NTNU
Norwegian University of
Science and Technology

Spatio-textual search on Spark

Tord Kloster

Master of Science in Informatics
Submission date: June 2017
Supervisor: Kjetil Nørvåg, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

The amount of spatially aware data is growing at a rapid rate, and challenges both processing and organizing such data is in focus in the scientific world and the industry. But spatial data seldom exists alone, usually accompanied by some form of textual property. The challenges increase as we attempt to process the spatio-textual documents that are created, and the usage of Big Data platforms become a necessity. This paper provides an insight into different approaches on how to meet the spatial challenges on Big Data platforms, and provides a way to extend a solution to a spatio-textual index on top of Apache Spark. The approach is evaluated to show good results on very large datasets.

Sammendrag

Mengden romlige data vokser stadig raskere, noe som skaper utfordringer både i den vitenskapelige verden og i industrien. Men ren romlig data eksisterer sjelden alene, da det ofte tilhører en eller annen form for tekst i tillegg. Utfordringene øker når vi må prosessere disse rom-tekstlige dataene som blir generert, og bruken av Big Data plattformer blir nødvendig. Dette arbeidet gir en innsikt i forskjellige løsninger på rom-utfordringene gjennom Big Data plattformer, og viser hvordan man kan utvide en slik metode for å løse disse rom-tekstlige utfordringene i Apache Spark. Det resulterende systemet har blitt evaluert, og viser god ytelse selv på veldig store datasett.

Acknowledgements

I would like to thank everyone involved with this paper, including my supervisor Kjetil Nørvåg who with candid help and support have guided me through this process. I would also like to thank my friends especially Jama Noor for supporting and encouraging me through the year, as well as my family for always being there for me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With social media being a huge part of anyone's life (Duggan and Brenner, 2013) we are sharing increasingly more and more life events, and as such the data volume in these social media platforms is also increasing. Event data that have a focus on personal life experiences usually contains some form of text and a location or place associated with the event. Photos uploaded to Flickr or Instagram allows the users to check-in to a location when posting, and smartphones are used as a geolocating service (Fox et al., 2013). With the increasing amount of location aware devices, social media platforms and willingness to share personal information, geographically tagged data are increasing rapidly. Tweets are another example of the data we are sharing through social media and often has a geographical tag in addition to the 240 text limit. Posts on Facebook can be geotagged to show where in the world you are posting from, and virtually any social media has some form of indicating to the world where their posts originates from.

As the volume of spatial data grow, traditional RDBMs can not keep ut with the sheer amount of data generated (Fox et al., 2013). Researchers and the industry has to figure out new ways of operating with data on a much bigger scale. Big data platforms have been created to handle the enormous generation of data (Dean and Ghemawat, 2008), and have previously been extended to support spatial data as well (Eldawy and Mokbel, 2015).

However there are even more generators for geographically tagged data than just social networks. GPS-logs from taxis in Shanghai and Shenzhen have been data mined by Tan et al. (2012), where the data are increasing by 1-2GB per day. NASA satellite data archives contains over 500TB spatially aware data that grow each year (Eldawy and Mokbel, 2015) and a major requirement for a system that manages all this data is a spatial architecture that delivers a quick query response on a scalable level (Aji et al., 2013).

Spatial data processing is then obviously a challenge, and there exists many different techniques to deal with the problem, but spatial data seldom come alone. The spatial objects generated are usually accompanied by some form of textual data as well. Consider spatially tagged twitter messages that was previously mentioned, which comes with an important textual description as well. Life events from personal lives in any social media platform usually have a way of spatially tagging the events, containing some description of

the situation at hand. Web searches on the Internet usually contain some form of spatially tagged data combined with some search terms.

A well defined field of spatio-textual indexes have already been established (Chen et al., 2013) for standard datasets, but extending these methods to Big Data platforms have yet to happen on a big scale. The construction of spatial indexes have been introduced to these platforms (Eldawy and Mokbel, 2015), however there still are not a huge influx of different approaches to the spatio-textual problem as of this moment.

