) = \min_{mbr \in M} \text{mindist}(q,\lambda, mbr)$, where $\text{mindist}$ returns the minimal distance between a point and a rectangle.

**Theorem 2.** Given a query $q$ and an extended compressed histogram entry $e_E$ that covers a set of objects $G$, the following holds:

$$\text{MinCost}(q, e_E) \leq \text{Cost}(q, G)$$

**Proof.** Since the set of MBR’s, $e_E.M$ encloses all objects in $G$, the minimum distance between $q,\lambda$ and the nearest MBR is no larger than the distance from $q,\lambda$ to any object in $G$:

$$D_{qM}(q,\lambda, M) \leq ||q,\lambda, o||, o \in G$$
According to the definition of minDist, the diameter of the objects in $G$ is at least $e_E \cdot \text{minDist}$. Thus, we have:

$$e_E \cdot \text{minDist} \leq \text{diameter}(G)$$

By Theorem 1 we have $GP(t, G) = GP(t, e)$ for single terms. Thus, with the extended compressed histogram entry, we have the following for multiple terms:

$$\prod_{t \in q, \phi} GP(t, e^t_E) = GP(q, \phi, G)$$

This completes the proof.

**Single Keyword Query Processing**

Single keyword queries are processed by means of best-first tree traversal. Since the objects in each result group can be distributed across several leaf nodes, the paths followed to obtain the result may include several non-leaf nodes. Therefore, when calculating the lower bound cost, all combinations of non-leaf node entries should be considered. A priority queue $U$ that maintains the order of the sets of nodes is used to keep track of candidate groups. Note that the candidate groups can contain both leaf and non-leaf nodes. Initially, the priority queue holds only the root node as seen from Line 2 in Algorithm 3.

The nodes in the first set are dequeued in Line 3 in Algorithm 4. In Line 6, all possible combinations of entries in the set of nodes are calculated. Recall that leaf node entries maintain information that enables calculation of the textual relevance so that a histogram entry can easily be created. Also, we always assume a single object is located on the MBR border. The possible outcomes are limited because of the minDist value since all objects within the same diameter per definition can be grouped together. This is due to Equation 2.2 that gives lower scores to groups with more relevant objects. This greatly reduces the set of combinations. An example of the reduction is provided in Example 4.

The approach described in Section 28 is used to create the combined entries. The possible combinations of entries are stored in an extended histogram entry.

Finally, the lower bound costs of the extended histograms are calculated in Lines 7–8, and the sets of nodes are added to the priority queue. From Line 10, when all nodes in $N$ are leaf nodes, we calculate the actual costs, $Cost(q, G)$, for all object combinations. The possible combinations can be reduced as stated above; again, an example is provided in Example 4.

**Example 4.** In Figure 2.2 the combinations \{o_1, o_4\}, \{o_1, o_3, o_4\}, \{o_1, o_2, o_4\} and \{o_1, o_2, o_3, o_4\} all have diameter 5.83 and the distance from the query to the group of objects is 3 in all cases. Therefore, it is
sufficient to include only the combination of \{o_1, o_2, o_3, o_4\}. The last combination has the lowest group proximity value, thus making the other combinations unnecessary. The same applies when combining histogram entries. Note that single entries, e.g., \{o_1\}, \{o_2\}, etc. are unique combinations and are always considered.

As seen from Line 11, if the cost is lower than that of any other candidate set of nodes in the priority queue, the group is added to the result list. Since an object can be in at most one group, the result list has to be examined before computing the cost of the following candidates, as seen from Line 10.

**Algorithm 3: SKSearch(index, q, k)**

1. \( U \leftarrow \) new min-priority queue;
2. \( U\text{.Enqueue}(\text{index.root}, 0); \)
3. \( \text{DoSearch(index, q, k, U);} \)

**Example 5.** Consider the objects and the query location in Figure 2.2 and the resulting GER-tree in Figure 2.3. We perform a top-1 query. Recall that for simplicity, all objects contain the same terms.

First, we enqueue the root node \( R_4 \) along with the best possible minimum cost according to Algorithm 3. Next, Algorithm 4 proceeds as follows. To simplify, we omit describing the extended histograms, but replace them with the identifiers of the corresponding nodes. Figure 2.3 describes the single entries and Figure 2.4 the combined ones. For example, combining the last entries describing \( R_1 \) and \( R_3 \) yields \( D_{qM}(q, \lambda, M) = 2.0 \) which follows from Figure 2.2. Thus, the lower bound cost can be calculated as:

\[
\text{MinCost}(q, e^{R_1, R_3}_E) = 0.4 \cdot \frac{0.4 \cdot 2.0 + 0.6 \cdot 3.6}{7} + 0.6 \cdot (((1 \cdot 6) + 1) \cdot 6)^{-1} = 0.184,
\]

where \( q \) is as given in Example 2 and \( e^{R_1, R_3}_E \) is the extended histogram created by combining the last compressed histogram entries describing \( R_1 \) and \( R_3 \).

1. Dequeue \( \{R_4\} \). Combine entries and calculate \( \text{MinCost} \) values.

Queue: \( 0.178 = \{R_3\}, 0.184 = \{R_1, R_3\}, 0.194 = \{R_1\}, 0.203 = \{R_1, R_2\}, 0.217 = \{R_2\}, 0.225 = \{R_1, R_2, R_3\} \)

Groups are always assumed to start on the nearest MBR border from the query location when calculating \( \text{MinCost} \) values.
2. Dequeue \( \{R_3\} \). Best cost: 0.199 (set \( \{o_6,o_7,o_8\} \)). Queue: \( \langle 0.184 = \{R_1,R_3\}, 0.194 = \{R_1\}, 0.203 = \{R_1,R_2\}, 0.217 = \{R_2\}, 0.225 = \{R_1,R_2,R_3\} \rangle \).

3. Dequeue \( \{R_1, R_3\} \). Best cost: 0.199 (\( \{o_6,o_7,o_8\} \)). Queue: \( \langle 0.194 = \{R_1\}, 0.203 = \{R_1,R_2\}, 0.217 = \{R_2\}, 0.225 = \{R_1,R_2,R_3\} \rangle \).

4. Dequeue \( \{R_1\} \). Best cost: 0.227 (\( \{o_1,o_2,o_3\} \)). Queue: \( \langle 0.203 = \{R_1,R_2\}, 0.217 = \{R_2\}, 0.225 = \{R_1,R_2,R_3\} \rangle \).

5. Report the set \( \{o_6,o_7,o_8\} \) as the best group since \( 0.199 \leq 0.203 \). $\square$

**Algorithm 4:** DoSearch(\(index, q, k, U\))

1. \( R \leftarrow \text{new result list}; \)
2. while \( U \neq \emptyset \) do
3. \( N \leftarrow U.\text{Dequeue}(); \)
4. if \( N \) contains a non-leaf node then
5. \( \text{candidates} \leftarrow \text{empty set}; \)
6. Calculate all possible combinations of entries, including single entries in \( N \), and add them to \( \text{candidates} \);
7. foreach \( \text{Entry } e \in \text{candidates} \) do
8. \( U.\text{Enqueue}(e.M, \text{MinCost}(q,e)) ; \)
9. else if \( N \) contain only leaf nodes then
10. Find a set of objects, \( O \) from \( N \), where \( O \subseteq N \) and \( \forall o \in O (o \notin R) \) such that \( \text{Cost}(q,O) \) is smallest;
11. if \( \text{Cost}(q,O) \leq U.\text{First}().\text{Value}() \) then
12. Output \( O \) as the next relevant group;
13. \( R.\text{Add}(O); \)
14. if \( k = |R| \) then
15. return;

**Multiple Keyword Query Processing**

We proceed to describe the algorithm for handling multiple keyword queries. The definitions of distance to a group and group size are independent of the terms occurring in a group. The group proximity defined in Equation 2.2 recursively multiplies the group proximity score of each term and thus provides a lower group proximity value when multiple terms occur. This has the desired effect of favoring groups that contain multiple keywords.
Recall that if a group does not contain any objects with a term from the set of query keywords, we cannot evaluate the function defined in Equation 2.2. Thus, we will not consider such groups as result candidates. This ensures that the resulting group will contain at least one object with the term.

As described above, Algorithm 4 finds groups of objects that can be located in several nodes. We can utilize this property when evaluating queries with multiple keywords. However, the algorithms presented so far only work on single GER-trees. Since we have a GER-tree per term, it is necessary to search several trees for candidate groups. Thus, we present a multiple keyword search algorithm, MKSearch, in Algorithm 5.

The MKSearch algorithm works with several GER-trees. The relevant trees are retrieved in Line 1 in Algorithm 5. Next, all root nodes from the sources are combined into a set and enqueued into a priority queue with the minimum cost 0. This queue is then passed to DoSearch Algorithm 4. The lower bound cost function defined in Equation 2.6 supports multiple keywords by using the extended histograms. Likewise, the cost function from Equation 2.3 supports multiple keywords. Thus, the search algorithm presented in the previous section can then be directly applied for several keywords.

Algorithm 5: MKSearch(index, q, k)

1. \( T_i \leftarrow \text{GER-Tree of the term } t_i \in q.\phi; \)
2. \( U \leftarrow \text{new min-priority queue}; \)
3. \( \text{Root} \leftarrow \text{empty set}; \)
4. \( \text{foreach} \ \text{GER-Tree } t \ \text{in} \ T \ \text{do} \)
   5. \( \ \ \ \ \text{Root} \leftarrow \text{Root} \cup t.\text{root}; \)
6. \( U.\text{Enqueue}(\text{Root}, 0); \)
7. \( \text{DoSearch}(index, q, k, U); \)

Enhancing the GER-Tree

The index histograms are built at insertion time, and when performing queries, leaf node entries are combined. Although the minDist bound limits the number of possible combinations, the inter-object distance calculations are still expensive. In the following, we propose two techniques that reduce the time spent on evaluating the combinations and present also a bulk insertion approach.

Storing Calculations

At insertion time, we have to perform inter-object calculations at every insertion. And during query processing, the possible combinations of leaf objects have to be evaluated. To limit the number of needed calculations at insertion
and query time, we propose a variant of the GER-tree, called the Precomputed GER-tree (PGER-tree), where leaf nodes store additional information.

In particular, we store a compressed representation of distances in leaf nodes such that a leaf node contains enough information to enable recreation of all possible combinations of leaf objects without computing any inter-object distances. At insertion time, this will reduce the time needed to create the compressed histograms in parent non-leaf nodes. At query time, when the leaf level is reached, the only distances that need to be calculated are those between objects located in different nodes.

We propose a simple representation from which all distances can be inferred. For each object, a list of distances to the other objects not already described are stored. The object identifiers are implied by the positions in the lists. For example, the representation of the objects in Figure 2.2 will result in a set of lists: $(2, 3.16, 5.83, 6.71, 6.32, 5.83, 5.00)$, $(1.41, 5.10, 6.08, 7.21, 7.07, 6.40)$, etc. The first list contains distances from object $o_1$ to the other 7 objects. The second list contains distances from $o_2$ to the others except $o_1$ that is already described in the previous list.

These compact representations are used in Line 2 and Line 10 in Algorithm 1 and Algorithm 4, respectively.

**Block Sharing Among Leaf Nodes**

The compressed histograms are built from the objects stored in the leaf nodes. Since only little space is available for histograms, these can only store limited information. And increasing the histogram size decreases node fanout. When leaf nodes store large numbers of objects, the histograms that describe them become less effective in pruning. Likewise, if we reduce the number of objects in a leaf node, we can achieve a more effective histogram for the leaf.

We thus propose a variant of the GER-tree, called the Combined Leaf GER-tree (CLGER-tree), where pairs of leaf nodes share a disk block. Since the result of a query may be located in several leaf nodes, two or more leaf nodes may have to be read to answer the query. When leaf nodes share disk blocks, a leaf node needed by a query may already be present in the buffer since the full disk block that contains the node may have been read from disk previously.

**Delayed Compressed Histogram Computation**

When inserting objects into the GER-tree, it is necessary to update the compressed histograms in non-leaf nodes. This has great impact on the insertion time. The insertion algorithm described previously uses the standard R-tree algorithms for splits and $MBR$ creation. Thus, we may simply create the R-tree, but only reserve space for the compressed histograms without computing them.
While the compressed histograms are initially empty, the spatial structure of the tree is identical to the spatial structure of the tree obtained when the compressed histograms are maintained at each insertion. After all objects have been inserted, the compressed histograms are created in bottom-up fashion, which reduces the insertion time.

The resulting GER-trees are identical to the ones obtained with eager histogram computation. Note also that eager histogram computation can be applied after the initial bulk insertion in order to enable ad-hoc insertions.

2.4 Experimental Evaluation

We proceed to report on an empirical study of the insertion and query performance of the proposed techniques. We first cover the experimental setup and then the findings from a range of specific experiments.

Experimental Setup

**Algorithms Considered** We implemented the techniques described in the paper, along with the different described enhancements. We use the name PCLGER-tree for tree that combines the PGER-tree and CLGER-tree, and we report on experiments with these three trees.

**Datasets** The experiments are performed on 3 datasets obtained from Google Places [36]. Objects from Google Places are each tagged with a set of terms from a list of 93 categories.

The first dataset, GPLACE1, consists of 27,171 real objects located in central San Francisco, USA. The maximum distance in the Euclidean space of this dataset is 14 km. The second dataset, GPLACE2, contains 43,062 real objects from Las Vegas that are within a diameter of 36 km. The third dataset, GPLACE3, is created by shifting the objects from four different US central city regions so they all appear in the Las Vegas region and thereby yield a dense, synthetic city with 139,246 real objects. Like GPLACE2, these objects are located within a region with maximum extent 36 km.

We use GPLACE1 in the experiments unless stated otherwise. We use the other two datasets when we vary the dataset size. The spatial distribution of the objects in GPLACE1 is depicted in Figure 2.5(a) with the most central part of San Francisco visible in the upper right corner.

**Queries** To study the performance of the GER-tree and the two enhancements, we create three sets of queries for each of the three datasets. Each query set contains 100 queries. The queries for a dataset are generated by first forming the bag of terms containing all occurrences of terms in all objects in the dataset. Terms that occur often in objects then occur often in the bag. The term distribution is shown in Figure 2.5(b). We then select 10 keywords at random from the bag.
2.4. EXPERIMENTAL EVALUATION

This procedure aims to ensure that the choice of query terms mirrors the term distribution in the data, which we believe is more realistic than selecting simply from the set of terms. For example, “restaurant” occurs more frequently than “plumber” in objects. The query construction then assumes that users will also query for “restaurant” more often than for “plumber.”

Finally, we generate 10 random locations within the dataset boundaries and obtain 100 queries by combining the 10 keywords with the 10 locations.

**Settings** The indexes are created and stored on disk using a page size of 4KB. The algorithms are implemented in Java, and the experiments are run on an Intel(R) Core(TM) i5-2520M CPU at 2.50GHz.
The maximum allowed main memory buffer size is set to 10% of the size of the tree used in a particular experiment. A random replacement buffer is used. If not specified otherwise, the compressed histogram size is set to 22 entries for the (P)GER-trees and to 27 entries for the PCLGER-tree since these values provide the best performance according to Figure 2.6(b).

If not specified otherwise, we fix the value of $\alpha$ at 0.9, $\beta$ at 0.2, and we retrieve the top-3 groups. Table 2.6 shows that these settings yield an average group size of 2.2 objects for the top-3 groups in a result. Further, Figure 2.8(b) shows that these settings result in an average query-to-group distance of 745 m and an average group diameter of 10 m.

**Experimental Findings**

**Varying the Compressed Histogram Size**  In this experiment, we vary the number of entries in the compressed histograms. This influences both the insertion and query performance.

We consider only histogram sizes that fully utilize the space available in a node. For example, we do not consider histograms with 12 entries because these are dominated by histograms with 13 entries—the reduction of the histogram size from 13 to 12 entries does not enable higher node fanout.

Since histogram size affects fanout, it also affects the number of splits that occur during insertion. This then increase the time to build the index. The (CL)GER-trees update the histograms at every object insertion, whereas the Bulk(CL)GER-trees postpone the histogram updates.

Figure 2.6(a) shows that the BulkGER-tree outperforms the other trees. This tree is lower than the CLGER-tree, and histogram updates are postponed.

The time needed to maintain histograms in the non-leaf nodes depends on the number of index entries (or subtrees) in the node. When propagating the histograms up the tree, all possible combinations of histogram entries from the histograms in the index entries must be examined. Therefore, more computations must be performed as the histogram size decreases. For very small histogram sizes, Figure 2.6(a) shows that all four trees become slower as the histogram size decreases. In this range, the GER-tree is faster than the CLGER-tree since it has a smaller height.

Larger histograms yield smaller fanout and thus more splits and nodes to maintain. Thus, we see increasing insertion time for the two bulk indexes since the time to build the R-tree becomes dominant. In contrast, the two non-bulk indexes become faster with smaller fanout since fewer of the dominating histogram entry combinations must be examined for every insert. The CLGER-tree becomes faster than the GER-tree since the small leaf nodes also reduce the number of possible object combinations.

Histogram size affects query performance greatly. With small histograms, the fanout is high, but little information is also available about subtrees.
This adversely affects the ability to prune, which reduces query performance significantly, as seen in Figure 2.6(b).

As the histogram size increases, query performance exhibits a decreasing trend. Only as the histograms get to be very large does the query time deteriorate slightly, which is due to the low fanout.

We attribute the “jumping” performance behavior seen in Figure 2.6(b) to the R-tree node splitting algorithms. The locations of objects in leaf nodes greatly impact the histograms, so the particular placements of objects into leaf nodes and the splits performed during insertion also affect query performance.

**Varying α** We proceed to consider an experiment designed to observe
the effect of varying the $\alpha$ parameter.

Recall from Equation 2.3 that parameter $\alpha$ allows us to prioritize between the spatial properties of a group and the group's textual relevance to the query, which takes into account both the number of objects and the textual relevance of the objects. As the value of $\alpha$ increases, the most relevant groups tend to be located closer to the query. This also means that the number of objects in the groups decrease with increasing $\alpha$. The average number of objects per group for different values of $\alpha$ are shown in Table 2.6.

If parameter $\alpha$ is set to 0, the most relevant group consists of all objects in the dataset that contain a query keyword in their text description. When set to 1, the number and text relevance of objects in a group are not considered, and the nearest object is returned.

Figure 2.7 reports on query performance for varying $\alpha$. Figure 2.7(a) show as expected an improvement in the performance for larger values of $\alpha$. Figure 2.7(b) shows that the number of page accesses decreases. The larger leaf nodes in the (P)GER-trees results in more possible group combinations and less descriptive compressed histograms.

**Varying $\beta$** The next experiment is designed to observe the effect of varying parameter $\beta$, which allows us to balance between the spatial distance from the query location to the group and the diameter of the group.

With $\beta$ close to 0, the diameter of a group becomes more important than the distance from the query to the group (cf. Equation 2.3). This has the effect of improving the query time because the objects in the group are more likely to be located close together in the index, which enables more effective pruning. This is seen in Figure 2.8(a).

The performance of the PCLGER-tree decreases less markedly than that for the other trees because it only holds half of the leaf node objects as do the (P)GER-trees. Thereby, more candidate groups are distributed across several leaf nodes, also for small values of $\beta$. As $\beta$ increases, the diameter of the groups tend to become larger, meaning that groups tend to be distributed across even more leaf nodes.

Figure 2.8(b) shows that, as expected, the group size increase while the distance to the group decrease.

**Varying $k$** Figure 2.9 shows the results when varying the number of result groups. As expected, query performance decrease with increasing $k$, which can be seen in Figure 2.9(a).

Also, the PCLGER-tree outperforms the (P)GER-trees as expected, and quite substantially so. This is for all $k$.

\[
\begin{array}{cccccccccccc}
\alpha: & 0.5 & 0.55 & 0.6 & 0.65 & 0.7 & 0.75 & 0.8 & 0.85 & 0.90 & 0.95 \\
|G|: & 9.2 & 8.7 & 7.7 & 7.0 & 6.3 & 5.5 & 4.4 & 3.5 & 2.2 & 1.2 \\
\end{array}
\]

Table 2.6: Average Group Size When Varying $\alpha$
For values of $k < 3$ the PCLGER-trees access slightly more pages than the (P)GER-trees as seen from Figure 2.7(b). The leaf nodes in the (P)GER-trees describe twice as many leaf objects as does the PCLGER-tree, which may then be of lower height, which in turn has the effect of generally incurring fewer page accesses. However, more possible object combinations exists for every leaf node, resulting in more calculations and thus slower performance of the PCLGER-tree.

Since PCLGER-tree leaf nodes contain only half as many objects as do
Figure 2.8: Query Time When Varying $\beta$

leaf nodes in the two other trees, the compressed histograms in the parent nodes of the PCLGER-tree are more descriptive than are those in the two other trees. Thus, fewer leaf nodes are fetched compared to the (P)GER-tree. The difference in page accesses increases as expected with $k$ since more leaf nodes are examined.

**Varying the number of keywords** Figure 2.10(a) shows the query time for different number of keywords for $k = 1$. As the number of keywords increases, more trees are fetched. Thus, the number of possible combinations
of node entries increases, and the size of the candidate set calculated in Line 6 in Algorithm 4 increases with the number of keywords. This results in more calculations, and since the number of combinations of histogram entries increases, the performance deteriorates for all three trees with increasing number of keywords.

**Varying the buffer size** When decreasing the buffer size from the maximum 10% of the index size, the number of page accesses increases for all three indexes—see Figure 2.10(b). This is as expected.
Figure 2.10: Performance For Varying Number of Keywords, Buffer Size, and Dataset
2.5. RELATED WORK

Varying the dataset In this experiment, we employ all three datasets. Dataset GPLACE2 contains 58% more objects than GPLACE1. However, Figure 2.10(c) shows that the query time increases only 12%. The objects in GPLACE2 are distributed across a larger region than those in GPLACE1, making it more easy to prune subtrees. Therefore, the query time does not increase significantly for any of the three algorithms. However, the GPLACE3 dataset contains a higher concentration of objects, which reduce the pruning power and yields a significant increase in the query time.

2.5 Related Work

Spatial keyword querying has recently started to attract substantial attention. With only four exceptions, to the best of the authors’ knowledge, all works in the literature compute results with a single-object granularity, and no other work addresses the problem studied in this paper. Here, we first consider indexes with aggregated information, then single-object solutions, and finally the solutions that compute a set of objects.

Aggregate Queries Our GER-tree maintains a particular kind of histogram in R-tree node entries. Previous work exists that has placed aggregate information about subtrees in node entries in R-trees and other trees in order to compute spatial aggregate queries [45, 47, 56, 82, 84]. However, no studies employ histograms to provide efficient pruning, but store only single aggregate values such as count and sum. Also, they do not consider objects with both spatial and textual values and thus do also not take into account the textual relevance of the objects indexed.

Top-k Most Relevant Spatial Web Objects Cao et al. [12] review recently proposed spatial keyword querying techniques, and Chen et al. [19] review and compare proposals for indexing in relation to spatial keyword querying.

Zhou et al. [98] propose to use inverted files for spatial keyword querying and create an R*-tree for each distinct term. This is similar to the Spatial Inverted Index (S2I) by Rocha-Junior et al. [76]. We employ the same general approach of having one index per data term. However, unlike our proposal, these previous works consider results with single-object granularity and do no employ compressed histograms to prune the search space.

Another approach, yielding the IR-tree family of indexes, has been proposed by Cong et al. [23, 90] and Li et al. [58]. Here, inverted files are attached to the nodes in R-tree type structures. Work with hybrid indexes have also been proposed by others [27, 11]. Li et al. [57] investigate direction-aware spatial keyword querying, the idea being to take into account the querying user’s movement direction. Wu et al. [92] study the processing of continuous spatial keyword queries for moving users.
Cao et al. [14] address the problem of retrieving the top-$k$ prestige-based spatial web objects. They introduce PageRank-like techniques that give objects prestige and improved rankings in results if they are located close to other textually similar objects. Unlike in all other works, objects are thus not independent of each other. However, only $k$ single objects are returned, and the IR-tree is used for query processing.

The above solutions return single-object granularity results, while we compute groups of objects. They store at most simple information in nodes, e.g., the count of the terms in a subtree, while we employ compressed histograms. We know of no way to adapt previous techniques to efficiently process the top-$k$ groups spatial keyword query.

**Sets of Spatial Web Objects** Four works consider queries that return a group of objects [15, 61, 96, 97]. However, their queries differ significantly from our proposal since they aim to find a set of objects that, when considered as one object, matches the query keywords. If a single object matches all the keywords in a query, only that object is returned. Thus, they do not aim to support browsing behavior, but instead return the minimal set of objects that together match the query keywords. Also, these queries return only a single group of objects rather than $k$ groups. Further, when evaluating textual relevance, the objects in a group are treated as a single object, and a group of objects satisfies the textual relevance if its objects collectively contain the query keywords, meaning that the textual matching is Boolean.

In contrast, our type of query ranks objects individually (each object must be relevant to the query) and numerically. Also, while the previous works find a spatially nearest (in a specific sense) minimal set of objects that collectively satisfies the query keywords, our query aims to find large groups of objects. As a result of these differences, the type of query we study calls for a very different solution. A demo of a system based on this paper’s proposals was presented at VLDB 2014 in a four-page demo paper [9].

### 2.6 Conclusions and Future Work

This paper presented a novel and carefully motivated and designed top-$k$ groups spatial keyword query that, given a query location and keywords, retrieves nearby, dense, and relevant groups of spatial web objects such as geo-tagged web pages, business directory entries, or check-ins.

Then techniques aimed at efficient processing of such queries were presented. These include a new R-tree based indexing technique that stores compact histograms in node entries that approximate subtrees in a particular manner while preserving reasonable node fanout. The accompanying query processing algorithms use the index and its histograms for pruning the search space and directing the search while taking into account group diameter and distance and relevance to the query.
An empirical study with real data offers insight into the design properties of the proposed techniques and suggests that the techniques are capable of supporting query processing in realistic, real-world settings.

Future research may consider other types of indexes and new kinds of histograms, which then calls for new query processing algorithms.

Acknowledgments

This research was supported in part by the European Union Seventh Framework Programme - Marie Curie Actions, Initial Training Network Geocrowd (http://www.geocrowd.eu) under grant agreement No. FP7-PEOPLE-2010-ITN-264994.
Chapter 3

GroupFinder: A New Approach to Top-K Point-of-Interest Group Retrieval

Abstract

The notion of point-of-interest (PoI) has existed since paper road maps began to include markings of useful places such as gas stations, hotels, and tourist attractions. With the introduction of geo-positioned mobile devices such as smartphones and mapping services such as Google Maps, the retrieval of PoIs relevant to a user’s intent has become a problem of automated spatio-textual information retrieval. Over the last several years, substantial research has gone into the invention of functionality and efficient implementations for retrieving nearby PoIs. However, with a couple of exceptions existing proposals retrieve results at single-PoI granularity. We assume that a mobile device user issues queries consisting of keywords and an automatically supplied geo-position, and we target the common case where the user wishes to find nearby groups of PoIs that are relevant to the keywords. Such groups are relevant to users who wish to conveniently explore several options before making a decision such as to purchase a specific product. Specifically, we demonstrate a practical proposal for finding top-$k$ PoI groups in response to a query. We show how problem parameter settings can be mapped to options that are meaningful to users. Further, although this kind of functionality is prone to combinatorial explosion, we will demonstrate that the functionality can be supported efficiently in practical settings.
CHAPTER 3.

3.1 Introduction

Many online services exist that support the retrieval of points-of-interest (POIs), including international services such as Google Maps [46] and country-specific services such as the Danish Krak [53]. In addition, vehicle navigation services offer category-based PoI search functionality based on the user’s current location. Services such as these retrieve nearest-neighbor PoIs relative to the user’s location. They do this at single-PoI granularity, meaning that they consider each PoI in isolation and return results consisting of $k$ single PoIs, each of which is a result object. In many use cases, this functionality is what users expect. For example, this can occur when a driver is looking for the most convenient gas station.

In contrast, we support the common browsing behavior use case, where users prefer to explore several options prior to making a decision. Such behavior can occur when a tourist is looking for a place to get dinner. In this case, the user may wish to look at the menu options and prices at several restaurants prior to making a dining decision, and seating availability may cause the user to consider additional options. Here, the standard $k$ nearest neighbor functionality falls short: While the nearest neighbor is close to the user’s location, it may be relatively far from the next nearest neighbor, which may again be relatively far from the third nearest neighbor. In the above browsing behavior case, the user is better served by a group of relevant PoIs that are both near the user’s location and near each other. We support the retrieval of $k$ such groups.

To improve the utility of single-object granularity results, previous research has tried to incorporate scoring functions to take the quality of each PoI into account [77], thus returning the best PoIs in terms of both distance and quality. While this does hold the potential for retrieving better results, they still have single-object granularity. For example, a nearby, high-quality restaurant may be too expensive, and the next nearest neighbor may still be relatively far away. One study considers two instance of a collective query that treat a set of PoIs as a single object [13, 15]. Both return only a single set (i.e., $k = 1$), and both apply Boolean matching to the keyword part of the queries and apply only ranking to the spatial parts of the queries. These queries support the different kind of use case where the PoIs in a result set complement each other and where a user wishes to visit every single PoI in the result set. In our use case, the PoIs in a result compete against each other, and the user intuitively wishes to find the best one.

We demonstrate GroupFinder, a web-based service that returns top-$k$ groups of PoIs according to a scoring function. In addition to taking into account the user’s current location and user-supplied query keywords, the function takes into account transportation-mode information that captures how the user intends to travel to a group of PoIs and how the user intends to travel between the PoIs in a group. Groups are retrieved by means of a
new type of index called a Group-Extended R-tree (GER-tree). Query results are displayed using the Google Maps infrastructure, and the responsive web design front end Twitter Bootstrap is leveraged in order to make the service equally useful on traditional desktops and new devices such as tablets and smartphones.

The paper is organized as follows. Section 3.2 introduces the indexing technique underlying GroupFinder. Section 3.3 presents an overview of the communication and architecture of GroupFinder. Then Section 3.4 presents the user interface and covers how results are displayed. Finally, Section 3.5 details the demonstration setup.

3.2 Preliminaries

GroupFinder exploits a new indexing technique, the GER-tree, that indexes PoIs with a geo-location and a text description. First, a vocabulary of searchable keywords is built based on the text descriptions of the PoIs. For each such word, an R-tree is built on the location of each PoI that contains the word in its text description. This yields a forest of R-trees that each enable spatial queries against the PoIs with descriptions that contain a particular word [76]. Next, to support the retrieval of groups rather than single objects, each R-tree is extended with purposefully designed, compressed histograms in all non-leaf nodes. The histogram in a node contains information on how densely grouped the subtrees of the node are, how many objects the groups contain, and give the minimum bounding rectangles of the subtrees. Further, a score is calculated for each PoI in a tree, to indicate its relevance to the tree’s keyword. These scores are averaged for groups and added to the compressed histograms. The resulting index enables efficient retrieval of groups in the vicinity of a location.

3.3 System Architecture

GroupFinder is completely web based and is thus accessible from any web browser. To provide the best user experience, we employ a client-based approach to retrieve the current location of the user, while we use the conventional client/server approach to process queries.

Geo-Locating a User

We apply several techniques to accurately and non-invasively geo-locate users in a generic, platform-independent manner.

We first use the ClientLocation tool of the Google Maps infrastructure to geo-locate a user based on the user’s IP. This approach allows us to identify the current city of the user in many cases, thus providing a rough estimate of
the user’s geo-location. When the user’s approximate location is found using this approach, we use the location to position the map at the user’s location at an overview zoom level.

Next, we ask the user for permission to use the HTML5 geo-location API. Having obtained this permission, we apply a finer zoom level to the map to better display the user’s more accurate location. It is also possible for the user to indicate a preferred position by simply clicking on the map. This functionality adds flexibility and can be used to correct positioning errors.

Since modern browsers, including those of tablets and smartphones, support the geolocation API, we can always locate the user if given permission. We do not need to use any native apps that may be available on different mobile platforms. The resulting device independence is very attractive. The approach taken also allows us to provide users with a good experience while affording the users control of whether or not to reveal their actual location.

**Client-Side Querying**

To explain the process of retrieving the \( k \) most relevant groups, we consider the case where a user is standing at the central railway station in the city of Aarhus, Denmark. This location is shown in Figure 3.1.

![Figure 3.1: Before Querying](image)

**Keyword Handling**

To query for groups of PoIs, the user types a query keyword. In particular, when the web service is loaded, the available keywords are generated server side and sent to the client so that the client has an up-to-date set of keywords. When the user starts to type, Google-style suggestions are presented to make it as easy as possible for the user to formulate a query.
In the demonstration, we use a dataset derived from Google Places, since this is one of the best sources of PoIs. In Figure 3.2, the user searches for restaurants.

Distances To and Within Groups

A user may prefer different transportation modes when traveling from the current location to a group. Likewise, a user may prefer different transportation modes when traveling between the PoIs in a group. For both types of travel, GroupFinder allows a choice among three transportation modes: by car, by bicycle, by foot. These settings are used to set problem parameters in the ranking function that control which groups are returned to the user. Specifically, when scoring a group, the ranking function takes into account the distance to the group and the group’s density, and both aspects have a weighting parameter that takes a value in the range $[0, 1]$. Each transportation mode is then mapped to a value for the two parameters.

The result is that nearby or dense groups are preferred when the turtle icon (by foot) is selected for the distance to groups and the intra-group distance. If the car icon is selected, spatial proximity and density are less important, and preference is instead given to groups with high average text relevance to the query and with many PoIs. In the query covered in Figure 3.3, the car icon has been chosen for both distances. Consequently, the result contains relatively large groups with textually very relevant objects, at the expense that one group being relatively far from the query location.
CHAPTER 3.

The Number of Groups

While the underlying techniques allow the retrieval of any number \( k \) of groups, GroupFinder allows the user to retrieve from 1 to 5 groups. We expect that if a user is facing a light-weight decision (e.g., buy milk), one nearest group is enough (e.g., to find a store that is open and has milk), while the user may want to see several groups if facing a more “complex” decision (e.g., finding a restaurant to have dinner at). As the underlying techniques return groups progressively, the service displays groups as soon as they are available. Figure 3.2 shows a query in progress: two groups have been found, and a blue progress bar above the map indicates the percentage of the query that has been completed.

Query Processing

Query processing as seen from the client is outlined in Algorithm 6. To maximize the amount of concurrent users, all communication is asynchronous. The first request in Line 3 simply sends the collected information to the server that responds with a searchid. The client then repeatedly requests additional groups with a small delay between requests until it has the desired number of groups. While several factors influence the query time, a typical query completes in .3 to 1.5 seconds on an Intel(R) Core(TM) i5-2520M CPU at 2.50Ghz. To offload the server, a group only contains a set of coordinate pairs, while the bounding region and the pan and zoom instructions are calculated on the client using JavaScript. Information about the individual PoIs in a group is obtained by a separate call when the user clicks on the group. On the server side, the web service is implemented using a Tomcat server and Java. The indexing and query processing are also implemented in Java, and they are
3.4. **THE GROUPFINDER WEB SERVICE**

We proceed to describe the web service prototype.

**User Interface**

The user interface consists of two parts—see Figure 3.4. On the left side, the user can provide the query parameters, as described in Section 3.3. We have chosen to use discrete options for the travel preferences. These options map well to different transportation modes, and this approach makes it easy to obtain query results. The fields for keywords and the number of groups are self explanatory. It should be mentioned that the quality of the groups found is independent of the number of groups to be retrieved. The best group is found solely based on the keyword and distance parameters. On the right side, we use Google Maps to let the user choose their location if the user wants to use a location different from the one found automatically.

---

**Algorithm 6**: Search from client

begin

1. \( \text{amountRecieved} \leftarrow 0; \)
2. \( \text{searchID} \leftarrow \text{HTTP GET:} \)
   \[ \text{[lat, lng, term, amount, distIGR, distTGR]} \]
3. while \( \text{amountRecieved} < \text{amount} \) do
   4. \( \text{group} \leftarrow \text{HTTP GET:} \) \([\text{searchID}]\);
   5. if \( \text{group} \neq \text{NULL} \) then
      6. \( \text{amountRecieved} + +; \)
      7. \( \text{PanAndZoom(group);} \)
      8. \( \text{Hull} \leftarrow \text{ConvexHull(group);} \)
      9. \( \text{DisplayGroup(Hull, group);} \)

deployed directly in the server to minimize response times. When a query is initiated, a new thread is started to query the index, and a mapping between the thread and \textit{searchId} is created. The query thread finds the tree to query and initiates a search in the tree. The threads that query the same tree share the same buffer. Thus, different threads that access the same nodes benefit from the shared buffer and save disk reads, which improves performance. When the client requests the next group, the server simply looks up the thread handling the query in the map and checks if new groups are available. If so, the top group is returned to the client.
Querying

Once a query has been issued, the client side scripts start to request results from the server. When the first group is received, it is displayed on the map by zooming and panning so that the user’s location and the group fit on the map. The pan and zoom adjustment of the map is done in a smooth animation, so that the user does not loose track of their location. When the next group is received, we again zoom and pan to display the user’s location and the two groups on the map. This process terminates when all groups have been presented. A progress bar indicates the degree of completion and disappears when the search is complete. Figures 3.1-3.3 depict the process of searching for three groups of restaurants near the central train station in the city of Aarhus, Denmark.
3.4. THE GROUPFINDER WEB SERVICE

Result Presentation

The results are returned and displayed as groups. To clearly outline a group, we calculate the convex hull of the group on the client and then color the polygon that represents the convex hull. This is seen in Figures 3.2 and 3.3. Each vertex of the convex hull is marked by a circle to ensure that the group is always visible, even when the objects in the group lie on a line. We overlay a text that describes each of the groups to help the user decide which group is most desirable. The text includes the rank of the group in the list of all results along with the number of objects in the group.

Since a group can contain many objects, it can be difficult to see how many objects a group actually contains when using normal maps markers. To provide a better overview, we have replaced the normal markers with rings. This ensures that objects close to each other can be told apart. If a user clicks on a group, the map automatically pans and zooms onto that group while keeping the user’s location on the map. That way, the user can get a better look at a specific group without losing track of where they are in relation to the group. Another click on the group makes the map zoom back out to again show all result groups.

Effect of Distance Settings

To show the impact of the distance settings, Figure 3.7 shows the result of the query considered so far (shown in Figure 3.3), but with the distance indicators set at the other end of the scale.

The groups found now are more dense, contain fewer PoIs, and are closer to the query location. The result matches the intuitive semantics well, since a person on foot is unlikely to want to walk far to visit a group where the
objects are also not in close proximity of each other. This result is “opposite” to the result in Figure 3.3, where the groups contain more objects, are further away from the query location, and contain objects that are further apart from each other.

Although we aim to support browsing behavior, a query can also return a result that contains a single PoI. If a very relevant PoI exists close to the user, but not close to other PoIs, a single-object group with that PoI may be returned.

Cross Platform Support

As mentioned, we use the Twitter Bootstrap front end framework to display our data. This allows for the same interface to be used across different client platforms without the need for any native apps on special devices such as tablets or smartphones.

Thus, a user is always presented with the same, familiar interface when using GroupFinder from different platforms. Figure 3.4 shows the desktop interface, while Figure 3.5 shows the interface on a tablet, and Figure 3.6 displays a possible smartphone look at the interface, although orientation and device resolution can affect the experience.

The responsive design of Twitter Bootstrap moves the individual parts of the interface around, but it is still easy to find the buttons needed, thus making the web service useful across platforms.

3.5 Demonstration

GroupFinder offers new functionality for PoI querying. This functionality targets use cases characterized by browsing behavior where users wish to explore several competing PoIs prior to making a decision such as choosing a restau-
rant for dinner or choosing which shoes to buy. We are not aware of any other services that target this behavior.

The demonstration is done in two parts. First, we introduce the underlying indexing and query processing techniques in more detail, showcasing how we rank groups of PoIs and how the query parameters affect the query processing.

Second, we give the participants the chance to experience the system themselves in a live demonstration with a publicly available web service. We provide data for Aarhus and other cities to make the demonstration as relevant as possible. The attendees are able to choose from a huge set of keywords and will be able to quickly get an overview of where in their vicinity to go for food, entertainment, or a drink.

The web based interface allows users to use the web service regardless of operating system, device hardware, or browser preference. This demonstration therefore allows the attendees to not only experience GroupFinder during the demonstration sessions. They can also use GroupFinder on their own at any time to explore the surroundings of the conference location.
Chapter 4

A Clustering Approach to the Discovery of Points of Interest from Geo-Tagged Microblog Posts

Abstract

Points of interest (PoI) data serves an important role as a foundation for a wide variety of location-based services. Such data is typically obtained from an authoritative source or from users through crowdsourcing. It can be costly to maintain an up-to-date authoritative source, and data obtained from users can vary greatly in coverage and quality. We are also witnessing a proliferation of both GPS-enabled mobile devices and geo-tagged content generated by users of such devices. This state of affairs motivates the paper’s proposal of techniques for the automatic discovery of PoI data from geo-tagged microblog posts. Specifically, the paper proposes a new clustering technique that takes into account both the spatial and textual attributes of microblog posts to obtain clusters that represent PoIs. The technique expands clusters based on a proposed quality function that enables clusters of arbitrary shape and density. An empirical study with a large database of real geo-tagged microblog posts offers insight into the properties of the proposed techniques and suggests that they are effective at discovering real-world points of interest.

4.1 Introduction

Points of Interest (PoIs) represent geographical entities (e.g., restaurants, museums, hotels), as a series of geo-referenced points with attached semantics (e.g., names or addresses). PoIs are important for a wide variety of services, e.g., Google/Yahoo/Bing Maps, Yelp, TripAdvisor. Also, they have high value
for companies like Factual and Google, who sell PoIs. PoIs can represent formal locations defined by authoritative sources of information (e.g., lists of landmarks and buildings for tourist attractions), or they can represent informal gatherings that result from local activities (e.g., places where demonstrations occur, picnic spots, public events). PoIs can be static (e.g., the Empire State Building) or changing (e.g., a specific exhibition at a museum). Also, PoIs can be relevant only during some time (e.g., during a festival).

PoIs may be collected in different ways that each has its shortcomings. Manual PoI collection and maintenance is expensive and affordable only to big corporations. It may be possible for PoI owners to contribute information themselves; however, it is likely that they only contribute to the most popular services and only if it is beneficial to them. Informal gatherings may be more difficult to capture due to the lack of knowledge about these or the absence of incentives. Collaborative projects such as OSM\footnote{http://www.openstreetmap.org} depend on volunteered, crowdsourced data, and thus coverage and quality vary greatly.

We leverage the increasing availability of geo-tagged microblog posts to enable a new way of obtaining PoIs. Increasingly large volumes of geo-tagged content are becoming available with the proliferation of GPS-enabled mobile devices. Specifically, users of such devices are creating large amounts of geo-tagged microblog posts describing their daily activities. Some users may report on information from their current location. For example, the circles in Figure 4.1a represent geo-tagged posts containing the term "hotel." Other users in the same region may report on information unrelated to the location (the square symbols in the figure). We aim to extract PoIs from such user-generated content.

Intuitively, users are likely to use specific textual descriptions more often in some regions than in others. For example, posts mentioning “hotel” might be more frequent in a region containing a hotel than in a region containing a supermarket. We assume a PoI can be created for a location if a sufficiently high percentage of the posts in the region of the location have a similar textual description. An example result is shown in Figure 4.1b where three clusters are formed from the objects in Figure 4.1a. When sufficiently many dissimilar posts exist in the same region, no cluster may be formed. Due to the nature of geo-tagged posts, the clusters may be of arbitrary shape and density.