There are obvious challenges to overcome when the data size increases over what a normal relational database is able to handle. Especially when working with data mining, and extracting information from the spatio-textual data. When the size of the input exceeds the capabilities of a standard centralized relational database the only way to remedy the problem would be to vertically extend the hardware of the database. Utilizing a decentralized scalable platform, like a implementation of the Map Reduce paradigm, would be both increase performance and be cost effective.

My motivation for this project is to explore the current implementations of spatial and textual indexes on Big Data platforms, and create a spatio-textual index on top of Apache Spark. As Apache Spark utilizes memory techinques to out perform standard Hadoop, a functional spatio-textual index would be valuable, as we can use the scalability to analyze very large datasets. In datasets such as tweets collected from Twitter, one can argue that there exists just as much information in the spatial and textual data. By exploring different indexing structures both within the domains of textual and spatial indexing, the goal is to find a combination that works well when applied to Apache Spark.

1.1 Problem Definition

In spatio-textual search, not only text but also location is part of the query, and results are ranked according to similarity to both of these. In order to perform such queries efficiently, spatial inverted indexes are used. We are interested in studying how such queries can best be supported in a system based on Spark

1.2 Research Questions

RQ1 What approaches exists in context of Spatial indexing on Big Data platforms?

RQ2 How can these be extended to spatio-textual?

RQ3 How can we implement these spatio-textual approaches on Spark?

1.3 Thesis Overview

Chapter 2 Contains preliminaries about Big Data platforms, introduces Apache Spark and provides an explanation of Geohash.

Chapter 3 Defines some basic concepts, theory about standard textual and spatial indexes, and what combination schemes exists. A comparison of different spatio-textual approaches is presented.

Chapter 4 Explores the related works in the field, and provides an overview of the research.

Chapter 5 Contains an overview of different approaches to spatial indexing on Big Data platforms, and provides a summary of the mentioned systems.

Chapter 6 Describes the approach of this paper, and details the important aspects and contributions the system provides.

Chapter 7 Presents results from the evaluation regarding the system, and provides performance results based on the different datasets used.

Chapter 8 Sums up the evaluation, answers the remaining research questions and suggests some topics for further research.

Chapter 2

Preliminaries

In this chapter the MapReduce framework will be introduced to the reader, and the open source implementation, Hadoop, will also be briefly presented. Apache Spark is then detailed, with an introduction to it's main modules especially Spark SQL. We will end the chapter by presenting a geocoding system named Geohash, and explain how it works.

2.1 MapReduce

MapReduce is a programming model and associated implementation for processing large datasets created at Google in 2003. Users create a map function that process some key/-value pairs into intermediate key/value pairs, and a reduce function that combine all the intermediate values with the same key (Yang et al., 2007). This model automatically allows the operation to be performed in parallel on a cluster (Dean and Ghemawat, 2008). This means that the users have to express every operation they wish to perform as a map-reduce implementation, which enables the application to run in parallel on a cluster.

Map Function

$map(k^1, v^1) \rightarrow n(k^2, v^2)$
A map function takes as parameters a key/value and produces some n other key/value pair(s). The library then groups all key-value pair and sends them to the reduce function for further processing.

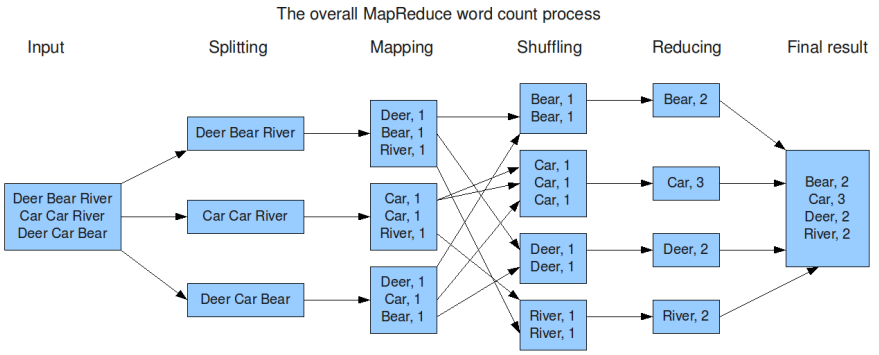
Reduce Function

$reduce((k^1, v^2), (k^1, v^3)) \rightarrow (k^1, v^4)$

A reduce function combines two key/value pairs of equal key, and produces one record with the corresponding key and applies a user defined function to merge the values.

On the example in Figure 2.1 the input is put through an initial splitting stage followed by a mapping stage where the maps are then shuffled before they are reduced and merged to the final result. Shuffling is an intermediate step that groups records of the same key, before the reduce function is applied.

Figure 2.1 Word count, map-reduce example [1]



The workflow of a MapReduce application starts with splitting the input files into *M* splits and multiple copies of the application is started in the cluster. The copies of the applications are all worker nodes, except one instance called the master. The master assigns map or reduce tasks to each idle worker. Worker nodes periodically writes the results of the tasks to disk, and passes the location of the results back to the master. As other worker nodes need the results they acquire it by using remote procedure calls to read the data. When the final reduce task is complete *R* output files are created, one per reduce task, and stored on disk. At this point, the MapReduce application returns to the user's code again.

2.1.1 Hadoop

Apache Hadoop is an open source framework for computing large datasets reliable, scalable and in a distributive manner. The framework is derived from Google's MapReduce and Google file system(GFS) (Ghemawat et al., 2003) and as such is fundamentally similar.

Hadoop File System

HDFS is a distributed file system based on GFS, designed to be fault tolerant and to run on commodity hardware. The goal of using HDFS is to store data distributed among the entire cluster. There are two main parts of HDFS as it is a master/slave architecture: the master called NameNode, and the slaves called DataNode. The NameNode is responsible for the namespace of the file system and file acces. The DataNodes are responsible for managing

[1] https://cs.calvin.edu/courses/cs/374/exercises/12/lab/ [Accessed: 03-May-2017]

the node they reside on, this includes handling read/write requests from the clients and replicating data on request from the NameNode.

The files stored in HDFS are split up and distributed across the whole cluster. Data can also be replicated which increases fault tolerance, but also improves performance when reading the data. As the read requests are received the most local replication will be serving the requests to save network bandwidth.

2.2 Spark

Apache Spark is a system for general purpose big data analytics. It is a quick, general purpose cluster computing system that provides high-level APIs in Java, Scala, Python and R. Apache Spark is highly scalable and as an example, a large Internet company uses Spark and it's associated module, Spark SQL, to build data pipelines and run queries on an 8000-node cluster with over 100 PB of data(Armbrust et al., 2015). Developed by UC Berkley in 2010 the purpose of the system was to support more applications than MapReduce by improving multi-pass operations such as:

Iterative algorithms like machine learning, and graph algorithms

Interactive data mining by loading data into ram and performing multiple queries

Streaming algorithms that can maintain an aggregate over time

At the core of every spark application is a Driver program that launches the parallel operations described in the user code. It submits the operations as tasks and distributes these to Executors on the cluster. Executors are processes that are launched on the worker nodes to complete tasks assigned by the Driver.

Fundamentally, Spark utilizes the same idea as MapReduce and Hadoop but extends the system to support applications that cannot be expressed efficiently as acyclic operations. The main abstraction in Spark is the introduction of the Resilient Distributed Dataset,)(RDD) which is a read-only representation of objects that are distributed across the cluster.

A RDD is computed lazily, and initially only the transformations for how to create the RDD is stored. Only when a RDD needs to return some data to the Driver, for example to show some results, is the RDD computed. The RDDs can be cached in the memory of the machines in the cluster, which enables it to be used in multiple MapReduce-like parallel operations (Zaharia et al., 2010) without being recomputed each time. This is the biggest advantage Spark achieves over MapReduce and Hadoop, and outperforms Hadoop by a factor of 20 when processing interactive applications (Zaharia et al., 2010). The possibility to compute an intermediate RDD, perform multiple and possibly various transformations on the data without recomputing the base multiple times is a big improvement to these types of operations.