A system that enables the creation of PoIs from geo-tagged microblog posts should satisfy several challenging requirements.

First, due to the diverse nature of spatial objects, the clusters may be of arbitrary shape. Second, in addition to the shapes being arbitrary, clusters can overlap, and a cluster can span several other clusters. Third, there can be significant amounts of irrelevant data in a region. A clustering algorithm therefore must be robust to noise or outliers (we do not distinguish between these two). Fourth, different clusters may have very different densities, and
4.1. INTRODUCTION

the density of posts inside a cluster representing a PoI may also vary. Fifth, a spatial clustering algorithm is required that takes into account not only the spatial proximity among posts, but also the similarity among their textual attributes. Sixth, since geo-tagged content is obtained in arbitrary order, the results of a good clustering approach should be independent of the ordering of the input data. In other words, it should be order-insensitive with respect to the input data.

To meet these requirements, we contribute a new type of clustering method for spatio-textual objects, termed CLUSTO (clustering of spatio-textual objects). Specifically, this method satisfies the following requirements:

1. Discovers clusters of arbitrary shape.
2. Enables overlapping clusters.
3. Is robust to noise and outliers.
4. Identifies clusters of varying density.
5. Takes into account both spatial and textual attributes.
6. Is insensitive to the ordering of the input data.

Many previous works have explored research directions related to microblogs [10, 11, 18, 30, 40, 66, 68, 78, 80, 81, 86, 93, 95]. However, to the best of our knowledge, there is no existing work that address the discovery of PoIs using geo-tagged microblog posts. Spatial data mining is a popular research area (see Section 4.5), where the most related work includes density-based clustering algorithms [31, 44, 50, 54, 79]. However, they fail to fulfill requirement 5, which is essential. In the experimental evaluation, we extend
 CHAPTER 4. 

DBSCAN to support non-spatial attributes, but show that it suffers from over-expansion.

CLUSTO is based on nearest neighbor chaining that fulfills requirements 1 and 6. The proposed quality-based clustering criterion solves, in combination with the nearest neighbor chaining, requirements 3, 4, and 5. CLUSTO allows overlapping clusters, depending on the setting of a quality-threshold (addressing requirement 2).

In the paper’s empirical study, we study the effectiveness of CLUSTO using a real-world dataset. By comparing with Google Places, we show that CLUSTO is able to successfully identify most PoIs from an abundance of geotagged tweets (we have accumulated more than 280,000 geo-tagged tweets from downtown San Francisco from the Twitter stream during a period of more than a year).

The remainder of the paper is structured as follows. Section 4.2 proposes the notion of quality based clustering. The proposed solution along with two base-lines are presented in Section 4.3. The experimental evaluation is given in Section 4.4. Section 4.5 covers related work. Finally, we conclude in Section 4.6.

4.2 Quality Based Clustering

A spatio-textual object has a spatial position and a textual description. An object may have a unique textual description, or parts of its description may be shared with other objects. Many different textual descriptions may exist in a dataset of spatio-textual objects, and we represent objects with different descriptions by different shapes in our examples. In Figure 4.2, squares may represent objects with text "hotel," while circles may represent objects with text "park."

Spatially nearby objects do not necessarily have similar textual descriptions. For example, consider the objects in Figure 4.2a. Although their spatial distribution or density is very similar, one can easily identify three clusters: two clusters of squares and one cluster of circles.

Objects that share a textual description may have neighboring objects with different textual descriptions. An example is provided in Figure 4.2b, where circles occur arbitrarily among the squares. The objects in this example may be regarded as a single cluster of squares with a small amount of noise. Also, the spatial distance between objects may vary greatly within a single cluster. For example, there may be a higher concentration of objects in a specific part of a park, but more sparse regions of the park may still be considered as part of the park.

When there is much variation in the textual description of the objects, it may not be possible to determine the type of the cluster. In this case, all the
4.2. QUALITY BASED CLUSTERING

Objects will be considered as noise. For example, in Figure 4.2c, there is no clear dominating textual description among the objects.

Figure 4.2: Sample Datasets where Shapes Represent Different Textual Descriptions

To simplify the problem of clustering spatio-textual objects, each textual description may be considered in an isolated manner. That is, clustering is performed for each set of objects that share a textual description in isolation from objects with different textual descriptions. As a result, clustering is based only on the spatial dimension. Many spatial clustering algorithms exist that take into account only the spatial dimension (see Section 4.5). However, this approach implies that the textual information is ignored and that clusters may expand over unrelated but spatially nearby objects. For example, when only the squares are considered in Figure 4.2a, the natural cluster is all the squares. As the spatial distribution among the square objects is similar, it becomes difficult to distinguish the actual clusters. When considering the circles, they clearly separate the two clusters of squares. Also, when textual descriptions are ignored, a set of noisy objects may form a false cluster. For example, the objects in Figure 4.2c may result in multiple clusters because the textual descriptions are considered in isolation. Therefore, purely spatial clustering is not adequate in our targeted setting.

Motivated by this, we aim for an approach that takes into account both textual and spatial attributes of an object and is able to detect the above clusters correctly. We continue to formalize the problem.

Clustering spatio-textual objects results in a set of objects which has a spatial region. The region may be defined using different techniques. In this paper, for ease of understanding, we define the region of a cluster to be the convex hull of all objects in the cluster. An example is provided in Figure 4.3a. Other techniques for enclosing the objects, like the orthogonal convex hull, the minimum bounding box, or the bounding sphere, may be used with the
proposed solution.

Definition 2. Let $D$ be a set of objects. Given a set of objects $O$ in $D$, function $\gamma: D \rightarrow D$ returns the minimal subset of $O$ that has the same convex hull as $O$.

The objects in the enclosing region, regardless of their textual description, are called $\text{ClusterEnclosed}$.

Definition 3. Let $D$ be a database of objects. Given a set of objects $O$ in $D$, function $\Lambda: D \rightarrow D$ returns the set of objects in $D$ that are enclosed by the convex hull of $O$. We say that the objects returned by $\Lambda(O)$ are $\text{ClusterEnclosed}$ by $\gamma(O)$.

An example of a set of $\text{ClusterEnclosed}$ objects is provided in Figure 4.3b. The set $O$ contains the squares, $\gamma(O)$ contains the 6 squares on the border, and $\Lambda(O)$ contains all 11 objects inside the convex hull. Note that the three circles do not contain the same textual description as the squares, but are still $\text{ClusterEnclosed}$ objects.

Figure 4.3: Sample Datasets where Shapes Illustrate Different Textual Descriptions

The addition of a single object to a given cluster may increase the number of $\text{ClusterEnclosed}$ objects by more than one. The new objects that become $\text{ClusterEnclosed}$ may or may not share the same textual description as the other objects in the set. In Figure 4.3b, three objects (circles) were added to $\text{ClusterEnclosed}$ when the top square object was included.

We proceed to define the quality of a given textual description, termed annotation, in a set of $\text{ClusterEnclosed}$ objects. Ideally, a set of $\text{ClusterEnclosed}$ objects is preferred, where all objects share the same textual description, as in Figure 4.3a.

Any cluster can easily be annotated with a textual description if the $\text{ClusterEnclosed}$ objects all share the same textual description. However, this may not always be the case. As seen in Figure 4.3b, objects with different
4.2. QUALITY BASED CLUSTERING

textual description may occur. These “noise” objects may be acceptable since
one cannot expect that all objects in a given region contain the same textual
description. For example, in a park, some may publish the information that
they are actually in the park, while others may publish information about
arbitrary topics. To accommodate this, we introduce means of measuring the
quality of an annotation of a cluster.

Intuitively, a cluster like the one in Figure 4.3c is of lower quality than the
cluster in Figure 4.3a. Figure 4.3a has seven objects that all share the same
textual description, making the annotation of the cluster obvious. However,
Figure 4.3c has six objects for each of two textual descriptions, making it
difficult to determine a good annotation of the cluster. We want to reflect this
in our quality function. The quality function depends on the number of noise
objects and the number of objects with a textual description that contains
the annotation, $a$. Thus, a smaller fraction of noise objects results in a better
quality of the cluster. More formally:

Definition 4. The quality of an annotation $a$ of a given set of objects $S$ is
given by:

$$q(S, a) = \frac{|S_a|}{|S|},$$

(4.1)

where $S_a$ is the objects in the set with a textual description containing $a$. The
function returns a value in $[0, 1]$, and the value 1 represents the best possible
quality.

Example 6. Consider the clusters in Figures 4.3b and 4.3c. The quality
of each cluster with respect to the square annotation is $8/11$ and $6/12$, re-
spectively. Clearly, the cluster in Figure 4.3b has a better quality since it is
dominated by square objects.

When a cluster is expanded by a new object or set of objects, the quality
of the cluster must be recomputed. Adding an object to a cluster with a large
number of objects has minimal impact on the quality score. Consider a cluster
like the one in Figure 4.3a with a large number of objects and a quality of
1. Adding an object to this cluster may also produce a good quality score,
even with a number of accompanying noise objects as seen from the example
in Figure 4.3b. Therefore, large clusters may expand excessively and may
include numerous noise objects. Large clusters with initially high quality are
in particular likely to suffer from this over-expansion.

To avoid expansions that decreases the quality of the resulting clusters,
we introduce a new measure of cluster quality when expanding clusters. The
expansion of a cluster involves merging two clusters that each consists of one
or more objects. The number of noise objects resulting from the merge may
have different impact on the quality when considering the clusters individuallly.
Therefore, we define the merged quality of two clusters to be determined by
the lower quality computed for each cluster in combination only with newly added (due to merge) objects.

Definition 5. The quality of a cluster with respect to an annotation $a$ when merging two clusters, $c_1$ and $c_2$ is given by:

$$mergedQ(c_1, c_2, a) = \min_{x \in \{c_1, c_2\}} q(\Lambda(c_1 \cup c_2) \setminus \Lambda(x), a)$$  \hspace{1cm} (4.2)$$

Example 7. We consider the clusters in Figure 4.4 and calculate the merged quality with respect to the square annotation. The 3 top square objects are cluster $c_1$ and the lower 7 objects are cluster $c_2$. When combining the two clusters, the merged quality is:

$$mergedQ(c_1, c_2, □) = \min\{\frac{3}{6}, \frac{7}{10}\} = 0.5$$  \hspace{1cm} (4.3)$$

Note that the quality of the complete cluster exceeds the merged quality. In this example the quality of the complete cluster is: $\frac{10}{13}$.

Figure 4.4: Two Clusters Separated by Three Noisy Objects

Candidate Annotations

A cluster consists of a number of objects that share a textual description, an annotation, and a number of noise objects. To find annotations that can form a cluster, we consider multiple combinations of the terms in the textual descriptions. We consider each single-term text and a multi-term text starting with capital letters as a candidate annotation. Thereby, we aim to capture places of interest and their descriptive terms.

Example 8. Table 4.1 lists all possible annotations for the textual description: 'We are having a barbecue in Central Park.' Note that stop words are removed.
4.3. PROPOSED SOLUTION

<table>
<thead>
<tr>
<th>annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨ barbecue ⟩</td>
</tr>
<tr>
<td>⟨ Central ⟩</td>
</tr>
<tr>
<td>⟨ Park ⟩</td>
</tr>
<tr>
<td>⟨ Central Park ⟩</td>
</tr>
</tbody>
</table>

Table 4.1: Example of Annotations

In the following, we consider clustering using only the above candidate annotations. Notably, many other approaches can be supported, e.g., all term combinations or all n-grams can be derived from textual descriptions. We also ignore the problem of lexical heterogeneity, i.e., when different textual descriptions refer to the same real-world objects (e.g., Museum of Modern Art versus MoMA). While we expect this to improve the quality of the clustering, such studies are beyond the scope of this paper.

4.3 Proposed Solution

We proceed to present the CLUSTO clustering technique. CLUSTO uses the above-defined merged cluster quality concept and is configured with two parameters. The first parameter, $\min Q$, defines the threshold for the $\text{mergedQ}$ value that controls when two clusters are merged. Two clusters $c_1$ and $c_2$ with annotation $a$ are merged if and only if $\text{mergedQ}(c_1, c_2, a) \geq \min Q$. The second parameter, $\min Size$, defines the minimum number of objects that a valid cluster must contain.

We start by giving a general description of how our quality-based function can be integrated into cluster expansion algorithms, including the approach taken by CLUSTO. Then we provide algorithm implementation details. We assume that each object is also a single-object cluster, and thus we use simply the term “cluster” in the following.

Spatial Cluster Expansion

There are several approaches to how $\text{mergedQ}$ can be integrated into spatial cluster expansion.

Range-based Expansion. Motivated by DBSCAN [31] and its cluster expansion based on a configured maximum range parameter ($\epsilon$), a similar approach can be taken to find clusters using the quality function. A random cluster is selected, and a range search with distance $\epsilon$ around it is performed. The cluster is merged with each of the clusters in the range if the merged quality of the resulting clusters ($\text{mergedQ}$) exceeds the given threshold ($\min Q$).

In addition to it being difficult to choose $\epsilon$ optimally, this approach suffers from the same problem as DBSCAN—varying density. Consider the example
dataset in Figure 4.2a, where the distances between the square objects are similar. In this case, it may be possible to select a suitable value for $\epsilon$. However, the distances between the objects in Figure 4.2b vary greatly. Therefore, to capture all clusters, the value must be set high in order to contend with objects that are far apart. This may result in large clusters since the quality is evaluated for all objects in the range. Thus, this approach fails to detect smaller clusters.

**Nearest Neighbor Expansion.** This approach expands a cluster based on its nearest neighbor (NN) cluster. Initially, a random cluster is selected, and an NN search is performed to find its NN cluster. As before, the two clusters are merged if the merged quality exceeds the given threshold. As such, this approach overcomes the varying-density problem of the previous approach. However, it is now sensitive to the order in which the clusters are considered for merging. Since the initial cluster is selected at random for each candidate annotation, the final clustering may vary from run to run.

In CLUSTO, we overcome the shortcomings of both approaches as follows.

**Reciprocal Nearest Neighbor Expansion.** Our approach builds on nearest neighbor (NN) chains [72]. An NN-chain consists of an arbitrary cluster, followed by its NN, which in turn is followed by its NN from the remaining clusters, and so on. Such a chain ends in a mutual or reciprocal (RNN) pair, i.e., a pair of clusters $c_1$ and $c_2$ such that the NN of $c_1$ is $c_2$, and vice versa. When such an RNN pair is found, the corresponding clusters are merged. Since such an expansion between two clusters is performed only if they have no other closer neighbors, the algorithm can recursively perform an agglomerative hierarchical clustering until all clusters are merged into a single (root) cluster. We employ single-linkage clustering. As discussed above, we extend this approach with a stopping condition based on the merged-quality function.

Since the next candidate is the nearest neighbor, it does not require any range parameters ($\epsilon$) and is immune to varying density.

**Correctness**

Using the NN-chaining algorithm, the resulting merged clusters are not affected by the order in which the clusters are selected for merging. That is, the algorithm eventually arrives at the same clustering independently of the order in which the clusters are traversed. The correctness relies on the reducibility property [72].

**Lemma 1.** The NN-chaining algorithm forms the same clusters independent of the ordering of the dataset.

**Proof:** We verify the reducibility property of the NN-chaining algorithm, implying that when two clusters are merged, they do not affect any other clusters.
Given two RNN clusters \( i \) and \( j \), and any other cluster \( c \), using single-linkage clustering, there is some (Euclidean distance) \( p \) satisfying the following:

\[
||i, j|| < p, ||i, c|| > p, \text{ and } ||j, c|| > p.
\]

With single-linkage clustering, no object is closer to \( i \) than \( j \) and vice versa. Thus, the following holds when merging \( i \) and \( j \):

\[
||i + j, c|| > p.
\]

This verifies the reducibility property. Thereby, \( i \) and \( j \) can be merged without effecting the RNN properties of any other clusters. \( \square \)

As a result of using a clustering algorithm that satisfies the reducibility property, the order in which clusters are discovered may vary, but the eventual clustering is the same. For example, consider the two dendrograms in Figure 4.5. The dendrogram in Figure 4.5a may have either object \( o_1 \) or \( o_2 \) as a starting point, whereas the dendrogram in Figure 4.5b may have any of \( o_3 \), \( o_4 \), or \( o_5 \) as a starting point. As seen from the figures, the order in which the clusters are formed may change depending on the starting point: however, the formed clusters are the same.

Intuitively, the order enforced by RNN expansion is favored because it first tries to merge the clusters that are closest to each other. This is confirmed by our empirical study (in Section 4.4).

This approach can handle excessive expansion of clusters and noise in the dataset. For example, consider the objects in Figure 4.6. Two clusters may be formed: one cluster with the five left-most objects and one cluster with the objects \( o_1, o_2, o_3, \) and \( o_4 \). With the reciprocal nearest neighbor requirement, object \( o_5 \) is regarded as noise since its NN is the cluster of four (and merging them would result in a much lower quality of the cluster). Similarly, such a stopping condition may prevent the inclusion of the upper-right object in Figure 4.3b.

When a cluster fails to expand to its NN, it is considered complete. Since the nearest neighbor sharing the same partial textual description is the most related, no other object may be more relevant for expansion. Continuing the
example in Figure 4.6, the cluster of four will not include any of the five left-
most objects. Intuitively, if the cluster should be expanded, there would be 
objects in that direction with shorter distance than the distance to the outlier o₅. Therefore, when we encounter a too low quality the first time, we stop 
and avoid over-expansion of the cluster.

Figure 4.6: Sample Dataset where Cluster Expansion is Stopped

CLUSTO Implementation

We proceed to describe the implementation of CLUSTO. The following 
algorithms integrate the selection of candidate annotations, nearest-neighbor 
chaining, and the stopping rule based on the proposed merged-quality func-
tion. The algorithms are designed to run on a large dataset of spatio-textual 
objects (such as geo-tagged microblog posts) for effective PoI identification.

Clustering Algorithm

Algorithms 17–9 show the clustering approach implemented in CLUSTO. The 
entry point is Algorithm 17 with the global parameters configured at the 
beginning (Lines 2–4). The for loop iterates through all candidate annotations. 
The annotations are sorted descendingly by the length of their textual descrip-
tion so that the most descriptive/specific (i.e., longest, probably containing 
multiple terms) annotations are considered first. Each annotation is passed 
to Algorithm 8 that performs cluster expansion.

Algorithm 7: CLUSTO(Dataset D)

\begin{verbatim}
1 /* Global variables: */
2 minSize ← minimum cluster size; // config. parameter
3 minQ ← quality threshold; // config. parameter
4 V ← dictionary to store clustering; // {string→clusters}
5 sort D by text length in descending order;
6 foreach Annotation a ∈ D do
7   V[a] ← ExpandCluster(a, D);
8 return V;
\end{verbatim}

Algorithm 8 is responsible for expanding all possible clusters for the given 
annotation. It starts by initializing five working variables (Lines 1–5): one
stack \( (A) \), two lists of clusters \((R \) and \( E \)) , and two sets of objects \((D_a \) and \( V_a ) \). The stack maintains clusters that may be further expanded when an RNN is found. The two lists of clusters contain the clusters that are no longer considered for expansion. Thus, the clusters in the list \( R \) are completed and may be returned at any time. The set \( D_a \) stores all candidate objects for clustering with the given annotation, i.e., objects with textual description containing \( a \). The set \( V_a \) stores objects that already formed valid clusters containing the given annotation.

\[
\text{Algorithm 8: } \text{ExpandCluster}(\text{Annota. } a, \text{ Dataset } D)
\]

\begin{verbatim}
1 A ← stack of active clusters;
2 R ← list of approved clusters;
3 E ← list of rejected clusters;
4 D_a ← \{o ∈ D | o.text contains a\};
5 V_a ← \{o ∈ c | c.a contains a, c ∈ V\};
6 \textbf{while} \( D_a \) contains non-visited objects \textbf{do}
7    select random unvisited object, \( o ∈ D_a \);
8    mark \( o \) as visited in \( D_a \);
9    push \( \{o\} \) to \( A \); // as a single-object cluster
10   \textbf{while} \( A \) is non-empty \textbf{do}
11      \( c ← \text{pop cluster from } A; \)
12      \( c_{nn} ← \text{NN cluster of } c \text{ in } D_a \setminus (c ∪ V_a); \)
13      mark \( c_{nn} \) as visited;
14      \textbf{if} \( c_{nn} \) is \textbf{null} \textbf{or} \( c_{nn} ∈ (E ∪ R) \) \textbf{then}
15         \texttt{FinishCluster}(c, A, R, E);
16         continue; /* Stop expanding - no more valid NNs */
17      \textbf{if} \( c_{nn} ∈ A \) \textbf{then} /* RNN pair found
18         \textbf{if} \( \text{mergedQ}(c, c_{nn}, a) ≥ \text{minQ} \) \textbf{then}
19            remove \( c \) \text{ and } c_{nn} \text{ from } A;
20            push \( (c ∪ c_{nn}) \) to \( A; \) // merge and push
21         \textbf{else} /* Stop expanding - the quality is too low
22            \text{FinishCluster}(c, A, R, E);
23            \text{FinishCluster}(c_{nn}, A, R, E);
24         \textbf{else} /* not RNN
25            push \( c_{nn} \) to \( A; \)
26   \textbf{return} \( R; \)
\end{verbatim}

After the initialization, a while loop follows that iterates through all unvisited objects containing the given annotation. Each object is randomly selected, marked as visited, and pushed to the stack of active clusters (Lines\textsuperscript{7-9}). Then,
the processing continues in the inner while loop until the stack of active clusters is empty (Lines 10–26).

The loop follows the logic behind NN-chaining, taking into account the stopping condition. A cluster \( (c) \) is popped out from the stack, and NN search is performed (Line 13). We use the Euclidean distance calculated from an object in \( c \) to its NN object. If the found NN object does not belong to any cluster, \( c_{nn} \) stores a single-object cluster. Otherwise, \( c_{nn} \) stores the cluster that the found object belongs to. Note that NN search excludes the objects from already formed clusters containing the same annotation \( (V_a) \).