There are multiple extensions of Spark, also called modules, that improves the functionality of specific areas in the field. The main four modules are: Spark SQL (Armbrust et al., 2015), Spark Streaming (Zaharia et al., 2013), MLib (Machine Learning) (Meng et al., 2016) and GraphX (Xin et al., 2014). As we are using Spark SQL to implement our final application, this module will be explained in the next section.

2.2.1 Spark SQL

Spark SQL is a module for Spark that integrates relational processing with Spark's functional programming API (Armbrust et al., 2015). Spark SQL extends the core Spark system by two main additions:

Relational processing , with a tighter integration between Spark SQL's declerative Dataframe API and Spark's procedual core.

Optimization through the implementation of a query optimizer called Catalyst, which is extensible and provide users with the possibility to add optimization rules.

Spark SQL provides a Dataframe API that can perform relational operations on both external data as well as standard Spark collections. Dataframes are computed lazily, similarly to RDDs, so that relational optimizations are performed before computing.

While standard spark provides a functional and general API, it does not offer any great opportunities to perform sophisticated optimizations. With the introduction of the optimizer Catalyst (Armbrust et al., 2015), automatic optimization is improved and provides up to 10x faster computations than standard Spark in operations that can be expressed as SQL statements.

Spark SQL is built based on the previous attempt to create a relational interface for Spark, Apache Shark (Xin et al., 2013), which had a couple of difficult to address challenges.

2.2.2 Partitioning

Spark can partition the data on a column in the dataset. Using a some string as the partitioning value we can distribute the records in such way that we can prune away the partitions that definitely does not contain the string we are querying for. Spark can store each dataset in a file format known as parquet (Apache, 2017) to store the partitions on disk. This file format keeps the partitioning scheme by storing each partition in a separate folder.

2.3 Geohashing

Geohash is a geocoding system that was originally created by Gustavo Niemeyer in 2008 to represent coordinates in a url friendly manner (Fox et al., 2013). To accomplish this, a space filling curve is utilized to map a coordinate in $(latitude, longitude)$ format into a one dimensional string. The general type of space filling curve used is a Z-Order curve, but the system can use other space filling curves as a replacement. The resulting geohash then represents a spatial bounding box (Balkić et al., 2012) in the spatial domain. Despite being intended for representing geographical data in URLs, Geohashes have turned out to be very useful as a way of indexing spatial data when it comes to databases. This is especially useful in Apache Spark, as we can use this value as our column to partition the index.

There are a few important characteristics in regards to a geohash which we are going to exploit when querying the index:

- Each rectangle can be interpreted as a latitude longitude rectangle. This is true as we will explain later because the geohash is constructed by a series of divisions on the full domain ([-180, 180] x [-90, 90]).

- Adding a character to the end of a geohash means further dividing the rectangle, and as such all geohashes that extend another geohash while sharing prefixes are located within the primary rectangle.

- Geohashes that share prefixes are considered to be close. This only applies one way, meaning close geohashes does not need to share prefix.

2.3.1 Decoding

When decoding a geohash, for example "gcpuz", which is located in the middle of London, the string is decoded into binary from the following base32 character map.

Figure 2.2 Base32 Character map

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base 32	0	1	2	3	4	5	6	7	8	9	b	c	d	e	f	g
Decimal	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Base 32	h	j	k	m	n	p	q	r	s	t	u	v	w	x	y	z

The geohash is read from left to right and decoded character by character into the binary: 01111010111010111011011111.

The latitude and longitude are interleaved, which means that reading every even bit creates the latitude binary, and every odd bit creates the longitude binary.
$longitude = 0111111111111$
$latitude = 110010010011$
Each bit in the respective binary string describes a division based on the minimum and maximum value of the coordinate system. In the case of geohash there are different values for latitude and longitude, as latitude ranges from -90 to 90, while longitude ranges from -180 to 180.

To calculate the correct latitude coordinate we divide the interval -90 to 90 by 2 which produces another 2 intervals $[-90, 0]$ and $[0, 90]$. As the first bit in the latitude binary is a 1 we discard the $[0, 90]$ interval and repeat by diving $[-90, 0]$ by 2 and selecting the correct interval to continue with according to the current bit in the binary sequence. A calculation of latitude by decoding the geohash "gcpuz" can be observed in the example listed in Table 2.1.

Minimum value	middle value	Maximum value	bit
-90	0	90	1
0	45	90	1
45	67.5	90	0
45	56.25	67.5	0
45	50.625	56.25	1
50.625	53.438	56.25	0
50.625	52.031	53.438	0
50.625	51.328	52.031	1
51.328	51.680	52.031	0
51.328	51.504	51.680	0
51.328	51.416	51.504	1
51.416	**51.460**	51.504	1

Table 2.1: Computation of latitude from Geohash

2.3.2 Encoding

Similarly, encoding a $(latitude, longitude)$ coordinate starts with comparing the middle value of the starting interval $[-90, 90]$ to the latitude value. When the middle value is bigger than the target, the lower bound interval is used, and the higher bound interval is used if the middle value is smaller. An example is provided in Table 2.2. After completing this task for both latitude and longitude the binary sequence for the latitude, longitude pair is interleaved before it is converted to Base32.

Minimum value	Middle value	Maximum value	$51.460 >$ Middle value
-90	0	90	1
0	45	90	1
45	67.5	90	0
45	56.25	67.5	0
45	50.625	56.25	1
50.625	53.438	56.25	0
50.625	52.031	53.438	0
50.625	51.328	52.031	1
51.328	51.680	52.031	0
51.328	51.504	51.680	0
51.328	51.416	51.504	1
51.416	**51.460**	51.504	1

Table 2.2: Computation of binary sequence from latitude

Since Geohash is an implementation which is based on the geographic coordinate system, the $latitude, longitude$ format, the distance between two geohashes does not represent real distance, but distance between the lat/long coordinates. Cell dimensions vary with latitude, so the worst case scenario can be observed in Table 2.3

Number of characters	Area Width		Area Length
1	5,000km	x	5,000km
2	1,250km	x	625km
3	156km	x	156km
4	39.1km	x	19.5km
5	4.89km	x	4.89km
6	1.22km	x	0.61km
7	153m	x	153m
8	38.2m	x	19.1m
9	4.77m	x	4.77m
10	1.19m	x	0.596m
11	14.9cm	x	14.9cm
12	3.7cm	x	1.9cm

Table 2.3: Worst case (Cell size) for Geohash is at the equator

Geohashes have been used in current systems to index spatial data, among them are: (Lee et al., 2014)s spatial query on HBase, Elasticsearch [2] and MongoDB [3].

[2]https://www.elastic.co/guide/en/elasticsearch/guide/current/geohashes.html [Accessed 15-May-2017]
[3]https://docs.mongodb.com/manual/core/geospatial-indexes/ Accessed 30-May-2017

Chapter 3

Working with Indexes

In this chapter we will cover the basics of textual and spatial indexes. Starting with the definitions of the dataset, different types of queries that are discussed and various index structures. There are a lot of different types of both spatial and textual indexes and this chapter will highlight some of the various combinations when considering a spatio-textual index.

3.1 Definition

The literature in this area usually define spatio-textual objects with a $(latitude, longitude)$ value and a textual description. There are three main ways of combining spatial and textual indexes: text-first loose, spatial-first loose or tightly combined combination scheme (Chen et al., 2013).

Example Dataset

D is a geo-textual dataset. An object, $o \in D$, is defined as a pair of spatial coordinates, and the textual description. More formally $o = \{l, t, i\}$, where $o.l = \{o.latitude, o.longitude\}$ and $o.t$ is a textual description of the geo-textual object represented as a text document. $o.i$ is the associated ID to the object.

TkQ

Top-k kNN query takes three arguments, $TkQ = q\{w, p, k\}$ where $q.w$ is a set of query keywords. $q.p$ is a spatial point of where the result should be compared to and $q.k$ is the number of results that should be returned. Top-k kNN finds the top k objects from the similarities of both keywords and spatial point.