Since longer annotations are considered first, later clustering with a shorter sub-annotation does not reuse the same objects. For example, objects that formed “Central Park” are not reused when clustering “Park.”

Next, the algorithm performs NN-chaining, taking into account the stopping condition based on the quality function. The NN of \( c \) is examined as follows. If no NN to \( c \) can be found or the found NN belongs to the list of previously rejected or accepted clusters (Line 14), the expansion of \( c \) is stopped by calling Algorithm 9 (Line 16), which finalizes the clustering of \( c \) (more on that in a moment). If the found NN and \( c \) are reciprocal (Line 18), the merged quality of the clusters is computed and compared to threshold \( minQ \) (Line 19). If the quality is satisfactory, \( c \) is expanded with its NN cluster by removing them from the stack of active clusters (Line 20) and pushing the newly merged cluster instead (Line 5). As such, the pushed cluster may be further expanded in the next iteration. Otherwise, the expansion of both \( c \) and its NN are finalized by calling Algorithm 9 as before (in Lines 23 and 24, respectively).

Finally, the found NN may not be an RNN to \( c \). In this case, it is simply pushed on the stack of active clusters (Line 26), implying that the search of its RNN is performed in the next iteration.

Algorithm 9 performs the final steps for the given cluster. First, if the cluster is large enough, it is added to the list of approved clusters \( (R) \). Otherwise, it is added to the list of rejected clusters \( (E) \). In either case, all the objects in the cluster are marked as visited. This implies that the objects are no longer considered for clustering, as they formed or failed to form a cluster under the given annotation. Lastly, the cluster is removed from the stack of active clusters \( (A) \).

**Algorithm 9: FinishCluster**(Cluster \( c \), Stack \( A \), List \( R \), List \( E \))

1. if \( |c| \geq minSize \) then
2. \[ R \leftarrow R \cup c; \]
3. else
4. \[ E \leftarrow E \cup c; \]
5. remove \( c \) from \( A \);
Complexity Analysis

Each object from $D_a$ is added only once to the stack of active clusters (Line 9), since it is marked as visited on entry, and, when it is removed it is either added to the list of approved or rejected clusters. Therefore, it may never enter the stack again as seen from Lines 7 and 14. This gives the space consumption in the stack of $O(|D_a|)$, for each annotation in $D$. The NN-chain is grown from the most recent cluster in the stack, and a cluster may merge with as many as $|D_a| - 1$ clusters. This gives a run-time of $O(|D_a|^2)$, for each annotation in $D$.

In the following example, we walk through the entire clustering process in CLUSTO.

Example 9. We run the algorithm with the 8 objects seen to the right in Figure 4.6 as the dataset and set the minimum quality threshold, $minQ$, to 0.5 and the minimum cluster size, $minSize$, to 2. We assume that the $\Box$ annotation is longer and thus select a random starting point $o_2$ (in Algorithm 8, Line 7). As such, the first single-object cluster is pushed to $A$.

$$A = \{\{o_2\}\}$$

In Line 13, we find that the NN to $o_2$ is $o_1$. Since $o_1$ does not exist in $A$, it is pushed to $A$ too (Line 26).

$$A = \{\{o_2\}, \{o_1\}\}$$

Next, the NN to object $o_1$ is object $o_2$. Since $o_2$ exists in $A$, they are RNNs, and the merged quality is computed (Line 19). The quality of the two objects is 1, since no other ClusterEnclosed objects exist. The quality exceeds the threshold, and the objects are merged and added to $A$.

$$A = \{\{o_1, o_2\}\}$$

The expansion continues:

$$A = \{\{o_1, o_2\}, \{o_3\}\}$$
$$A = \{\{o_1, o_2\}, \{o_3\}, \{o_4\}\}$$
$$A = \{\{o_1, o_2\}, \{o_3, o_4\}\}$$
$$A = \{\{o_1, o_2, o_3, o_4\}\}$$
$$A = \{\{o_1, o_2, o_3, o_4\}, \{o_5\}\}$$

The process stops when $o_5$ becomes the RNN of the cluster of the other four objects. The merged quality is $mergedQ(\{o_1, o_2, o_3, o_4\}, \{o_5\}, \Box) = 0.25$, which is below the threshold $minQ = 0.5$. Thus, the clusters are not expanded any further as seen from the two calls to Algorithm 9. The cluster of four is added to the set of approved clusters, $R$, while the cluster consisting of $o_5$ is added to the list of rejected clusters, $E$.

The order in this example follows the order of the dendrogram in Figure 4.5a. Had $o_3$, $o_4$, or $o_5$ been the random starting point, the order would have followed the dendrogram in Figure 4.5b, but it would have produced the same clustering. □
4.4 Experimental Evaluation

We empirically evaluate CLUSTO’s effectiveness at discovering points of interests using a real dataset and compare it against the DBSCAN algorithm. First, we present our experimental setup. Then, we describe the results of a number of experiments.

Experimental Setup and Data

We use a commodity machine with a quad-core Intel Core i5-2520M (2.5 GHz) processor and 8 GB of main memory. In all experiments, the data is loaded into PostgreSQL with geo-spatial indexes capable of measuring distances on the Earth’s surface. All data in the database is stored on disk. The proposed solution is implemented in a single-threaded Java application. All possible annotations are pre-processed and stored in main memory using an inverted file index (a mapping from textual annotation to the corresponding object identifiers in the database).

To evaluate the quality of the computed clusters, we utilize the Google Places API\footnote{http://developers.google.com/places/} that enables us to retrieve place information (including points of interests) within a given geographic region.

We collected all geo-tagged messages from the public Twitter FireHose in the period from September, 2012 to November, 2013. For this study, we extracted all tweets issued within a 6 km\(^2\) down-town region in San Francisco, USA. In our micro-benchmarks, we found the region to be very well covered by Google Places. The dataset contains 285,173 geo-tagged tweets in total. To accommodate the inaccuracy of GPS, we assume an object to be located anywhere in a 10 meter radius of the point reported by the Twitter FireHose.

In the following, we conduct three sets of experiments and discuss the results. First, we test how CLUSTO configuration parameters, i.e., \(\minQ\) and \(\minSize\), affect clustering quality. This enables us to choose an optimal configuration for the subsequent experiments. Second, we measure the runtime of CLUSTO. Third, we compare our approach against DBSCAN.

Varying \(\minQ\) and \(\minSize\)

In the first set of experiments, we study how varying \(\minQ\) and \(\minSize\) impact clustering in CLUSTO. Intuitively, the closer \(\minQ\) is to zero, the more clusters are formed. However, the formed clusters are of poor quality because more noise objects are allowed to be in each cluster. On the other hand, the closer \(\minQ\) is to 1, the higher the quality is of the clusters that are created. However, setting it too high can result in missing valid clusters. Choosing a value for \(\minSize\) involves a similar trade-off.
4.4. EXPERIMENTAL EVALUATION

Cluster Coverage

We start with evaluating the coverage of the discovered clusters. We make use of the Google Places API as follows. Initially, a random object is fetched from Google Places within the region considered. Google Places may return road names or other objects that are not considered as points of interest. To prevent these type of places, we require objects to have one or more ratings.

Then, we search among the clusters of CLUSTO to see if a similarly annotated cluster exists. Google Places objects have specific coordinates, while ours may span a larger region. To accommodate for imprecision in Google Places or in user locations, we allow a search radius of 25 meters from the Google Places object. We report a match if a cluster exists within the search radius that has an annotation that contains one or more similar terms. For example, if a randomly selected object from Google Places has the name 'Hotel Tomo' and CLUSTO detected a cluster with either of the annotations 'hotel,' 'tomo,' or 'hotel tomo,' it is considered a match. We remove all stop words before the matching.

For each clustering, we fetch 100 random places from Google Places and perform the above matching. Figure 4.7 shows the results. As expected, more matches are found when more noise is allowed, i.e., with minQ values closer to 0. However, a weaker requirement implies that false clusters might be created. The matching decreases from almost 80% to less than 10% when increasing minQ from 0.1 to 0.9. This includes a sharp decrease when minQ > 0.5. In contrast, varying minSize has just a slight impact on the matching.

![Figure 4.7: CLUSTO Matches of Google Places](image)

To show whether CLUSTO discovered clusters exist in Google Places, we perform an “opposite” matching, too. That is, we randomly select 100 CLUSTO clusters and search (using their annotations) for analogous PoIs in

---

3We verified that this procedure yields a PoI in the region.
Google Places. The same setting (search radius, term matching) as above is used for the matching. The results are shown in Figure 4.8 (the value of \( \min Q = 0.9 \) is omitted as not enough clusters are formed for all settings). In contrast to the previous experiment, the impact of \( \min Q \) is smaller, while the impact of \( \min Size \) is bigger. This indicates that setting \( \min Size \) too low may result in more false clusters. CLUSTO achieves a relatively high and stable 80–85% matching unless very poor clusters are allowed. That is, only with \( \min Q = 0.1 \) and \( \min Size = 2 \), the matching drops below 65%.

In the following experiments, we fix the value of \( \min Size \) to 5 since it demonstrates good results for both experiments.

Cluster Properties

In this experiment, we consider how the characteristics of the discovered clusters change when \( \min Q \) is varied. We ran the experiment on the same dataset (containing 285,173 tweets) and analyzed the formed clusters with the following five popular annotations: “hotel,” “food,” “park,” “coffee,” and “restaurant.” The number of posts mentioning the popular annotations are given in Table 4.2.

<table>
<thead>
<tr>
<th></th>
<th>hotel</th>
<th>food</th>
<th>park</th>
<th>coffee</th>
<th>restaurant</th>
</tr>
</thead>
<tbody>
<tr>
<td>posts</td>
<td>5273</td>
<td>3591</td>
<td>2641</td>
<td>2484</td>
<td>2306</td>
</tr>
</tbody>
</table>

Table 4.2: Number of Objects in the Dataset

First, we examine the changes in the spatial extent of the clusters by measuring their maximum diameter. Recall that a too low value of \( \min Q \) can result in over-expanding the clusters. For example, if two “hotels” are
4.4. EXPERIMENTAL EVALUATION

Figure 4.9: Cluster Properties

(a) Sizes of the Clusters

(b) Number of Clusters

(c) Runtime of CLUSTO
located within close proximity, they can be merged into one. Therefore, while we still can have a match of “hotel” in Google Places, the clustering does not provide accurate information about the region. Figure 4.9 shows the results. As expected, the diameters of the clusters decrease when $\minQ$ approaches 1. However, for the clusters that are expected to be larger (i.e., “park”), CLUSTO is able to perform relatively stable (the diameter remains large) also with higher values of $\minQ$. A cluster with the maximum diameter of 1 meter corresponds to a precisely located cluster (recall that we assume a GPS inaccuracy of 10 meters).

Next, Figure 4.9b shows the total number of clusters formed with each of the five annotations. The numbers decrease as expected for all annotations as $\minQ$ approaches 1, due to the stronger requirement for forming clusters. We notice that for values close to 0, more clusters are created in close proximity to clusters with similar annotations. For example, a cluster with annotation "park" may have a cluster with the same annotation in close proximity, separated by a number of noise objects. This is the result of the weaker requirement that may introduce more clusters, but may not always be able to merge the clusters when too many noise objects exist.

As seen from Figure 4.9a and 4.9b, a fixed value of $\minQ$ is suitable for different annotations. The diameter and number of clusters formed generally decreases for all annotations when $\minQ$ approaches 1.

Runtime

We continue with the five popular annotations and measure the runtime of each clustering. CLUSTO runs offline and only once for a dataset, making the runtime less critical. The results are shown in Figure 4.9c. The runtime of a particular clustering depends on the number of candidate objects and roughly follows the order given in Table 4.2. As such, “hotel” with most objects (5,273) takes the longest to process. Although “park” has slightly more objects than “coffee,” clustering the former is faster than clustering the latter. We notice that this is because “park” objects are less spread in the region and are closer to each other than are “coffee” objects. Also, the running time tends to decrease for all annotations when $\minQ$ increases. This is because the higher $\minQ$ is set, the earlier the stopping condition is triggered in each clustering. This is especially true for the numerous “hotel.”

Annotation Properties

The proposed solution prioritizes the processing of longer annotations for two reasons. First, it makes the entire clustering run faster because each call to Algorithm 8 operates on smaller candidate sets. That is, the number of objects sharing the same longer (more specific) annotation (e.g., “Central Park”) are likely to be smaller than the number of objects sharing the same
shorter (more general) annotation (e.g., “Park”). As such, the processing of longer annotations is faster. Also, since the objects that successfully form a cluster are not reused, the candidate sets for shorter annotations shrink, making their processing faster as well. Second, longer annotations are likely to describe PoIs better. CLUSTO is therefore able to report relevant PoIs to the user as soon as it starts running.

In this experiment, we vary $\minQ$ and count the number of terms in each annotation forming a cluster. The results are shown in Figure 4.10. When $\minQ$ is set to 0.1, the majority of the annotations have 1 or 2 terms. This is because the low threshold allows more noise, and it is more likely that users share single terms rather than multiple. When $\minQ$ increases to 0.3, 0.5, or 0.7, the number of terms in each annotation increases. Thus, single-term clusters are more likely to contain noise and thus are rejected by the stronger requirement. In general, the results confirm that CLUSTO is able to detect multi-term annotated clusters representing more descriptive PoIs. The results for $\minQ = 0.9$ are omitted because not enough clusters are formed.

![Figure 4.10: Distribution of Number of Terms in Annotations](image)

With the results so far, we propose a good value of $\minQ$ for the current dataset. First, we want to avoid over-expansion, reducing noisy clusters, and to have the most descriptive annotations. Therefore, we avoid the value 0.1. Second, we want to maximize the number of clusters. Thus, we propose to use either 0.3 or 0.5. With a value of 0.3, we get more clusters and more matches with Google Places while preserving a reasonable size of the clusters.

**Popular Annotation Properties**

In this experiment, we fix $\minQ$ at 0.3 and report the number of terms each of the five popular annotations contains. Naturally, the number of terms vary greatly for each annotation as seen in Figure 4.11. The names of the
places and how users mention them affect the clustering. For example, when the majority of the users mention "food" without any other common term, most of the resulting annotations have only this single term. The annotation "hotel" most often occurs with one additional term, and "restaurant" is most often combined with two additional terms.

![Graph showing number of terms in popular annotations](image)

**Figure 4.11: Number of Terms in Popular Annotations**

**CLUSTO versus DBSCAN**

In the last set of experiments, we compare the clustering of CLUSTO with the clustering of DBSCAN [31]. In CLUSTO, we fix $minSize$ and $minQ$ to 5 and 0.3, respectively.

Similarly, DBSCAN requires a minimum number of points ($MinPts$) in a cluster, which we set to 5. Also, it requires a distance threshold parameter ($\epsilon$). Setting it too high may result in over-expanded clusters, while setting it too low may miss larger clusters. Based on the previous experiments, we know that cluster sizes vary and can reach 50 meters in diameter. To ensure the capability to form clusters with this diameter, the value of $\epsilon$ must be equally high. However, to avoid over-expansion, which we show in the following experiment, we choose a relatively low value of 25 meters, to make the comparison fair. Note that with this low value, DBSCAN is not be able to combine objects located more than 25 meters apart.

The resulting clustering for “hotel” is shown in Figure 4.12. It is clear from Figure 4.12a that DBSCAN over-expands clusters when the data is dense. As a result, the formed clusters do not correspond to the actual points of interest. CLUSTO, on the other hand, provides quite accurate clusters as seen from Figure 4.12b.

DBSCAN does not take into account the textual attribute of other nearby objects, and it performs clustering based purely on spatial density. As a re-
4.4. EXPERIMENTAL EVALUATION

(a) DBSCAN, $\epsilon = 25 \text{ m}$

(b) CLUSTO, $\min Q = 0.3$

Figure 4.12: Clustering for 'hotel'

Result, it may form clusters with annotations that are not dominating in the region. This effect is clearly visible in Figure 4.13a. DBSCAN finds 'park' clusters where there are obviously no parks. CLUSTO takes into account the textual attribute and expands the clusters based on the merged quality function. Therefore, it forms clusters only with annotations that are dominating in the region. In Figure 4.13b, we see that no false “park” points of interests are formed.

Finally, we compare the maximum diameters of the clusters in Figure 4.14. We use the five popular annotations given in Table 4.2. The diameters of the clusters formed by DBSCAN are up to two orders of magnitude larger than those of CLUSTO. While a 200 meter diameter for “park” clusters may be rea-
reasonable, a 1,000 meter diameter for clusters “hotel” and “food” are unrealistic. In contrast, CLUSTO provides meaningful diameters for all annotations.

![Figure 4.14: Sizes of the Clusters](image)

### 4.5 Related Work

Recently, substantial research efforts have explored various research directions related to microblogs. This includes indexing of microblogs [10, 18, 93, 95], event detection from microblogs [67, 68, 78], news extraction from microblogs [80], microblog posts based recommendations [40] and decision making [11] systems, microblog ranking [30, 86], visualization [66], and spatio-temporal aggregation [81].

Despite such rich research on microblogs, to the best of our knowledge, no existing work that addresses the discovery of points of interests using geotagged microblog posts, which is the focus of this paper. However, the paper’s proposal is closely related to spatial clustering.

**Spatial Clustering.** For decades, there has been extensive research on clustering algorithms and their applications in many areas [50]. Our work is closely related to density-based partitioning algorithms [51]. In these, a set of data objects form a cluster if they are spread in the data space over a contiguous region of high object density. Density-based algorithms aim to identify all such dense regions that are separated by low-density regions. Data objects located in low-density regions are considered as noise. Well-known examples of such algorithms include DBSCAN [31], its extension GDBSCAN [79], and DENCLUE [44]. The major advantage of these algorithms over the classic partitioning approaches (e.g., $k$-means [62] or CLARANS [73]) are that they do not require the number of clusters as an input parameter ($k$), which would be a crucial limitations in our setting, too. To apply density-based clustering to our setting, i.e., to take into account the textual attribute, the algorithm has to be run for each text annotation separately. Thus, density-based clus-
tering does not consider objects with other text annotations, which may result in over-expansion of clusters. Further, density-based is limited by the search range, which on the other hand may result in under expansion of clusters. Both cases are evaluated in Section 4.4.

In CLUSTO, the proposed quality function takes into account all nearby objects, and an object is added to a cluster only if their merged quality is not below the given threshold. Although similar to a minimum density (defined by $\epsilon$ and $\text{MinPts}$ in DBSCAN), our approach, roughly speaking, expands a cluster as long as the annotation is dominating in the region. Thus, clusters of varying density can be formed, which is not possible with density-based clustering.

To choose the next candidate for a cluster, CLUSTO borrows techniques from agglomerative clustering based on nearest neighbor (NN) chains [72]. An NN-chain consists of an arbitrary cluster, followed by its NN, which is again followed by its NN from among the remaining clusters, and so on. Such an NN-chain ends in a mutual or reciprocal NN (RNN) pair, i.e., a pair of clusters $c_1$ and $c_2$ such that the NN of $c_1$ is $c_2$, and vice versa. We found RNN clustering particularly efficient in our setting for two reasons. First, the merged RNN pair does not affect the remaining chain members, and thus can be reused for the subsequent agglomerations. Second, the spatial NN search can be accelerated using spatial indexing.

4.6 Conclusions and Future Work

We present a spatio-textual clustering method for the discovery of points of interest from geo-tagged microblog posts. The method takes into account both spatial proximity and textual relevance and is able to form clusters of arbitrary shape and density. A proposed merged cluster quality function serves as a criterion for cluster expansion and, in combination with nearest-neighbor chaining, prevents over-expansion of clusters. An experimental study with real data offers insight into the properties of the resulting clusters; and it demonstrates that the method is able to extract accurate and comprehensive PoIs in a realistic, real-world setting.

Since all microblog posts are timestamped, interesting future work includes adding a temporal dimension to the spatio-textual clustering. This may enrich the points of interest further, e.g., by labeling them with inferred opening hours. Also, in the proposed solution, we perform simple preprocessing of the textual description of objects. This may be extended using more advanced natural-language processing techniques.

Acknowledgments We thank the reviewers for their helpful comments. The research was supported in part by the Danish National Research Foundation.
grant DNRF84 through Center for Massive Data Algorithmics (MADALGO) and by a grant from the Obel Family Foundation.
Chapter 5

Scalable Top-k Spatio-Temporal Term Querying

Abstract

With the rapidly increasing deployment of Internet-connected, location-aware mobile devices, very large and increasing amounts of geo-tagged and timestamped user-generated content, such as microblog posts, are being generated. We present indexing, update, and query processing techniques that are capable of providing the top-k terms seen in posts in a user-specified spatio-temporal range. The techniques enable interactive response times in the millisecond range in a realistic setting where the arrival rate of posts exceeds today’s average tweet arrival rate by a factor of 4–10. The techniques adaptively maintain the most frequent items at various spatial and temporal granularities. They extend existing frequent item counting techniques to maintain exact counts rather than approximations. An extensive empirical study with a large collection of geo-tagged tweets shows that the proposed techniques enable online aggregation and query processing at scale in realistic settings.

5.1 Introduction

The digital universe is expanding exponentially, and the amount of text content available on the web, including news items, web pages, and microblog posts, grows rapidly. Today, everyone can generate such content. In particular, services such as Twitter, Facebook, and Blogger make it easy for all to contribute.

Users of social media services often generate similar content in response to events that catch their attention. For example, when natural disasters occur, multiple users are likely to report on this independently. Some users may
discuss evacuations and the traffic situation, while other users may simultane-
ously discuss unrelated topics such as food recipes or a sporting event. With
thousands of pieces of content being made available each second, techniques
are needed to maintain an overview of what occupies minds of users.

Proposals existing that are capable of finding currently popular topics in
streaming text content [67, 68]. We extend these by supporting also queries
on past data and by allowing spatio-temporal range queries. By supporting
queries that retrieve the top-k most frequent terms in content in user-specified
spatio-temporal regions, we support different services that aim to find the
“word in the street” or “talk of the town” in some time period. To illustrate,
Figure 5.1a shows a few tweets located in a query region that covers part of
New York City during some time period. The most popular terms can be
represented by a tag-cloud, as shown in Figure 5.1b.