Specifically the ranking score of an object o for a TkQ q can be defined as such: $ST(o, q) = \alpha \cdot SDist(o.l, q.p) + (1 - \alpha) \cdot TRel(o.t, q.w)$ Where $SDist$ is the spatial distance between the object $o.l$ and the query spatial point $q.p$ calculated by using the

normalized euclidean distance. $TRel$ is the textual relevance between the object $o.t$ and the query keywords $q.w$ and is calculated with an information retrieval model, in this case the language model (Chen et al., 2013). As this is only an example note that there exists different approaches to ranking the spatio-textual objects.

BkQ

Boolean kNN query takes the same arguments as TkQ, $BkQ = q\{w, p, k\}$ and ranks the objects that contains the keywords $q.w$ based on the spatial point. The query returns the top k results of the ranking.
The query ranking is defined as: $\forall o \in q(D)((\nexists o' \in D \backslash q(D))(dist(o'.l, q.p) \leq dist(o.l, q.p)) \wedge q.w \subseteq o'.t)$(Chen et al., 2013)

BRQ

Boolean Range Query takes two arguments $BRQ = q\{w, r\}$ where w is a set of keywords, and r is a spatial region. For q(D) the result of BRQ is a subset of D, where the objects $\forall o \in q(D)(o.l \in q.r \wedge q.w \subseteq o.t)$ Or in other words, the result are objects that are contained withing the query region and contains all the query keywords.

3.2 Textual Indexes

The value of any information retrieval system can be expressed as a variety of factors, such as processing power, disk efficiency and the quality of the returned data. Using an index to improve the efficiency of the queries we want to perform is valuable as it increases at least two of these factors.

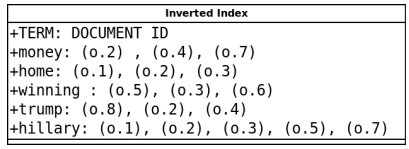
3.2.1 Inverted Index

An inverted index is a textual indexing structure where each occurring term t is mapped to the documents $o \in D$ containing the term. There are two main variants of inverted index:

Record-level index which maps each word to the corresponding document ID

Word-level index which extends the record-level with the position of each word in the document.

Figure 3.1 Example of an Inverted Index



3.2.2 Bitmaps

Bitmaps can be used as a textual index, where the each bit indicates whether the object contains the term or not. Some variations of the index exists where each term have a bitmap to indicate which document contains the term.

	(o.1)	(o.2)	(o.3)	(o.4)	(o.5)	(o.6)	(o.7)	(o.8)
money	0	1	0	1	0	0	1	0
home	1	1	1	0	0	0	1	0
winning	0	0	1	0	1	1	0	0
trump	0	1	0	1	0	0	0	1
hillary	1	1	1	0	1	0	1	0

Table 3.1: Bitmap example

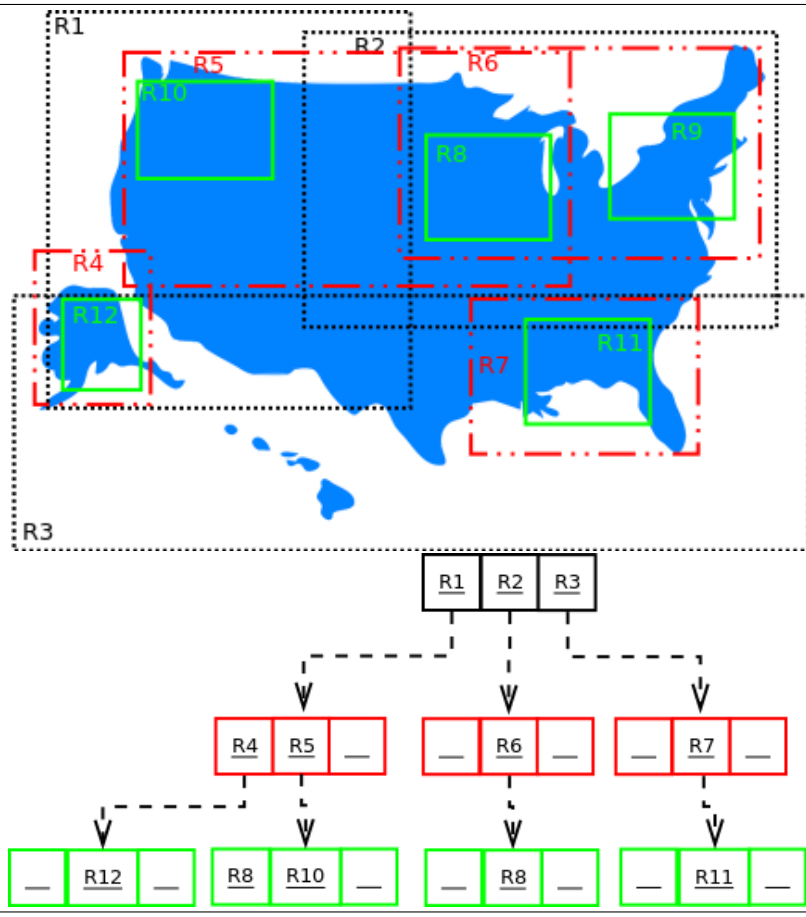
3.3 Spatial Indexes

As we observe that more and more spatial data are generated, we want to efficiently process the data by utilizing indexing strategies. Creating spatial indexes usually comes down to expressing a spatial point or area in such a way that a query algorithm can traverse and return the result efficiently.

R-tree

A R-tree has similar structure to a B^+-tree and stores multidimensional rectangles. Non-leaf nodes store the bounding rectangles for its sub-tree (Beckmann et al., 1990), providing an efficient way of checking whether the sub-tree contains any close objects. The structure must allow for overlapping rectangles, meaning it is not guaranteed to provide only one search path through the tree.

Figure 3.2 Example of a R-tree



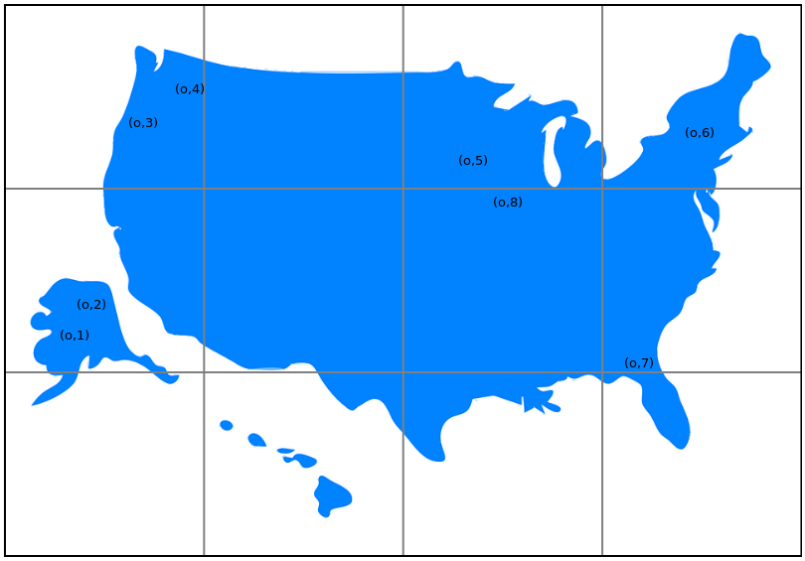
R*-tree

A R*-tree is a version of the R-tree where the overlap of the rectangles are optimized by implementing a new split algorithm and a forced reinstate (Beckmann et al., 1990). The forced reinstate improve the overlap by deleting ”old” entries and inserting them again. This reduces the overlap of the bounding and leaf query regions.

Grid

Grid indexes divide the spatial domain into a number of equally sized square or rectangles. The documents are then associated with the grid cell based on the spatial data. Working with grid indexes are usually more easy as the grid can be created before processing any input, and the structure does usually not need to change during processing.