In a real-world setting, spatio-temporal regions may contain millions of
tweets.

A key design goal for the techniques needed to support this functionality
is scalability. Specifically, we seek a solution that is capable of supporting the
entire world and a long history, as well as supporting a stream with many more
posts per second than what the average of some 3,000 tweets per second that
Twitter sees (and of which only a fraction is geo-tagged). This goal is set to
achieve a future-proof solution that anticipates the rapid growth in geo-tagged
posts.

Index structures, e.g., based on the R-tree, have been proposed to support
spatio-temporal queries. However, these are not well suited for our setting
with a rapid content stream and spatio-temporal aggregate queries [89]. We
thus propose new indexing techniques that aim to support efficient querying of
streaming and historical data. We extend a popular algorithm that maintains
static-sized summaries of the most frequent items in data streams to support
dynamic, variable-sized summaries so it can adapt to diverse data. Further,
we provide a new merging algorithm for these summaries.

The paper makes four contributions. First, we provide techniques capable of supporting a holistic aggregate function \[37\]: the top-\(k\) most frequent terms. We propose a variant of frequent item counting that maintains top-\(k\) most frequent terms error-free. This allows us to combine aggregates as needed to support arbitrary spatio-temporal regions with low probability of introducing errors, and it allows us to provide correctness guarantees.

Second, we provide techniques that maintain and use spatio-temporal aggregates that grow and shrink according to the activity (the number of distinct items to aggregate) and the required \(k\). The aggregates are partially persistent, i.e., all versions can be accessed, but only the current version can be modified. Once aggregated, the top-\(k\) terms are compact in comparison to the currently active aggregates.

Third, we integrate the above techniques into the Adaptive Frequent Item Aggregator (AFIA) system that supports spatio-temporal top-\(k\) frequent term queries. The finest spatial granularity supported is \(100 \times 100 \text{ m}^2\), as finer granularities make little sense in our setting, and the finest temporal granularity is an hour.

Fourth, we study AFIA empirically using a real-world data stream from Twitter. On server class machines, AFIA supports a stream with more than 13,000 tweets per second. One month of aggregates requires up to 120 GB of memory. The accuracy depends on the number of precomputed aggregates needed to answer a query. Since we maintain aggregates at multiple spatial and temporal granularities, the lowest observed accuracy was 97%. The study suggests that AFIA meets the scalability design goal.

The remainder of the paper is structured as follows. Section 5.2 defines the problem and covers related work. Section 5.3 presents the indexing and aggregation framework. An experimental evaluation is given in Section 5.4. Finally, we conclude in Section 5.5.

### 5.2 Preliminaries and Related Work

#### Problem Definition

Let \(D\) be a set of spatio-temporal web objects \(o = (\lambda, \text{doc}, ts)\), where \(\lambda\) is a point location in 2D Euclidean space, \(\text{doc}\) is a text document, and \(ts\) is a time point. The score of a term \(t\) for a set \(S\) of objects, \(\text{score}(t, S)\), is the count of objects in \(S\) whose text document contains \(t\):

\[
\text{score}(t, S) = |\{o \in S \mid t \in o.\text{doc}\}|
\]  

(5.1)

A top-\(k\) most popular terms query \(q = (R, k, I)\) returns a pair consisting of (i) \(k\) terms from objects in \(D\) that belong to the spatio-temporal range defined by 2D rectangle \(R\) and time interval \(I = [t_s, t_e]\) and (ii) an integer \(k_g \leq k\). The
first $k_g$ terms are guaranteed to have the highest scores for the set considered, and the remaining $k - k_g$ terms are approximate. When $k_g = k$, the result is exact.

The query is designed to take advantage of the setting, where exact results are not necessary, in order to achieve high performance. We aim to compute results were $k_g$ is close to $k$ and where the $k - k_g$ unguaranteed terms either are top-$k$ terms or are close to being so.

**Example 10.** Consider six objects with these documents:

$s_1$ = “Hurricane Sandy causes evacuation of NYTMetro.”  
$s_2$ = “NYC under water.”  
$s_3$ = “NYTMetro not running.”  
$s_4$ = “NYTMetro down because of sandy.”  
$s_5$ = “Sandy Evacuation in New York.”  
$s_6$ = “Flooding due to the storm.”

Ignoring word case and so-called stop words (in grey color), 17 terms are considered in total. An exact query with $k = 3$ on these objects yields the result $\{\text{sandy, nytmetro, evacuation}\}, 3$.

---

**Aggregation**

Aggregation is an important kind of operation and has been studied widely. In OLAP settings, it is typical to precompute intermediate aggregation results and then to reuse these for computing higher level results. The choice of which intermediate results to precompute is based on frequently-asked queries and apriori known data groupings and hierarchies [6, 37, 42]. In stream processing, running aggregates of stream data may be maintained so that up-to-date summaries of the data are readily available (e.g., [61]).

In a spatio-temporal setting, Papadias et al. [74] propose the aggregation R-tree (aR-tree) that augments each node with an aggregate value (e.g., count) for all objects in the node’s subtree. Then, during querying, there is no need to perform the aggregation for objects in MBRs that are covered by the query region. Mamoulis et al. [63] show how using the aR-tree to support top-$k$ queries. Yang and Widom [94] propose the SB-tree that maintains a hierarchy of intervals associated with partially computed aggregates.

There are three types of aggregation functions [37]: distributive, algebraic and holistic. Distributive (e.g., count, sum, max) and algebraic (e.g., avg) aggregates can be computed by further aggregating intermediate aggregates, while holistic aggregates cannot. The pre-aggregation techniques described above work only for non-holistic aggregation functions, while the top-$k$ query considered in this paper is holistic. We are not aware of any existing spatio-temporal grouping techniques that support holistic aggregates.
5.2. PRELIMINARIES AND RELATED WORK

Frequent Item Counting

The aggregation of frequent items in data streams has also been studied widely. The $\phi$-frequent items in a set $S$ are $\{i \in S \mid f_i > \phi |S|\}$, where $f_i$ is the frequency of an item $i$ in $S$. In Example 10, $f_{\text{sandy}} = 3$, $f_{\text{nytmetro}} = 3$, $f_{\text{evacuation}} = 2$, $f_{\text{hurricane}} = 1$, etc. For $\phi = 0.1$, the frequent items are $\{\text{sandy}, \text{nytmetro, evacuation}\}$. Since solving the $\phi$-frequent problem requires linear space ($\Omega(|S|)$), an approximate version is often considered: the $\epsilon$-approximate frequent items in a set $S$ are $\{i \in S \mid f_i > (\phi - \epsilon)|S|\}$.

Approaches to frequent item counting can be categorized as counter-based and sketch-based. The former monitor the items in a stream using a fixed number of counters. If an item monitored by a counter arrives, its count is incremented. Otherwise, depending on the algorithm, the item is ignored, or it takes the place of a currently monitored item. Notable counter-based algorithms include LossyCounting [65], Frequent [28, 49, 71], and SpaceSaving [70].

Sketch-based approaches maintain the approximate frequency for all items using hashing. Items are mapped into a (smaller) space of counters, and a hashed-to counter is updated for every occurrence of a corresponding item. Notable sketch-based algorithms include CountSketch [17], GroupTest [26], and Count-Min Sketch [25].

Because the sketch-based approaches provide no guarantees about the relative order of items or their estimated frequency and also suffer from much higher per item processing costs, we turn to counter-based techniques. In particular, we adapt the SpaceSaving algorithm. In addition to finding a set of frequent items, this algorithm also finds the top-$k$ items [70]. It can also provide relative order guarantees for individual items because each counter value is associated with its maximum error. Further, our micro-benchmarks as well as recent experimental evaluations [24, 64] find that SpaceSaving is a top performer.

SpaceSaving is outlined in Algorithm 10. Each monitored item $i$ is associated with its count, $c_i$, and a maximum error, $\Delta_i$. The counters are initialized by the first $m$ distinct items, and their exact counts (Lines 3–8). When an un-monitored item arrives, it replaces a currently monitored item with the lowest count. As Lines 10–13 indicate, the new item gets the maximum possible count and records its maximum error in $\Delta_i$. Intuitively, frequent items reside in counters with large values and will not be hit by infrequent items that grow more slowly. SpaceSaving is guaranteed to capture all $\epsilon$-frequent items with $m = 1/\epsilon$ number of counters and the maximum counter error is $\lfloor n/m \rfloor$ [70].

SpaceSaving can provide three levels of guarantees. First, it can report only items that are guaranteed to be $\epsilon$-approximate frequent: report a monitored item $i$ if $c_i - \Delta_i > \phi n$. Second, it guarantees that the first $k$ items are
Algorithm 10: SpaceSaving(counters $m$, stream $S$)

1. $T \leftarrow \emptyset$; // monitored items
2. foreach $i \in S$ do
3.      if $i \in T$ then
4.          $c_i \leftarrow c_i + 1$;
5.      else if $|T| < m$ then
6.          $T \leftarrow T \cup \{i\}$;
7.          $c_i \leftarrow 1$;
8.      else
9.          $j \leftarrow \min_{j \in T} c_j$; // item with min count
10.         $c_i \leftarrow c_j + 1$; // new count
11.         $\Delta_i \leftarrow c_j$; // max error
12.         $T \leftarrow T \cup \{i\} \setminus \{j\}$;

the top-$k$ most frequent items if $\forall i \leq k (c_i - \Delta_i \geq c_{k+1})$. Third, it guarantees that the top-$k$ items are correctly ordered if $\forall i \leq k (c_i - \Delta_i \geq c_i + 1)$.

We extend SpaceSaving to support exact top-$k$ ordered queries. To support arbitrary spatio-temporal regions, we rely on the ability to merge aggregated top-$k$ terms across multiple spatio-temporal granularities. In doing so, we rely on the notion of mergeability of data summaries introduced by Agarwal et al. [1]. They show that frequent item aggregates computed using Frequent (or SpaceSaving) are mergeable and preserve space and error guarantees. Our SpaceSaving extension and its accompanying merging algorithm preserve exact query results. We achieve this by trading the space (memory) for being error-free in an adaptive manner.

Related Systems

Several recent systems target social media streams. BlogScope [5] (and its commercial counterpart, Sysomos) collects text documents from news feeds, mailing lists, forums, newsgroups, blogs, etc. Each document is temporally ordered and is associated with the profile of its author, which may include a location. The system supports the discovery and tracking of real-world entities (stories, events, etc.) [4], the monitoring of most popular keywords (“trends”) [65], and the monitoring of temporal and/or spatial bursts [67]. However, the system does not aggregate keywords according to user-specified spatio-temporal regions. Also, only currently popular keywords are identified (with a span of up to few minutes), whereas our system supports also queries on the past.

NewsStand [85], a spatio-textual news aggregator, extracts geographic content from RSS feeds and groups articles into story clusters. Users can then
5.3. PROPOSED SOLUTION

We present a framework that maintains spatio-temporal grids at multiple granularities that partition Earth and time. A precomputed summary is maintained for each grid cell. The sizes of the summaries adapt dynamically to the data, and a summary merging algorithm enables query processing.

Adaptive Frequent Item Aggregator

We proceed to describe the multi-granularity index structure and then present an overview of the framework.

Grid-Based Indexing of Precomputed Summaries

Large volumes of queries combined with large spatio-temporal query regions can lead to the need for aggregating billions of messages. To achieve interactive response times, we introduce an index structure that supports pre-aggregation at multiple granularities. This way, a user-specified spatio-temporal range query can be partitioned into a collection of coarsest spatio-temporal regions for which aggregates are available, and these aggregates can then be combined efficiently to produce the query result.

We employ a static grid-based approach that uses uniform grids [2] with predefined and fixed cell sizes. By introducing multiple layers of grids with
increasing cell sizes, we partition space at multiple granularities. By precom-
puting information about the most frequent terms in each grid cell, queries
with regions of any size larger than the finest granularity cell can be efficiently
supported. By indexing the information precomputed for each cell in a hash
table, this approach supports constant-time cell lookup.

Therefore, to efficiently support queries with different spatial extents, we
divide the world into a grid with multiple granularities as seen in Figure 5.2.
The lowest granularity defines the spatial accuracy of our summaries (which
translates to GPS accuracy, for instance). Precomputed summaries are as-
sociated with each cell at each granularity. Coarse granularities are used to
more efficiently support queries with large regions as they obviate the need
for many summaries at finer granularities.

Frequent Term Counting

Each lowest-granularity cell covers a small part of the world, and different
cells may receive very different amounts of objects. Also, the terms in objects
may vary substantially, e.g., because users use different languages. Thus, any
given cell may see a large number of terms that may not be used in other cells.
To store, for each cell, each term in list, e.g., ordered by frequency, requires
linear space and is infeasible for large-scale processing.

Instead, we keep track only of the most frequent terms in each cell. This
is achieved by extending the counter-based SpaceSaving algorithm described
in Section 6. As a result, only a fraction of the terms is stored, while we
maintain guarantees about the results of queries. The precomputed most
frequent term counts are maintained at every object insertion. The result is
a ranked summary for each cell at each granularity of the terms used in the
objects that fall into the cell, as shown in Figure 5.2.
5.3. PROPOSED SOLUTION

Temporal Support

To accommodate the temporal extent of queries, the spatial grid is extended with a temporal dimension. The finest temporal granularity is the smallest query-enabled time interval, e.g., an hour. We create a new instance of every spatial grid cell for every new such time interval. This is illustrated in Figure 5.3 where each cell has frequent term counts for each time interval. This arrangement enables efficient support for temporal query ranges at the finest granularity, but it does not offer support for long temporal query ranges, e.g., weeks or months.

To support queries with arbitrary temporal extent, we also create new grid cells for multiple time granularities. For example we may define month, week, day, and hour as our time granularities. For each spatial cell the corresponding objects are grouped for each time granule. The temporal granules each maintain a summary of frequent term counts. This is illustrated in Figure 5.4. With this arrangement, a two-week query result can be obtained by merging only two frequent term summaries.
Combining Multiple Granularities

For each granule, we store aggregate information that allows us to return the top-k result for the granule. Since a coarser granule aggregates the same terms as a number of finer granules at a finer granularity, the coarser granule can be computed by merging the corresponding finer granules [1]. However, the computed aggregate looses its accuracy. Figure 5.5 illustrates the problem. A coarse granule contains two finer granules, and we maintain information needed to support \( k = 3 \) for these two granules. Combining them yields an incorrect top-3 list for the coarse granule. The correct result contains “Sandy” with count 4.

![Figure 5.5: Example of Incorrect Merging](image)

To avoid such incorrect merging, we maintain separate frequent term summaries for every granularity. Thus, when an object is inserted, its terms are reflected in the aggregate information stored for each granule the object falls into. The same principle applies to the temporal dimension.

System Overview

Figure 5.6 shows an overview of the adaptive frequent item aggregator (AFIA) system. The index structure in the middle consists of the above-mentioned spatial (SI) and temporal (TI) granularities aggregated with frequent item counting summaries (FIC).

High-level pseudo-code for stream data processing in AFIA is given in Algorithm 11. Each message from the data stream is pre-processed by splitting the message into terms and removing stop words. The corresponding summary is found in Line 3. As seen from Figure 5.6, finding the summary entails lookups in the index structure. If a summary is able to hold the new message, it is inserted as shown in Line 12. However, in some cases, it may be necessary to expand a summary before insertion. This is described in detail in Section 6.
If the summary cannot be expanded, it becomes inactive, and a new empty summary is created. This is called a checkpoint, which is covered in Section 23.

High-level pseudo-code for query processing is given in Algorithm 12. First, the query region is mapped to the corresponding cells. Next, a lookup is performed for each cell that each produces a set of summaries as seen in Lines 3–4. Each summary is a partial result. The lookup utilizes the query parameters $q.R$ and $q.I$ to limit the number of partial results. Finally, the results are merged to compute the top-$k$ terms and $k_g$ (detailed in Section 6).

**Algorithm 11: AFIASStreamProcessing** (stream $S$)

1. $\text{foreach } i \in S$ do
2. \hspace{1em} $\text{preprocess}(i)$;
3. \hspace{2em} $\text{summary} \leftarrow \text{find}(i)$;
4. \hspace{3em} if ! summary.insert($i$) then
5. \hspace{4em} $\text{checkpoint}(\text{summary})$;

**Algorithm 12: $[R, k_g]$ AFIASQuery** (query $Q$)

1. $\text{cells} \leftarrow$ retrieve cells that $Q.R$ covers;
2. $\text{summaries} \leftarrow$ empty list of summaries;
3. $\text{foreach } c \in \text{cells} \text{ do}$
4. \hspace{1em} $\text{summaries}.add(\text{lookup}(c, Q))$;
5. $[R, k_g] \leftarrow \text{merge}(Q.k, \text{summaries})$;
6. return $[R, k_g]$;
Dynamic Summaries

Motivation

Existing frequent item counting techniques utilize static-sized summaries. While this works when the vocabulary is known beforehand, any static summary size falls short when the vocabulary changes considerably. And since we aim to cover the entire world at multiple spatial and temporal granularities, the vocabulary is guaranteed to vary across cells. If we use static-sized summaries with too few counters, we cannot use the summaries to determine the top-\(k\) most frequent terms. In Example 11, too few counters are used to capture the top-2 most frequent terms. An obvious solution is to use summaries with an extremely large number of counts. However, this is either not feasible or unattractive because it require excessive storage space. Instead, we develop a new technique that dynamically adapts the number of counters in a summary to accommodate the changes in the data.

Example 11. Assume we wish to support top-2 queries with the SpaceSaving algorithm and that a summary with 2 counters is used. The following terms are inserted in the given order: evacuation, evacuation, sandy, evacuation, sandy, storm.

The summary then becomes: \{evacuation = 3(\Delta = 0), storm = 3(\Delta = 2)\} (where \(\Delta\) captures the maximum overcounting error. In contrast, the correct result is \{evacuation, sandy\}. □

Aggressive Increment

The 2 counters in Example 11 failed to support top-2 queries for the data given. This type of problem may occur in every cell if the number of counters is fixed and too small. In the new approach, we increase counters when needed. Thus, the summaries can be initialized with a low number of counters, and more counters can be introduced in cells when needed. Cells with low activity and skewed term occurrence will use few counters, while cells with high activity and temporal variation in the terms will use many counters.

It is important to capture all terms since a miss may lead to an incorrect top-\(k\) result. Thus, we perform what we call Aggressive Increment. The procedure is covered in Algorithm 16.

Term counts are recorded in a summary until all available counters are utilized, as shown starting in Line 6. The counts are maintained for each processed term, as shown in Line 18. Assume we start with 2 counters, i.e., \(limC = 2\). Using the data in Example 11 we are out of counters when we reach the sixth term and need a new counter.

When all counters are used and a new counter is needed for a new term, we find the term with the minimum count as seen in Line 8. If the new term does not influence the targeted top-\(k\) result, \(q.k \leq \text{targeted-}k\), we proceed...
5.3. PROPOSED SOLUTION

Algorithm 13: bool insert(item i)

1. $T \leftarrow$ currently monitored items ;
2. $\text{limC} \leftarrow$ maximum #counters ;
3. $\text{eFree} \leftarrow$ all-counts-error-free flag ;
4. $\text{targeted-k} \leftarrow$ the targeted $k$ for top-$k$ queries ;
5. if $i \notin T$ then
   6. if $|T| < \text{limC}$ then $T \leftarrow T \cup \{i\}$ ; \hfill // $c_i \leftarrow 0$
   7. else
      8. $j \leftarrow \min_{j \in T} c_j$ ; \hfill // item with min count
      9. if $c_{\text{targeted-k}} < c_j + 1$ then
         10. if $\text{eFree}$ then return false ; \hfill // checkpoint
         11. $\text{limC} \leftarrow \text{limC} \times 2$ ; \hfill // aggressive increment
         12. $T \leftarrow T \cup \{i\}$ ; \hfill // $c_i \leftarrow 0$
      13. else \hfill // else top-$k$ is high enough
         14. $c_i \leftarrow c_j$ ; \hfill // previous min count
         15. $\Delta_i \leftarrow c_j$ ;
         16. $T \leftarrow T \cup \{i\} \setminus \{j\}$ ;
         17. $\text{eFree} \leftarrow \text{false}$ ; \hfill // error introduced
      18. $c_i \leftarrow c_i + 1$ ; \hfill // increment the counter
      19. if $c_{\text{targeted-k}} < c_i \& \Delta_i \neq 0$ then
         20. \hfill // targeted-$k$ items are not error-free anymore, so restore:
            21. rollback($i$) ;
            22. return false ; \hfill // checkpoint
   23. else return true ;

as in the algorithm SpaceSaving to take over the counter from the currently least frequent term and record the new term’s maximum possible count and maximum possible error (Lines [14][17]).

However, if the new term influences the top-$k$ result and the summary does not contain errors, the number of counters in the summary is doubled (Line [11]). This prevents the term from entering the top-$k$ result with an error. In Example [11] the summary would be doubled from 2 to 4 counters when term storm is to be inserted, thus allowing the term to be counted without any errors being introduced into the top-2 result. It is important to maintain error-free top-$k$ results for all cells since we merge cells and summaries when computing query results. With error-free results for the cells, the top-$k$ result of a query is more likely to contain many guaranteed top-$k$ terms: in terms of the query definition in Section 5.2 $k_g$ is close to $k$. We return to this in Section 6.

When a new term is to be inserted that may influence the top-$k$, possible errors may have been recorded for counters outside the top-$k$. As seen from
Line 10, we do not double the number of counters when this occurs. Doubling the number of counters when errors exist, the new terms will not maintain any guarantees about previously inserted terms, leaving the summary in an inconsistent state. Instead, we avoid this inconsistent state where new terms have wrong counts by performing a checkpoint, to be described in Section 23.

When the occurrence of a term that already exists in the summary is to be captured, the associated counter is incremented as shown in Line 18. If the term is outside the top-\(k\) counters, the term’s counter may record a possible error. When the counter is incremented, it may enter the top-\(k\) counters along with its possible error, as seen in Line 19. To prevent this, a rollback restores all counters to their state before the term was inserted, and a checkpoint is performed. Thereby, the targeted top-\(k\) counters remain error-free.

**Lemma 2.** The incrementing of counters preserves existing counts and their associated possible errors.

**Proof.** Before the number of counters is doubled, no errors exist in the summary according to the algorithm. Thus, no items have been lost and all counters are exact. Therefore, the introduction of new counters has no impact on already inserted items. □

**Lemma 3.** For all elements \(e_i\) where \(i \leq k\), \(\Delta_i = 0\).

**Proof.** The insertion algorithm allow possible errors to be associated with elements \(e_i, i > k\). Before an element with an associated possible error can become an element \(e_i, i \leq k\), a checkpoint is performed, yielding a new empty summary. Therefore, no element \(e_i, i \leq k\) can have an associated possible error. □

The ability of the insertion procedure to increment the number of counters in a summary allows us to start with summaries with few counters. We introduce a parameter \(\text{initCounters}\) to control the initial number of counters. The experimental evaluations study the impact of different settings of this parameter.

### Checkpointing

To prevent an inconsistent state and a top-\(k\) result with a possible error, we introduce a checkpointing mechanism. A checkpoint is performed before an inconsistent state is reached. The currently active summary is rendered inactive and is archived in its consistent state and is replaced by a new active summary with twice as many counters. Thus, earlier changes to the size of the archived summary due to Aggressive Increment, if any, are retained in the new summary, and since that number of counters did not suffice, it is doubled.

Due to the introduction of checkpointing, each cell can contain multiple archived summaries. However, at any point in time, only one summary is
5.3. PROPOSED SOLUTION

Figure 5.7: Cell with One Archived and One Active Summary.

active and is maintained as seen in Figure 5.7, where the left summary is achieved and the right is active.

Example 12. Assume the same setting as in Example 11 and that the terms are processed according to Algorithm 16. When reaching term storm, the number of counters is doubled, yielding this summary: \{evacuation = 3 (\Delta = 0), sandy = 2 (\Delta = 0), storm = 1 (\Delta = 0)\}. Then, inserting terms hurricane, flooding and water yield: \{evacuation = 3 (\Delta = 0), sandy = 2 (\Delta = 0), flooding = 2 (\Delta = 1), water = 2 (\Delta = 1)\}.

Inserting the term NYTMetro triggers a checkpoint (as seen from Line 10) before errors are introduced in the targeted top-k result. The resulting summaries are shown in Figure 5.7.

Compaction of Inactive Summaries

Summaries can become inactive for two reasons: (i) due to a checkpoint as just described and (ii) because their time interval ceases to overlap with the current time so that no updates will apply to them. As described in the following section, only the targeted top-k counters are considered during query processing. As part of the archiving of inactive summaries, we thus remove the counters \(c_j \) with \(j > \text{targeted}-k\). This releases memory that is no longer needed. In the setting of Example 12, the two last grey counters of the left summary in Figure 5.7 are removed from the summary before archiving.

Relaxed Decrement

The number of updates that apply to a cell may vary across time. The Aggressive Increment procedure effectively handles the case of increasing activity.
However, activity may also decrease, causing already allocated counters become unnecessary and resulting in poor memory utilization. There is thus a need to be able decrement the number of counters in a summary.

Care must be taken when decreasing the number of counters since an overly aggressive decrease may lead to more checkpoints. When performing checkpoint too frequently, query performance is reduced because more merging of summaries is needed.

The Relaxed Decrement procedure described in Algorithm 14 is designed to reduce the number of counters in a summary without causing excessive checkpoints. The procedure is invoked after any given time granule has passed. As shown in Lines 4–5, the number of counters is halved if the summary’s number of error-free counters is at least double targeted-$k$. Then the summary maintains many more error-free counters than required. Note that if any checkpoint was created during the previous time granule, this means that the number of counters has been increasing. Therefore, the activity level may be expected to remain at the same level or continue to increase. Consequently, no reduction is performed.

\[
\text{Algorithm 14: Summary RelaxedDecrement(int cp, Summary s)}
\]

1. $cp \leftarrow \#\text{checkpoints in the previous time granularity}$;
2. $s.limC \leftarrow \text{maximum } \#\text{counters}$;
3. $s.errorfree \leftarrow \#\text{counters without error}$;
4. \textbf{if} $cp == 0$ \& $s.targeted-k > s.errorfree \times 2$ \textbf{then}
5. \hspace{1em} $s.limC \leftarrow \frac{s.limC}{2}$;
6. \hspace{1em} \textbf{return} $s$;

\[\]

Query Processing

Overview

The spatio-temporal range of a query generally covers a number of cells. We cover a query region and interval with the fewest possible cells from the multi-granularity structure and use the summaries of these cells to produce the query result. Specifically, any query region can be covered by a unique set of most coarse cells. Starting from the coarsest granularity it is checked if each cell is contained in the query region. We refine a cell that merely intersects the query range by the cells it covers at the next, finer granularity until the cells considered are contained in the query region. If a query region partially overlaps other cells, we “snap” the boundary of the query region to the nearest inclusive border. Next, as shown in Figure 5.6, we proceed to find the summaries that overlap with the query time interval. This results in a number of summaries that are to be “merged” to produce the query result.
5.3. **PROPOSED SOLUTION**

The summaries of the cells needed to cover the query range are used to produce the result. This calls for a technique to merge such summaries into a query result (in Line 5 in Algorithm 12). Also, if a query region and interval are identical to those of an existing cell, the same merging procedure applies to possibly several archived summaries associated with that cell.

### Merging Dynamic Summaries

Recall that each summary maintains counters for only the most frequent terms seen in that summary. Thus, different summaries maintain counters for different sets of terms. Figure 5.5 illustrates how a wrong result can be produced when merging summaries. The problem occurs when terms that do not occur in all summaries make it into the merged summary.

However, we can still provide guarantees about the merged summaries. Recall that targeted-$k$ counters of each summary are without any errors. For each summary, we maintain targeted-$k + 1$ counters to provide guarantees about the merged summary. Thus, before running Algorithm 16, the value of targeted-$k$ is incremented with one more counter. Algorithm 15 details the merging procedure for dynamic summaries.

We can guarantee that a merged summary is correct if no other term can possibly make it into the summary. To do this, we utilize the extra counter maintained for each summary—see Line 4. These extra counters provide information about the best possible counts of terms outside the top targeted-$k$ terms. The best case for the remaining terms is calculated by adding up the values of these counters in each summary. If the resulting value is below the lowest count for a term in the merged summary, the terms are not competitive, and the summary is correct—see Line 21.

In the example in Figure 5.8a, targeted-$k$ is 3, the shading identifies the targeted-$k + 1$st counter maintained by a summary, and the last, grey rows are not stored in the summaries, but are only included for illustration. In this example, the merged summary is correct because no other term can have a count that places it among the top targeted-$k$ terms. A count that exceeds 3 is needed to enter the merged summary, and the best possible count of a term not in the merged summary is 3.

We cannot always return a correct merged summary. In the example in Figure 5.8b, the merged summary fails to include the term “Sandy” that has a higher count than “York.” The targeted-$k + 1$st counters maintained by the two cells tell that a term not in the merged summary, e.g., “Sandy,” may be able to gather a score of 2 from each cell. The counters guarantee that terms in the merged summary with scores no less than 2 + 2 = 4 are correct. See Line 27 in the algorithm. In the example, we return a result with 3 terms along with the information that the 2 first are guaranteed to be correct (while the last may be correct). The correctness of the Algorithm 15 is given in Theorem 3.

Algorithm 15: $[R, k_g] \text{Merge}(\text{int } k, \text{parResults } P)$

1. $M \leftarrow$ set used for merging ;
2. foreach $p \in P$ do
3.   $p.T \leftarrow$ terms in the result summary ;
4.   $p.lc \leftarrow$ the best count outside targeted-$k$ ;
5. foreach $t \in p.T$ do
6.   $t.count \leftarrow$ frequency of the term ;
7.   if $t \in M.\text{terms}$ then
8.     $M[t].\text{count} \leftarrow M[t].\text{count} + t.count$ ;
9.   else
10.     $sumlc \leftarrow 0$ ;
11.    foreach $p_s \in P$ do
12.        if $t \in p_s.T$ then
13.           continue ;
14.        $sumlc \leftarrow sumlc + p_s.lc$ ; // best case
15.     $t.count \leftarrow t.count + sumlc$ ;
16.     $t.\Delta \leftarrow sumlc$ ;
17.     $M[t] \leftarrow M[t] \cup \{t\}$ ;
18.     $low \leftarrow low + p.lc$ ;
19. $R \leftarrow k$ most frequent terms from $M$ ;
20. if $R.\text{minCount} \geq low$ & $\forall t \in R (t.\Delta = 0)$ then
21.     return $[R, k]$ ; // they are guaranteed
22. else
23.     $G \leftarrow$ set used for guaranteed terms ;
24.     $A \leftarrow$ set used for approximate terms ;
25.     foreach $t \in R$ do
26.          if $t.count \geq low$ & $t.count - t.\Delta \geq R.\text{next}.\text{count}$ then
27.              $G \leftarrow G \cup \{t\}$ ;
28.          else
29.              $A \leftarrow A \cup \{t\}$ ;
30.     return $[G \cup A, |G|]$ ;

Experimental evaluation studies the correctness of the merged summaries.

**Theorem 3.** Let a query $Q = (R, k, I)$, $k \leq$ targeted-$k$ be given that covers dynamic summaries $S_1$ and $S_2$ that describe datasets $D_1$ and $D_2$, respectively. Then there does not exist a term $t$ such that $\forall e \in \text{Merge}(k, \{S_1, S_2\})_g$ ($\text{score}(t, (D_1 \cup D_2) \cap \text{Merge}(k, \{S_1, S_2\})_g > e.\text{count}$), where $\text{Merge}(k, \{S_1, S_2\})_g$ contains only the guaranteed elements. Further, $\text{Merge}(k, \{S_1, S_2\})_g$ is or-
5.3. **PROPOSED SOLUTION**

(a) Guaranteed

<table>
<thead>
<tr>
<th>Term</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evacuation</td>
<td>5</td>
</tr>
<tr>
<td>New</td>
<td>4</td>
</tr>
<tr>
<td>York</td>
<td>3</td>
</tr>
</tbody>
</table>

(b) Approximate

<table>
<thead>
<tr>
<th>Term</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evacuation</td>
<td>5</td>
</tr>
<tr>
<td>New</td>
<td>4</td>
</tr>
<tr>
<td>Sandy</td>
<td>4</td>
</tr>
<tr>
<td>York</td>
<td>3</td>
</tr>
</tbody>
</table>

**Figure 5.8:** Guaranteed and Approximate Merging of Summaries

...dered descendingly on the count of the item.

**Proof.** It follows from Lemmas 2 and 3 that the counts in $S_1$ and $S_2$ are exact and have no errors for the first $\text{targeted}-k$ terms. Merging the summaries produces up to $2\text{targeted}-k$ terms.

There are two possibilities for the count and order guarantees of the merged summary:

First, the counts and order of the merged summary can be fully guaranteed. If the sum of the counts of the $\text{targeted}-k + 1$-st terms in $S_1$ and $S_2$ is lower than the count of each of the terms in the merged result, the merged summary contains the terms with the highest counts. Also, if the merged summary contains only $\text{targeted}-k$ terms, all the terms are known to be in both $S_1$ and $S_2$. Thus, no better combination of terms can exist.

Second, the order of the first $k_g$, $k_g \leq \text{targeted}-k$, terms can be fully guaranteed. If the $k_g$-th term has a higher count than the sum of the counts of the $\text{targeted}-k + 1$-st terms in $S_1$ and $S_2$, the term’s count exceeds that of any terms not in the merged summary. Thus, the terms in the merged summary are the ones with highest counts. The count of any term, $t_i$, $i \leq k_g$ that only occurs in one of the summaries, $S_1$ or $S_2$, is increased by the count of the $\text{targeted}-k + 1$-st item of the other summary. If the ordering of the first $k_g$ terms remains after the increments, no terms can exist in $D_1$ or $D_2$ that can change the order of the first $k_g$ terms.  

□
5.4 Experimental Evaluation

We proceed to evaluate the AFIA framework and also compare with other available techniques.

Experimental Setup

We use two different machines for the experimental study. For small-scale experiments (see Section 5.4), we use a local machine (termed local): a quad-core Intel Core i7-3770 with 8 hardware threads and 16 GB of main memory. For large-scale experiments (Sections 5.4–5.4), we employ an Amazon EC2 High-Memory Cluster (termed HMC) backed by a dual Intel Xeon E5-2670 processor with 32 hardware threads and 244 GB of main-memory. In all experiments, the processing occurs in-memory.

The framework is implemented in Java and we utilize the available parallelism on the machines as follows. One hardware thread is dedicated to a concurrent garbage collector and one to accepting incoming messages. The remaining hardware threads are assigned to equal-sized partitions of the world, as constructed for this purpose.

For the spatial partitioning, we use the Military Grid Reference System (MGRS) [39]. Thus way, we represent any location on the surface of the Earth at multiple granularities, ranging from $1 \times 1$ m$^2$ to $100 \times 100$ km$^2$, by an alphanumeric value. For temporal partitioning, we use the typical temporal hierarchies from data warehousing (hour, day, week, month). We drop cells with very low activity, specifically cells where the most popular term occurs in fewer than 5 messages at the finest granularity (hour). We will shortly describe how we fix the spatio-temporal hierarchy in AFIA based on the considered workload.

We perform three sets of experiments. The first measures the space and accuracy of the proposed algorithms and compares them with two baselines. The second evaluates the throughput, memory consumption, and number of checkpoints when varying the targeted-$k$ parameter. The third evaluates the query processing performance when varying the size of the spatio-temporal query range. We also show the number of checkpoints created during a time period.

Data and Queries

We collected all geo-tagged messages from the public Twitter FireHose during May, 2013. The dataset contains 110,426,053 tweets, yielding an average rate of circa 41 tweet per second. Figure 5.9 provides insight into the workload and helps choose an appropriate spatio-temporal hierarchy.

Figure 5.9a shows the total number of cells that receive at least one tweet. Thus, the graph shows the expected number of active summaries at the given
5.4. **EXPERIMENTAL EVALUATION**

![Graph](image)

(a) Total number of active cells.

![Graph](image)

(b) Terms per hour per 0.1×0.1 km² cell.

![Graph](image)

(c) Terms per week per 100×100 km² cell.

**Figure 5.9**: Twitter Workload Intensity at Different Spatio-Temporal Granularities
CHAPTER 5.

granularities. For example, every hour, the number of $0.1 \times 0.1 \text{ km}^2$, $1 \times 1 \text{ km}^2$, $10 \times 10 \text{ km}^2$, and $100 \times 100 \text{ km}^2$ spatial grid cells that receive at least one tweet is circa $0.5\text{M}$, $200\text{K}$, $40\text{K}$, and $10\text{K}$, respectively (i.e., the “Hour” curve in the figure).

Figure 5.9b considers the cells at the $0.1 \text{ km} \times 0.1 \text{ km} \times 1 \text{ hour}$ granularity. The y-axis depicts the number of such cells subject to different loads (the number of terms $t$ processed). This shows that the number of “interesting” cells (with at least 100 terms) is quite low, barely reaching 100. Therefore, we fix the finest temporal granularity at an hour.

Figure 5.9c considers the cells at the $100 \text{ km} \times 100 \text{ km} \times 1 \text{ week}$ granularity. The figure shows that thousands of such cells have summaries that receive more than 1 thousand terms.

We fix $k$ at 25 and generate 2 query sets each consisting of 100 queries. Each query in the first set has a temporal extent with a length chosen at random among 1 hour, 1 day, and 1 week, and it has a randomly selected spatial region. The spatial granularity of the coarsest cell that is contained in the spatial query region is the query’s spatial granularity, and the query region is enlarged to cover all the cells at that granularity that it intersects with. In visual terms, this procedure corresponds to “snapping” the spatial query region to the coarser granularity. The underlying rationale for adopting this procedure is that finer-granularity cells do not contribute with term counts at the same magnitude as do cells at the granularity of the coarsest fully covered cell. Since any rectangle can be created from a number of merged squares, we limit the experiments to squares. Each cell contains $10 \times 10$ cells at the next, finer granularity. Thus, the largest number of cells any query square can cover is $18 \times 18 = 324$. As a result, we employ queries that may cover up to $1,800 \times 1,800 \text{ km}^2$.

While the first query set focuses on variation in the spatial range, the second focuses on temporal variation. Each of the 100 queries has a spatial region chosen at random from one of the spatial granularities. The query resulting in most temporal summary merges is the query that spans from the first hour after midnight on the first day in May until the hour before midnight on the last day of May. This query covers $2 \times 23$ hours, $3 + 5$ days, and 3 weeks, which amounts to 57 temporal summaries.

To avoid empty results, we do not consider regions that only cover empty cells. Note, that we do not perform compaction on the last (active) temporal granularities, to better reflect the real-life scenario where a stream is constantly keeping these summaries active. To simplify the experiments, the initCounters parameter, which determines the initial number of counters is set to the same value as targeted-$k$.

Baselines

We consider two baselines for comparison and validation of our proposal.
SS: In each active spatio-temporal summary, we aggregate the most frequent items using the SpaceSaving algorithm. Active summaries maintain a fixed number \((m)\) of items and their counts, while archived summaries are compacted by keeping only the counters for the \(k\) most frequent terms. Choosing a value for \(m\) represents a trade-off between storage size and result accuracy.

HT: In each active spatio-temporal summary, we maintain a full list of terms using a hash table. Once the summary becomes inactive and is archived (its time granularity passes), the list is sorted and cleaned up by discarding all but the counters for the targeted top-\(k\) terms. This approach maintains the exact top-\(k\) items at all spatio-temporal granularities and serves to measure the accuracy of the other approaches. However, since HT has excessive storage requirements, we do not expect it to scale under the considered workloads.

The Space Versus Accuracy Trade-Off

In the first set of experiments, we evaluate the accuracy of our approach by comparing it with the baselines. Since HT maintains the exact counts at all granularities, its storage requirements are too high for large scale experiments. Similarly, SS maintains a fixed number \(m\) of counters at all granularities, and \(m\) cannot be set too high for large scale experiments. Thus, we consider only tweets occurring in the spatial region with the 11S grid zone designator (a region in the US of size ca. \(900 \times 550 \text{ km}^2\)). We use the local machine.

We consider variants of SS where \(m\) is set to 50, 100, and 200. In AFIA, we set targeted-\(k\) to 50. The first week (four days in May) is used for warm-up, and only the accuracies of the subsequent summaries are evaluated. We perform top-\(k\) queries on all summaries at each spatio-temporal granularity. We compare the top-\(k\) most frequent terms in each summary produced by AFIA and SS and compare with the corresponding top-\(k\) items produced by HT. We set \(k = 20\).

The results are shown in Figure 5.10. We measure the fraction of correct top-\(k\) for varying spatial granularities, and we report separate graph for each temporal granularity. The coarser spatio-temporal summaries are subject to higher loads and more diverse terms. At the finest granularity \((0.1 \times 0.1 \text{ km}^2\) in Figure 5.10a), all approaches produce quite accurate results. However, as the spatial or temporal granularity gets coarser, the accuracy of SS deteriorates. Increasing \(m\) does not solve the problem, but just postpones it. On the other hand, AFIA is virtually as accurate as HT at all spatial and temporal granularities. Its Aggressive Increment procedure is effective in making it adapt its number of counters as needed to capture the most frequent items very accurately. Due to checkpointing (when several summaries have to be merged), few top-\(k\) terms are missed, and we did not observe accuracies below 0.99, 0.98, and 0.97 for hours, days, and weeks, respectively.

Figure 6.9 reports the average number of counters maintained per summary in the same experiment. Note the log-scaled y-axis. The figure confirms the
impracticality of using HT in large scale scenarios. SS maintains a fixed number of counters. The figure shows how AFIA adapts. At hours and days (Figures 5.11a–5.11b), AFIA follows SS ($m = 100$), implying that on average 100 counters is enough to achieve the desired accuracy. At weeks (Figures 5.11c), the adaptivity of AFIA is even clearer: for finer cells, it uses much fewer counters than for coarser cells.

This experiment shows how AFIA achieves the best of two worlds: accuracy is at the level of HT, which is 100% accurate, while the space requirements are at the level of SS with a small number of counters. Having demonstrated the accuracy of our approach, we proceed to large scale performance experiments on the HMC machine.

Varying targeted-$k$

In this experiment we vary targeted-$k$ and consider two cases where the finest spatial granularity is $0.1 \times 0.1 \text{ km}^2$ and $1 \times 1 \text{ km}^2$. For each case, we also maintain all the coarser granularities. Figure 5.12 shows the results. AFIA always has a throughput of more than 13,000 items per second (Figure 5.12a). As expected, the throughput increases as either targeted-$k$ decreases or the finest granularity becomes more coarse.

Figure 5.12b shows how the memory consumption increases with increasing targeted-$k$. This is because increasing numbers of counters have to be maintained to guarantee correctness when there are more terms. The memory consumption increases slowly at first, due to fewer checkpoints occurring (Figure 5.12c). Thus, for small values of targeted-$k$, much of the space consumption is due to checkpoints. As expected, more memory is required for maintaining finer granularities. The results show that a single HMC machine can maintain between 2 and 3 months of data, meaning that it takes 4 to 6 machines to support a full year. At current cost, this is feasible for larger companies. Note that the measured memory consumption represents the entire Java heap memory used by the application. The actual aggregates (not shown) require much less.

When targeted-$k$ is set to a low value, the number of required counters will most often also be low. With only a few counters, any small change in activity of a cell may impact the targeted-$k$ counters, resulting in a checkpoint. Therefore, we see that the number of checkpoints decreases with targeted-$k$. Having a high number of counters makes a cell more robust to changes in activity, and any small changes can be captured without checkpoints. The number of checkpoints increases as expected when a finer granularity is maintained.

---

1The price for a Spot Instance in June 2013 was USD 0.343 per hour.
Figure 5.10: Accuracy at Different Spatio-Temporal Granularities
Figure 5.11: Average Number of Counters Maintained at Different Spatio-Temporal Granularities
Figure 5.12: Varying targeted-\(k\)
Varying Q.R and Q.T

We fix targeted-k at 100 and vary the spatial and temporal extents of the queries. We employ the two query sets and measure the average runtime. Figures 5.13a and 5.13b show the results.

Retrieving a summary requires one or more constant time lookups (in Algorithm 12, Line 3), depending on the spatial or temporal granularity of the query, and it may also require the merging of checkpointed summaries. Because finer granularity cells generally receive fewer items than coarser granularity cells, the runtime is lower at finer granularities. Fewer merges are required because more cells are empty. The results show that this applies to both the spatial and the temporal granularities.

When the spatial extent increases, more lookups are performed, resulting in an expected higher runtime. With more coarse temporal granularities, more merges are performed, increasing the runtime further. The runtime is not made worse to the same degree when varying the temporal extent since the temporal lookups are performed after the summaries are filtered spatially. Thus, significantly fewer entries exist, resulting in faster lookup times. However, as the spatial extent increases, the runtime also increases since more cells are retrieved and merged.

Checkpoints Over Time

In the last experiment, we explore the checkpointing activity over time. We fix targeted-k at 100 and count the number of checkpoints per hour for the first 5 days. Figure 5.14 shows the results. The first day sees more than 18,000 checkpoints, but the number of checkpoints drops quickly. This is because a new day is initialized based on the ending state for the previous day. On the fifth day, only half as many checkpoints are performed. Overall, the number of checkpoints keeps decreasing, but may increase occasionally. This demonstrates the adaptivity of our approach, as the number of counters in the cells adjust to the activity levels of the cells.

5.5 Conclusions

We present a new framework for the processing of spatio-temporally constrained top-k most popular terms queries on streaming, spatio-temporally tagged text content, such as microblog posts. The framework’s index extends existing techniques for counting frequent items in summaries to allow the summaries to grow and shrink dynamically to adapt to changes in the incoming data. Query processing works by merging relevant summaries. To provide guarantees about the query results, a new merging algorithm is proposed that supports spatio-temporal query regions. Experimental studies with an implementation of the framework offers insight into the accuracy, scalability and
5.5. CONCLUSIONS

Figure 5.13: Runtime, Varying Spatial and Temporal Query Extents

Figure 5.14: Checkpoints Over Time
performance of the processing of incoming data and queries; and they demonstrate that the framework is capable of offering very accurate query results at high performance at scale.

Acknowledgments

This research was supported in part by the MADALGO research center and in part by the Geocrowd Initial Training Network, funded by the European Commission as an FP7 Peoples Marie Curie Action under grant agreement number 264994.
Chapter 6

Scalable Spatio-Textual Region Querying in Key-Value Stores

Abstract

With the proliferation of Internet-connected, location-aware mobile devices, such as smartphones, we are also witnessing a proliferation and increased use of map-based services that serve information about relevant Points of Interest (PoIs) to their users. We provide an efficient and practical foundation for the processing of queries that take a keyword as argument and return the $k$ most relevant PoIs that belong to the part of the map covered by the user’s screen. The paper proposes a novel techniques that encodes the spatio-textual part of a PoI as a compact bit string. This technique extends an existing spatial encoding to also encode the textual aspect of a PoI in compressed form. The resulting bit strings may then be indexed using indexing approaches such as B-trees or hashing, that are standard in any DBMS or key-value store. The paper also proposes a novel top-$k$ query algorithm that merges partial results while providing an exact results. An extensive empirical study with real-world data shows that the proposed techniques provide excellent indexing and query execution performance on a standard DBMS.

6.1 Introduction

The proliferation of internet-enabled, geo-positioned mobile devices has created an increased demand for local information. Thus, we are also witnessing an increased proliferation and use of different map-based services that serve local information. Local information is typically organized around Points of Interest (PoIs) that may be of different types, e.g., hotels, restaurants, and parks, that include exact locations, names, and textual descriptions. As of
May 2014, Google Places covers more than 57 million hotels, 41 million restaurants, and 51 million parks in addition to many other types of PoIs. In total, Google Places contains many more than 100 million PoIs.

Map-based services allow users to retrieve such different types of PoIs near them. For example, users may look for hotels or restaurants in neighborhoods where they are currently present or are going to be at a later time. Typical map-based functionality supports the retrieval of relevant PoIs that belong to the part of space visible on the user’s screen. This functionality supports users who wish to browse or explore their surroundings and is not limited to the retrieval of PoIs that are closest to an exact user location.

Several proposals exist that are capable of finding the most relevant places that are closest to an exact location. They are either R-tree based [16, 23, 27, 41, 58, 76, 90, 91, 98], grid based [51, 87], or space filling curve based [20, 22]. The proposals either support Boolean top-\(k\) or queries, or they support top-\(k\) queries with a ranking function. However, they all suffer from two limitations: (i) they involve the use of a special index structure, making it difficult to benefit from existing DBMS technologies, and (ii) they focus on finding the top-\(k\) most relevant objects, that are closest to an exact location instead of providing the top-\(k\) most relevant objects in a region.

![Figure 6.1: Top-5 results for a region query (left) and a nearest neighbor query (right)](image)

We propose a new type of query that returns the top-\(k\) most relevant PoIs in a spatial region. An example is given in Figure 6.1 where the top-5 most relevant objects are given for both a region and a nearest neighbor query. The left-most example supports exploratory behavior of users, since the top-5 most relevant objects may be located anywhere within the region. In this example, the three best places are near the beach. This approach is used by all of the major providers of map-based services, including Google Maps, Bing Maps, and Yahoo! Maps. The right-most example, with the top-5 most relevant nearest neighbors, retrieves objects close to the query location and is
6.2. RELATED WORK

The paper makes four contributions. First, we propose and give a precise definition of a new query that finds the top-$k$ most relevant objects in a spatial region with respect to their textual descriptions. The query is similar to the approach used by major providers of map-based services. A baseline algorithm is given that modifies state-of-the-art algorithms for nearest neighbor queries to support the proposed region query.

Second, we propose a novel approach to the storing and indexing of spatio-textual objects in any DBMS or key-value store. The objects are first encoded into a bit string with a spatial and a textual part. The spatial part is encoded using a grid index, while the textual part employs a hash function. Collisions among the bit strings that encode the text may occur, and a parameter is introduced that controls the balance between the query execution time and the space consumption. The bit strings may be maintained by any existing indexing technique.

Third, we present an query processing algorithm that provides the exact result to the proposed query. The algorithm queries the spatio-textual objects using the indexed bit strings. The query region may overlap several grid cells, which are merged while guaranteeing the exact result.

Fourth, we give an extensive experimental study of the proposed techniques using real-world data. More than 100 million objects are encoded, indexed, and queried. The study suggests that the proposed solution is scalable and capable of efficiently supporting the proposed query.

The remainder of the paper is structured as follows. Section 6.2 covers related work. The problem definition is given in Section 6.3. Section 6.4 presents the proposed solution, encompassing the spatio-textual bit string encoding and indexing strategy followed by an exact query processing algorithm. The experimental evaluation is given in Section 6.5. Finally, we conclude in Section 6.6.

6.2 Related Work

Recently, substantial research efforts on geo-textual indices have been reported. The proposed solutions often use a combination of the R-tree [38] or a variant [5] for the spatial indexing and inverted files for indexing the text [23, 41, 58, 76, 90, 98]. Others use bitmap files for text indexing [16, 27, 91]. Solutions using grids [51, 87] and space-filling-curves [20, 22] combined with inverted files have also been proposed. The spatio-textual parts are combined in three different ways: (i) either closely combined, with no clear separation [23, 23, 27, 41, 51, 58, 90, 91], (ii) with the spatial index followed by the text index [16, 20, 87, 98], (iii) or with the text index followed by the spatial index [76, 87, 98]. A comprehensive experimental evaluation of geo-textual indices is also available [19].
The majority of the related work targets Boolean queries and does not return ranked results. This paper focuses on providing a ranked result to provide a size-limited result. With the vast amounts of data that may exist in a region, it is often not helpful to return all objects in a region that contain a given term.

Some of the related work is able to efficiently return the top-\(k\) most relevant neighbors \[23, 58, 76, 90\]. These works all use the R-tree and the query processing algorithms may be modified to find the top-\(k\) most relevant objects in a region. A baseline algorithm that uses a geo-textual index is given in Section 6.3. However, this related work uses custom index structures, while this paper proposes a technique that may be employed with existing standard indexing techniques. Despite the substantial research on answering spatio-textual queries, to the best of our knowledge, no existing work addresses encoding of spatio-textual objects in combination with existing indexing techniques.

Multi-dimensional spaces can be mapped to a one-dimensional space in order to support the indexing of multi-dimensional points in a standard DBMS. The mapping of multi-dimensional space into a one-dimensional space is often done using a space-filling curve such as a Z-curve or a Hilbert-curve\[34\]. The existing techniques consider only the spatial dimension, whereas we propose a new problem that considers both the spatial and the textual properties of the objects.

The Google S2 Geometry Library \[35\] provides an open-source implementation that maps two-dimensional objects on the Earth’s surface to a one-dimensional representation. It encloses the Earth sphere in a cube and imposes a quad-tree \[33\] type partitioning to each of the 6 faces. The cells are encoded and decoded by means of a Hilbert curve \[43\]. This provides a bit string representation of a location; this paper leverages and extends this work.

Hash functions can map text strings of arbitrary lengths to fixed-length hash values \[52\]. When two different strings produce the same hash value, a collision occurs. As the length of the hash value decreases, the number of collisions increases. A perfect hash function maps the input to unique hash values and is thus collision-free. Hash functions are used in the proposed solution to limit the number of textual descriptions and their lengths.

6.3 Preliminaries

We proceed with giving a formal definition of the problem addressed in the paper followed by a baseline algorithm that modifies state-of-the-art algorithms.

Problem Definition

Let \(D\) be a set of spatio-textual objects \(o = (\lambda, \text{doc})\), where \(\lambda\) is a point location, and \(\text{doc}\) is a text document. Let \(D_R\) be the set of objects in \(D\) that are
inside the spatial region $R$.

A top-$k$ spatio-textual region (TkSTR) query $TkSTR = (R, keyword, k)$ returns an ordered list of $k$ objects from $D_R$ that are the most relevant to $keyword$ according to a textual relevance function computed by language models similar to the one presented in related work [23]. For ease of understanding, we use the term frequency function $tf(t, o.doc)$, that returns the number of occurrences of term $t$ in the text document $o.doc$. However, other textual relevance functions may be applied.

Definition 6. With the above definitions, we can define the result of the TkSTR query with arguments $(R, keyword, k)$ as follows. The query returns $k$ objects from $D_R, o_1, \ldots, o_k$, where $tf(keyword, o_i.doc) \geq tf(keyword, o_{i+1}.doc)$, $i = 1, \ldots, k - 1$. Further, there does not exist an object $o \in D_R$ such that $tf(keyword, o.doc) > tf(keyword, o_i.doc)$, $i = 1, \ldots, k$.

![Fig. 2 & Table 1: A Spatial Region and Five Objects with their Term Frequencies](image)

Example 13. Figure 2 describes five objects $o_1, \ldots, o_5$, and Table 1 shows the document-term matrix of their documents.

Given the query $Q$ with $Q.R$ be the spatial region $R$ in Figure 2, $Q.keyword = \text{pizza}$, and $Q.k = 3$ the ordered result is $(o_4, o_2, o_5)$. No object inside $R$ has a higher frequency than the objects in the ordered result set. The object $o_3$ has the highest term frequency but lies outside $Q.R$.

For the same arguments but with $Q.keyword = \text{sushi}$ the ordered result is $(o_1, o_5, o_2)$.

Baseline Algorithm

No baseline algorithm exists for the TkSTR query. Thus, we instead modify existing techniques to process the TkSTR query before we present the proposed solution. Recent related work proposes geo-textual index structures that aim to enable efficient computation of the top-$k$ most relevant nearest
neighbors [23, 58, 76, 90]. They all employ the R-tree, which may also be used to return the result for a region query.

Each of the related works has a parameter $\alpha$ to balance spatial proximity and textual relevancy. The result of the $T_k$STR query does not depend on the spatial proximity of the objects. Therefore, $\alpha$ may be set to 0 in some studies [23, 76, 90] and 1 in another study [58] in order to eliminate the spatial proximity constraint. The query processing algorithms from related work take a query location as argument, which we may set to be any location inside the query region. Then the search is expanded from this location by iterating through the nearest neighbors. Since only objects within the query region should be considered, the query processing algorithms may return when the next nearest neighbor is further away than the longest distance from the query location to the border of the query region. Then, all objects inside the query region have been considered and no additional relevant objects exists.

The query processing algorithms in related work have to consider each nearest neighbor until the $Q.k$ most relevant objects have been found. In the worst case, all objects inside the query region have to be examined since the most relevant object may be located anywhere in the query region. However, the nodes of the R-trees in the existing proposals are augmented with information about the best term frequencies in their subtrees. This information may be used to prune some of the R-tree nodes. However, it is not known which and the amount of the objects that contain the specific term frequencies, which results in expensive full node scans.

The difficulty in providing an efficient baseline combined with the requirement of a custom index structure suggests that it is relevant to consider the invention of a new solution.

6.4 Proposed Solution

We present a framework that encodes spatio-textual objects into compact bit strings that may be stored in standard DBMSs and indexed using existing index structures. It includes an exact query processing algorithm capable of computing $T_k$STR queries using the bit strings and by combining partial results.

Spatio-Textual Bit String Encoding

We proceed to describe the encoding of the spatio-textual objects and then provide an overview of the framework.

Modern DBMSs have a number of indices that are optimized for processing queries efficiently. Standard indices include B-trees [7] and hash tables [52], and some DBMSs also support R-trees [38]. However, many queries may not be directly applicable to these standard index structures, e.g., the $T_k$STR query or the top-$k$ most relevant nearest neighbor query.
6.4. PROPOSED SOLUTION

We aim to develop an approach that can process the TkSTR query by utilizing the standard indices that are implemented in virtually any DBMS or key-value store: B-trees or the hash tables. To achieve this, we first encode each spatio-textual objects into a bit string so that they may be indexed by standard indices. The bit string has two parts: (i) a spatial part that is used to determine whether the corresponding object is within the query region, and (ii) a textual part that helps determine whether the text document of the object indeed contains the query term.

**Spatial Encoding**

We have the following requirements to the spatial part of the bit string:

1. Compact representation.
2. Fast lookup of regions of arbitrary size and location.
3. Sufficiently high resolution.

It is desirable to have a compact bit string to reduce the storage requirements and access cost. We aim at efficiently processing queries independent of the size and location of the query region. Also, we aim at providing a result efficiently for even very small query regions. Finally, the performance should not vary greatly depending on the size or location of the query region.

The Google S2 Geometry Library\[35\] utilizes a quad-tree style partitioning which yields cells of uniform area at multiple granularities. This satisfies the requirements of supporting arbitrary region sizes and locations. A bit string is created by enclosing the Earth in a cube and imposing a quad-tree partitioning on each of the 6 faces as illustrated in the example in Figure 6.3. The quad-tree partitioning has multiple levels, and the cells are enumerated using a Hilbert curve\[43\] that makes it possible to efficiently find the corresponding cell at each level.

The faces of the cube may be represented with 3 bits while the 4 cells at each level in the partitioning is encoded using 2 bits. With a total surface area of the Earth of 510,072,000 km\(^2\), each of the faces covers an area of 85,012,000 km\(^2\). To provide an accuracy of less than 1 cm\(^2\), 30 levels are required, which can be captured by using 63-bits. We may not need this level of accuracy and may reduce the number of levels, thus making the bit string more compact while still providing sufficient resolution. The experimental evaluation studies the impact of different numbers of levels. This fulfils our requirement of achieving a compact bit string.

**Example 14.** Consider the object \(o_1\) in Figure 6.3. It is on the first face of the cube which is represented by 3 bits: 000. By following the Hilbert curve...
in the grid tree, the object $o_1$ is in the cell with position 1 at the first level, represented by the bits 01. At the second level, it is position 4, represented by 0100. The complete encoded level 2 bit string of $o_1$ is: 0000100.

With this approach we are able to encode the spatial locations of objects as compact bit strings. The resulting bit string can be maintained by standard indexing techniques. We proceed with extending this approach with a textual bit string representation.

**Textual Encoding**

Large amounts of PoIs that cover all countries in the world are maintained by the map providers. Thus, the textual descriptions may be in any language, resulting in a large vocabulary. We have two requirements to the textual part of the bit string:

1. Compact representation.
2. Limited number of bit strings.

Having a compact bit string representation instead of string with characters in any language may reduce the storage requirements. Since standard indexing techniques like B-trees and hash tables are affected by the number of indexed objects, it is important to have a known and limited number of bit strings. Therefore, we aim at providing a limited number of compact bit strings that may be used for direct lookup in a standard index.

The TkSTR query takes the argument *keyword*, which is a term, e.g., "pizza” or "sushi." By encoding each term in the object document, $o.doc$, we may use the bit strings to produce a result for the TkSTR queries. To encode a term into a bit string of a fixed length, we propose to use the hash value of each term in the textual description. By using a one-way hash function, we achieve both of the above requirements. First, the length of the bit string
may be set to be sufficiently short by the choice of the hash function. Second, a hash function produces a fixed number of hash values. Next, it is possible to use any hash function and to use truncation of the hash values to obtain a limited number of compact bit strings. In the experimental evaluations, we study the use of different lengths of the hash values.

Hash functions may introduce collisions, which occur when two or more input text strings produce the same output hash value. An example is shown in Figure 6.4 where both the terms 'pizza' and 'sushi' are hashed into the same value '00'.

Figure 6.4: Two Input Strings Produce the Same Output Hash Value

When the hash function produces a small number of output hash values or when the hash values are truncated, it is more likely to see more collisions. However, by increasing the amount and length of the output hash values, the storage requirements and the time to perform indexing and lookups also increase.

By encoding the terms as compact bit strings, we may concatenate these with the spatial bit strings to achieve a single, compact bit string. The resulting compact bit strings hold information about the locations and textual descriptions of objects.

Example 15. Consider the object \( o_1 \) in Figure 6.3 and the term frequencies from Table 1. With the hash function from Figure 6.4, the combined bit string with level 1 spatial resolution for the terms 'pizza' and 'sushi' is: 0000100, while 'shoe' encodes to 0000110.

Handling Collisions

The textual documents of objects that are located in the same spatial grid cell may contain the same terms, resulting in identical bit strings (we call this a bit string collision). Also, objects located in the same cell but with different terms may be encoded to the same bit string since we use one-way hash functions that produce collisions (we call this a term collision). To be able to provide an exact result for the queries we need to be able to distinguish between co-located objects that share the same terms. We proceed to present
techniques to handle objects with the two types of collisions; the bit string collisions, and the term collisions.

**Bit String Collision** Multiple objects may exist in the same grid cell resulting in the same bit string. This is more likely in the first levels of the grid since they cover a larger area. With the large amounts of data any grid cell may contain numerous objects. These objects are also contained in grid cells at the lower levels. We want to avoid the duplicate storage of these large amounts of objects.

According to Definition 6 the $T_k STR$ query returns $Q.k$ objects with the highest term frequency. Therefore, in order to provide an efficient and exact result for any given grid cell it is only necessary to store the $Q.k$ objects with highest term frequency. The remainder of the objects will not be used to answer queries with a region that exactly matches the size of a grid cell. We propose to set a maximum value, $targetedK \geq Q.k$, for the number of objects to store in each cell. Thus, we can reduce the storage requirements and process all queries with $Q.k \leq targetedK$ without using any cells from the lower levels. Naturally, the value of $targetedK$ should be set sufficiently high in order to provide a useful number of objects in the result set.

The query region may not have the exact same size of the grid cell and thereby contain the $Q.k$ objects. The query region may be smaller than the size of the grid cells and even smaller than the size of the grid cells at the lowest level. To support these queries, we propose to store all objects at the lowest level. Thus, the lower level grid cell will not only store $targetedK$ objects, but will store all objects that are inside the given grid cell. Since we store all objects at the lowest level no information is lost, and an exact result may be provided for any query region.

**Term Collision** With the hashing technique we get a limited number of compact bit strings that may reduce storage requirements and the amount of identifiers to index. However, this may introduce collisions as seen from the example in Figure 6.4. We propose a method that handles the term collisions such that it may be employed using any key-value store.

The grid cells may describe up to $targetedK$ objects for any given term. For the cells at the lowest level it may be more than $targetedK$. The mapping of a term to the actual objects can easily be done when there are no term
collisions. The hash value simply serves as a unique identifier for the specific term and may point directly to a bucket with the objects. Consider Figure 6.5 where the same input and output is used as in Figure 6.4 and targetedK is set to 2. The input term "shoe" encodes to "10" and points directly to objects $o_5$ and $o_2$ from Figure 6.3 ordered by the term frequencies from Table 1.

However, when term collisions occur we propose to store a separate chain of terms in a bucket since the hash value no longer uniquely identifies the terms. Each of these stored terms identifies the corresponding objects. In Figure 6.5 both of the terms "pizza" and "sushi" encodes to "00." Therefore, the bucket contains both terms along with the corresponding objects. With this approach we can distinguishes between objects that are located in the same grid cell with the same terms.

Framework Overview

We proceed to give an overview of the framework. First, the data is preprocessed such that each term of the object documents is handled with the proposed spatial and textual encoding techniques as seen from Figure 6.6. The result is a bit string for each term of the object documents. The bit string is then stored and indexed in a standard DBMS using standard indexing techniques. A detailed description of the storing of objects is provided in the following section.

The query processing is shown in the lower part of Figure 6.6 and is described in detail in Section 17. First, the query region and keyword is preprocessed such that the spatial and textual parts of query are encoded into a bit string. This bit string is then used to perform a lookup in a standard DBMS that returns a number of objects. Lower levels of the grid structure may have to be searched in order to produce an exact result. Therefore, the bit string is refined and used to perform additional lookups. When enough objects necessary to answer the query have been retrieved, the result may be returned.

Figure 6.6: Framework Overview with the Spatial and Textual Encoding (two left-most figures) and a Standard DBMS (right-most figure)
Storing Objects in a Key-Value Store

Existing DBMS and cloud database technologies all support key-value stores with standard indexing techniques such as B-tree and hash Tables. We propose a method that works with all key-value store indexing techniques. The proposed method encodes objects to a bit string that may be represented by data types such as integers or strings, depending on the types available in the DBMS. When storing the objects, each term of the object documents is encoded into a bit string. To apply the proposed techniques to a key-value store, we propose to store the bit strings as the key. By having an index on the key we are able to perform lookups of the bit strings efficiently. Since the query region may be of arbitrary size we store a bit string for each level of the grid as seen from the key column in Table 6.1 that describes all objects from Figure 6.3. If we only stored objects at the finest levels, a large number of cell would have to be fetched to provide a result for queries with large regions.

Without term collisions, any key will at most map to targeted_k object references. With a standard page size of either 4 or 8 KB it can hold 32 bit integer references to 1,000 or 2,000 objects, respectively. To avoid fetching each object when performing query processing, we propose to store the term frequency in a 16 bit integer along with a 32 bit integer for both the latitude and longitude for each object. Thus, the object representation will have the following structure:

\[
\text{id(}\text{INT}_32\text{)}\text{tf(}\text{INT}_16\text{)}\text{lat(}\text{INT}_32\text{)}\text{lng(}\text{INT}_32\text{)}.
\]

In total 14 bytes are required to store this object reference. A page size of 4 or 8 KB can hold approximately 300 or 600 objects, respectively. We assume that Q_k will be much less than this number and thus an equally low value of targetedK. With these storage requirements we propose to store all object references directly as the value as seen from Table 6.1. The table describes the objects from Figure 6.3 ordered by the term frequencies from Table 1. To simplify we only give the identifier of the object and not the full object representation.

When objects produce term collisions we lose the direct mapping from bit string to a term. The term of the stored objects is required in order to detect when a term collision occurs. Therefore, we also store the term along with the object representations to avoid fetching the complete textual descriptions of the objects. Term collisions may eventually occur which is why we store the terms pro-actively for all values. The storing of terms requires more space depending on the number of term collisions. The experimental evaluations provide an overview of the storage requirements.

All objects are stored in sorted order with regard to the term frequencies. The insertion algorithm is given in Algorithm 16 which takes a dataset of spatio-textual objects and a grid level to process. With no bit string collision the object can be directly inserted as seen from Line 12 since no other object with the term exists. When a term is creating a term collision for the first time the same applies. If the cell contains less than targetedK objects for a given
### 6.4. PROPOSED SOLUTION

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>pizza(3,4), sushi(1,5)</td>
</tr>
<tr>
<td>00010</td>
<td>shoe(5,2)</td>
</tr>
<tr>
<td>0000000</td>
<td>pizza(5), sushi(5)</td>
</tr>
<tr>
<td>0000010</td>
<td>shoe(5)</td>
</tr>
<tr>
<td>0000100</td>
<td>pizza(3,1), sushi(1,3)</td>
</tr>
<tr>
<td>000110</td>
<td>shoe(1)</td>
</tr>
<tr>
<td>001000</td>
<td>pizza(4), sushi(4)</td>
</tr>
<tr>
<td>001010</td>
<td>shoe(4)</td>
</tr>
<tr>
<td>001100</td>
<td>pizza(2), sushi(2)</td>
</tr>
<tr>
<td>001110</td>
<td>shoe(2)</td>
</tr>
</tbody>
</table>

Table 6.1: Key-Value Database with targetedK set to 2

**Algorithm 16: Insert** (Dataset $D$, GridLevel $gl$)

```plaintext
1 /* Global variables: */
2 $db$ ← Key-Value Store ;
3 targetedK ← max obj. ref. stored in $DB$;  // conf. param.
4 /* Local variables: */
5 $bs$ ← empty bit string;
6 foreach $o \in D$ do
7     foreach $t \in o.doc$ do
8         $bs$ ← spatialEncode($gl$, $o.\lambda$) ;
9         $bs$ ← $bs \cup$ termEncode($t$) ;
10        value ← $db$.lookup($bs$) ;
11        if value is empty ∨ value does not contain $t$ then
12            $db$.insert($o$) ;  // no objects with $t$ exists.
13        else if value contains $t$ then
14            if $|value| < targetedK$ then
15                $db$.insert($o$) ;  // there is more room.
16            else if $tf(t,o.doc) > value.mintf$ then
17                $db$.sortAndUpdate($o$) ;  // more relevant.
```

Because we store the term frequency as part of the object representation, dynamic updates are supported without fetching the actual objects. This is seen from Line 16 where the term frequency of the new object is compared with the lowest term frequency of already existing object representations in the cell. Therefore, we do not need to fetch already inserted objects in order...
CHAPTER 6.

to maintain the targetedK most relevant objects of a cell.

**Query Processing**

We proceed by giving an exact query processing algorithm that uses the encoded objects in a key-value store. The TkSTR query has 3 arguments: a region, a keyword, and the desired number of objects in the result. In order to use the bit strings for query processing, we first encode the query region to match a relevant grid cell. Any cell describes at most targetedK objects except the cells at the finest level. The query region may be equal or smaller to a grid cell giving three different scenarios:

1. Q.k or more of the objects described by the cell are inside the query region.
2. Less than Q.k of the objects described by the cell are inside the query region and the cell contains less than targetedK objects.
3. Less than Q.k objects described by the cell are inside the query region and the cell contains targetedK objects.

When Q.k objects are described by the cell and they all are inside the query region they may be returned as the result since the objects described by the cell are sorted by term frequency and no object may be more relevant. With less than targetedK objects in the cell the objects inside the query region may be returned as a result since no more objects can exists at any lower level. However, with targetedK objects in the cell, more objects may exist at lower levels and no exact result can be provided without using lower cells.

Since a cell that covers the query region provides an exact result in the first two scenarios, we initially encode the query region to the smallest grid cell that covers it. More cells have to be examined when the initial cell cannot provide a result as seen from the third scenario. The query processing procedure is described in Algorithm 17. The first two scenarios are covered from Line 7–12. When the result set contains Q.k objects the loop in Line 13 is not entered since there are no candidate cells and the result set is returned. With a result set smaller than Q.k the result is only returned if the cell contains less than targetedK objects.

The third scenario is covered from Line 13. For each iteration the cells from the lower level are fetched until no better object can exist in any lower cells. Only cells from the next level that are overlapping Q.R are examined as seen from Line 16. Lower levels for cells that contain targetedK objects may have to be fetched as seen from Line 20. Cells with less than targetedK objects can have no lower cells with new objects and is therefore not refined. Thereby, dense cells with a large number of objects may be examined to the finest level whereas lower levels of sparse cells may not be fetched.
All objects that are inside \( Q.R \) are added to the result set in Line 19. Notice that the result set is ordered and maintains only \( Q.k \) objects. Lower levels are fetched for cells with targeted\( K \) objects when the result set contains less than \( Q.k \) objects. When the result set contains \( Q.k \) objects there may exist better objects in a lower cell. Consider the example in Figure 2 where the most relevant object is outside the query region. In the case where the targeted\( K \) most relevant objects are outside the query region, there may still exist an objects that can enter the result set. Therefore, the lower levels are fetched until no better object can exists inside the query region as seen from Line 25 or when the lowest level has been examined.

The query processing algorithm iterates through all objects in a cell before continuing to the next cell in the candidate set. The \( Q.k \) most relevant objects may be found by combining the first few objects of a number of cells. In this special case Algorithm 17 has a computational overhead of examining objects that are not relevant in order to provide the result. The algorithm may be modified to process the candidate cell in a parallel fashion similar to the Threshold Algorithm \[32\]. With this approach the cells with the most relevant objects is iterated first resulting in less computations. The pseudo-code is omitted due to space limitations.

**Example 16.** Consider the two queries \( Q_1 \) and \( Q_2 \) with \( Q_1.k = 2, Q_1.keyword = "pizza", Q_2.k = 2, Q_2.keyword = "shoe" \) and with the query regions from Figure 6.7: \( Q_1.R = R_1 \) and \( Q_2.R = R_2 \). We first run Algorithm 17 using the database from Table 6.1 and the query \( Q_1 \):

- Fetching key: 000000
- \( R: (3,4) \)

No object can exists in any cell with a better term frequency since \( Q.k \) objects are inside the query region (Line 11). The result of the algorithm is \((3,4)\). Next, we run the algorithm with the same settings for the query \( Q_2 \):

- Fetching key: 000100
- \( R: (5) \) (object 2 is outside \( Q.R \))

Since the result set is smaller than \( Q.k \) and the cell contains targeted\( K \) objects
Algorithm 17: STRQuery(Database DB, Query Q)

1 /* Global variables: */
2 targetedK ← max obj. ref. stored in DB; // conf. param.
3 /* Local variables: */
4 R ← result set with Q.k objects ordered by term frequency;
5 minCell ← smallest cell that covers Q.R;
6 candCells ← empty set of candidate cells;
7 foreach Object o ∈ minCellQ.keyword do
8     // Objects are sorted - most relevant first
9     if o is inside Q.R then
10        R ← R ∪ o; // Can only be the best
11 if |R| < Q.k ∧ |minCellQ.keyword| = targetedK then
12    candCells ← minCell; // More objects may exists
13 while candCells ≠ ∅ do
14    lowerCells ← empty set of cells;
15    foreach cCell ∈ candCells do
16        foreach cell inside cCell ∧ overlapping Q.R do
17            foreach Object o ∈ cellQ.keyword do
18                if o is inside Q.R then
19                    R ← R ∪ o;
20                if |cellQ.keyword| = targetedK then
21                    // More objects may exist
22                    if |R| < Q.k then
23                        // Provide Q.k objects
24                        lowerCells ← lowerCells ∪ cell;
25                    else if cellQ.keyword.mintf > R/Q.k.tf then
26                        // Something better may exist
27                        lowerCells ← lowerCells ∪ cell;
28    candCells ← lowerCells;
29 return R;

more relevant objects may exist (Line 11).

Fetching keys: 0000010 and 0000110 (overlaps with Q.R)
R: (5,1)

Both cells has less than targetedK cells. Thus, no more objects may exist (Line 20). The result of the algorithm is (5,1).
6.5 Experimental Evaluation

We proceed to evaluate the proposed solution using a standard DBMS and real datasets. First, we present our experimental setup followed by a description of the datasets. Finally, we describe a number of experiments.

Experimental Setup

All experiments are performed on a commodity machine with a 64-bit quad-core Intel i5-2520M (2.5 GHz) processor with 8 GB of main memory. We use a standard installation of PostgreSQL version 9.3.4 with a 8 KB page size. All data is loaded into a table with two columns (key, value) where the key column is set to primary key. The primary key is indexed as default with a B-tree. The storage mode of the value column is set to external to prevent compression when the data cannot fit into the 8 KB page. The proposed solution is implemented in a single-threaded Java application that executes SQL commands using JDBC for database communications. The data is read from an external disk and no data is maintained in main-memory by the Java application. We employ the default hashing function provided by the Java Framework.

We perform two sets of experiments. The first reports on the properties of the data stored in the PostgreSQL database when varying on targeted\(K\), the number of grid levels, and the hash value length using two datasets. The second evaluates the query processing algorithm when varying on the query arguments, dataset, and storage parameters.

Datasets

We collected all world-wide geo-tagged messages from the public Twitter Streaming API during February 2013. For this study, we create two datasets from these messages. The first dataset (termed GPCAT) simulates Google Places by filtering the Twitter messages with the 95 categories used by Google Places\footnote{http://developers.google.com/places/documentation/supported_types}. The first 100,000 messages that contain any of the category names in their textual description are added to the dataset GPCAT. The resulting dataset contains messages with 47 of the categories. Thus, the number of terms to encode to a bit string is 47. Since many users make use of services such as Foursquare that automatically reports on the location of a user, e.g., restaurants, parks, airports, etc., this dataset may reflect many of the actual POIs provided by Google Places.

The second dataset (termed FULL) is created by taking the first 100,000 messages from the Twitter dataset without filtering the textual descriptions. This dataset contains textual descriptions in any language containing any terms. In total it contains 53,247 unique terms.
With dataset GPCAT we may see a large number of bit string collisions since the messages are filtered by 95 categories in English. The dataset FULL may provide a large number of unique bit string since terms may exist in any language from anywhere in the world. Therefore, the two different datasets are used to show the properties of the indexing and query processing when the data varies.

**Storing Objects**

In the first set of experiments we evaluate the performance of the proposed storage and indexing technique. Also, we report on the properties of the stored data. We vary on a number of settings for all of the experiments.

**Varying Number of Grid Levels**

We aim at storing data such that reasonable sized query regions can be answered efficiently. The number of grid levels to consider in encoding the bit string influences the effectiveness. Therefore, we vary on the number of maximum grid levels such that we store both datasets GPCAT and FULL with a maximum number of grid levels that results in the finest cell being approximately 10, 100, and 1,000 meters. This corresponds to the grid levels 21, 18, and 14, which are represented by 45, 39, and 31 bits, respectively.

**Varying Hash Value Length**

The dataset GPCAT contains many objects that have a common term which may result in bit string collisions. The amount of bit string collisions vary with the length of the hash value. We propose to fit all bit strings into 64 bit which matches the supported memory addresses of our CPU. With the above mentioned maximum grid levels we have 19, 25, and 33 bits available for the textual part of the bit string. This provides a reasonable number of possible hash values considering the properties of the two datasets.

**Varying targetedK**

The parameter targetedK determines the number of objects to store for each term in each cell, and has influence on the effectiveness of the query processing algorithm and also depends on the setting of $Q.k$. We vary on targetedK such that we store the two dataset with targetedK set to 50, 100, 150, 200, and 250.

**Collisions**

In the first experiments we insert the datasets into the DBMS and report on the number of collisions. When a bit string representation of the location and the term of an object is stored it may already exists since another object may
6.5. EXPERIMENTAL EVALUATION

Figure 6.8: Storage Statistics for the GPCAT dataset
(a) Number of Bit String Collisions

(b) Number of Term Collisions

(c) Storage Requirements

Figure 6.9: Storage Statistics for the FULL dataset
contain the same term in the same cell. The larger the cell the more likely it is that another object exists with the same term. This is seen from Figures 6.8a and 6.9a that show the number of bit string collisions in both datasets.

For dataset GPCAT that contains only a few terms we see almost 100,000 collisions at grid level 0 since the objects with the same term in each of the cells will have the same bit string representation. As the cells becomes smaller the number of collisions decreases for both datasets since it is less likely that the objects are in the same cells. Notice that each of the three maximum levels has a different hash value length, but for dataset GPCAT the levels have the same number of collisions. This is because the dataset GPCAT contains very few terms resulting in no term collisions as seen from Figure 6.8b. This means that each unique term results in a unique hash value for all three hash value lengths.

Dataset FULL produces a different number of bit string collisions for the three maximum levels since the length of the hash value vary. As seen from Figure 6.9a there are many more collisions when the hash value is 19 bits compared to 25 and 33 bits. This is because of the number of unique hash values that increases with the number of bits available. Figure 6.9b describes the number of term collisions for dataset FULL, i.e., when different terms are hashed to the same value. With 25 bits hash values (maximum grid level 18) we already see a relatively low number of term collisions compared to the number of bit string collisions. Therefore, increasing the number of bits for the textual part of the bit string does not provide a large improvement in the number of term collisions for these settings.

Dataset GPCAT contains many more bit string collisions than dataset FULL as seen from Figures 6.8a and 6.9a because of the textual filtering. The filter enforces a limited number of English terms and therefore the objects are more likely to be located in same parts of the world compared to dataset FULL.

Storage Requirements

The object representations stored for each bit string may occupy more space than the page size of 8 KB. Figures 6.8c and 6.9c report on the space consumption for dataset GPCAT and FULL. The lower part of the bars reports on the amount of storage necessary to store data that fit inside a single page. The upper part of the bars reports on the amount of data that did not fit into a single page. As expected, less data in total is stored for small values of targetedK and for a low number of levels. With large values of targetedK we see an increase in overflowed pages for dataset GPCAT since more objects are stored for each term. The dataset FULL creates very few overflowed pages since there are few bit string collisions. Dataset GPCAT has many collisions which results in more overflowed pages. As seen from Figure 6.8c the number of
overflown pages increases with targeted\( K \) since more objects has to be stored in each cell.

The number of tuples required to store the datasets at each of the maximum grid levels is listed in Table 6.2. The dataset FULL contains many more tuples than dataset GPCAT since it has fewer collisions. This increases the number of unique bit strings.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Level 21</th>
<th>Level 18</th>
<th>Level 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPCAT</td>
<td>860,655</td>
<td>628,138</td>
<td>333,766</td>
</tr>
<tr>
<td>FULL</td>
<td>1,783,060</td>
<td>1,516,271</td>
<td>1,133,870</td>
</tr>
</tbody>
</table>

Table 6.2: Number of Tuples for the Datasets

Storage Time

In this experiment we examine the time to store the objects. As seen from Figures 10 and 11 the storage time increases with the number of levels to store since more tuples have to be created. In Figure 10 we also see and increase in the storage time for dataset GPCAT when targeted\( K \) is increased since it takes more time to store the additional objects. The dataset produces a large number of collisions resulting in more objects to store for each bit string.

In Figure 11 the storage time for dataset FULL is more constant since there are fewer collisions making it less sensitive to the size of targeted\( K \). Overall it is faster to process the FULL dataset even though it contains more tuples. This is do to the fewer collisions that increase the time to sort and update the order of the most relevant objects.
6.5. EXPERIMENTAL EVALUATION

**Fig. 11 & Table 5:** Storage Time and Index Size for FULL

<table>
<thead>
<tr>
<th>Level</th>
<th>Index Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>54</td>
</tr>
<tr>
<td>18</td>
<td>46</td>
</tr>
<tr>
<td>14</td>
<td>34</td>
</tr>
</tbody>
</table>

**Index Size**

The size of the indices does not vary with the value of targeted\(K\) since only the bit string is indexed. Therefore, only the number of bit strings to index has influence on the size of the index as seen from Tables 4 and 5. The index sizes for the two datasets follow as expected the number of rows seen in Table 3.

**Query Processing**

In this set of experiments we study the performance of the query processing algorithm. We employ both datasets and design two query sets for each of the dataset. For both query sets we create 1,000 queries for each parameter value, and we vary on \(Q.k\) and targeted\(K\). Also, we randomly choose a term for \(Q.keyword\) from the bag of all terms. Thereby, we will query the most frequent terms more often. We randomly choose a query region that may have a size between that of grid level 0 and 1 cm\(^2\).

**Dataset GPCAT**

For the dataset GPCAT we make all query regions cover New York City, USA in order to ensure we use an area with a large number of objects that contain the terms from the Google Places categories. As seen from in Figure 6.12a the query runtime increases when \(Q.k\) is close targeted\(K\) since more cells have to be fetched to produce a result. The number of fetched cell is shown in Figure 6.12b where the number of cells required to answer the query decreases when targeted\(K\) increases. The query runtime increases as expected with \(Q.k\) since more objects has to be examined. When the value of \(Q.k\) equals targeted\(K\) the query runtime increases since more cells have to be fetched in order to produce a result.
Dataset FULL

We allow the query region for the dataset FULL to be located anywhere in the world. As seen from Figure 6.13a the query runtime is similar regardless of the values of $Q.k$ and targeted$K$. Only a single cell lookup is required for all queries as seen from Figure 6.13b. This is either because of the few collisions that result in cell with less than targeted$K$ objects stored for most bit strings, or that the query region may find an empty region.

Varying Maximum Number of Levels

In the next experiment we study dataset GPCAT and use the same query sets as before but fix targeted$K$ to 250 and vary on the starting grid level. We force
a query to start at a given grid level and evaluate the runtime and number of fetched cells. As seen from Figure 6.14a, the query runtime increases when the starting grid level approaches 0. This is because of the large number of collisions at these grid levels as seen from Figure 6.8a. The same applies to the number of fetched cells as seen from Figure 6.14b. Almost the same number of cells are fetched when varying on $Q.k$ because $targetedK$ is sufficiently large. Therefore, even though $Q.k$ varies, the query may be answered with the same number of cells. This was also verified in the experiment shown in Figure 6.12b. Notice that there is a small computation overhead when $Q.k$ increases since more objects must be examined as seen from Figure 6.14a. At grid level 10 it is seen that a single cell is sufficient to answer the queries since few collisions exists at this grid level. We omit describing the same study for dataset FULL since the result can be returned with a single cell at any grid
level as seen from Figure 6.13b.

![Graph showing query runtime and number of fetched cells](image)

Figure 6.14: Query Runtime for Dataset GPCAT

### 6.6 Conclusions

We present a new framework for the processing of spatio-textual region queries. The framework leverages existing DBMSs and key-value stores, and it requires no new index structures to store and retrieve the data efficiently. The spatio-textual objects are encoded into a compact bit string that may be stored and indexed using an index that is standard in DBMSs and key-value stores. The spatial part of the bit string is obtained using a multi-layered grid-structure, and the textual part is obtained using hashing. To reduce the storage requirements, only a limited number of objects are stored for each grid cell. Query
processing works by combining these objects from multiple grid levels until an exact result can be returned.

An experimental study with real data offers insight into the properties of the stored objects and query processing performance. It demonstrates that the framework is capable of efficiently offering query results with existing DBMSs, and with different types of datasets.
Bibliography


BIBLIOGRAPHY