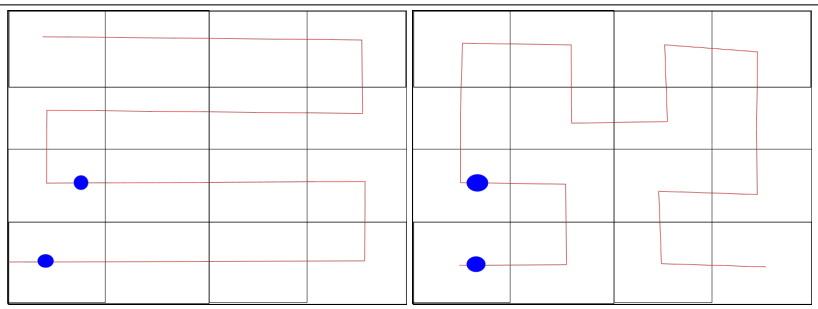
Figure 3.3 Example of a Grid index



Space Filling Curves

Space filling curves are continuous lines that cover the entire space of an plane. The idea of a SFC is to place the points that are close to each other in space, together on a curve. This way we can represent spatial distance as points on a line. There are some common SFCs in the literature, the most popular ones being Z-curve and Hilbert curve (Christoforaki et al., 2011). As we will see in the next section space filling curve in the context of spatio-textual indexing schemes can be combined with inverted files (Chen et al., 2013). This allows for skipping large parts of the list when querying with spatial information.

Figure 3.4 Points on a normal line compared to points on a space-filling-curve(Hilbert)



3.4 Spatio-textual Indexes

As data with both spatial and textual information increases the need for processing the queries of both spatial and textual parameters increases. Vaid et al. (2005) described a study of the Excite engine, where one fifth of all searches included some form of spatial awareness. This includes search terms that was considered geographical: a postal code, any directional qualifier, place name etc. To service such requests the need for an index to handle both spatial and textual arises, and we will introduce a couple of examples below. The formal definition of the spatio-textual problem can be defined as: given a set of spatio-textual objects $o \in D$, where each object contain a spatial location, a textual description and an ID $o = \{l, t, i\}$. We aim to combine a textual index based on $o.t$ and a spatial index based on $o.l$ in a way that we can perform a query to retrieve the objects based on both $o.t$ and $o.l$.

3.4.1 Text-First Loose Combination Scheme

Text-first combination first employ a textual index as the top level index, and then arrange the result in the leaf nodes as a spatial index. An example is using an inverted file as a top level index, and then arranging the postings in the inverted file in a R-tree.

Index	Spatial Part	Textual Part	BkQ	TkQ	BRQ
Text Primary Index (TS)	Grid	inverted file	text-first		✓
Inverted-File R*-tree	R*-tree	inverted file	text-first	Δ	✓

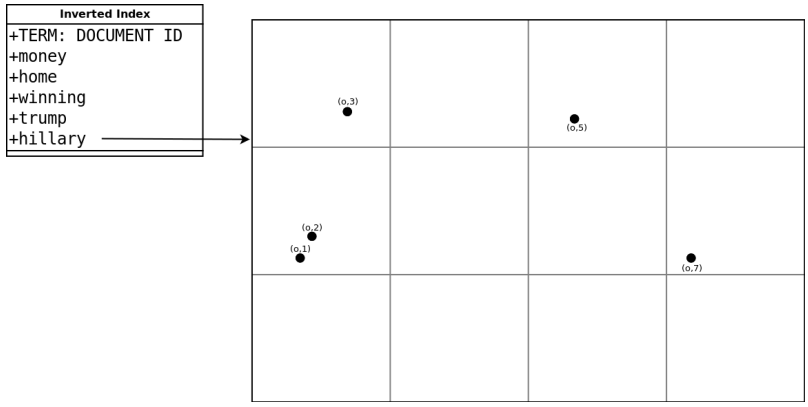
Table 3.2: Comparison of existing textual-first spatio-textual indexes (Chen et al., 2013)

Text Primary Index (TS)

TS (Vaid et al., 2005) is one of the first grid-based spatio-textual indexes, and is classified as a text-first loose combination in terms of it's combination scheme (Chen et al., 2013). TS consists of a modified inverted file, where each indexed term is associated with cell-specific sub-lists to represent their location (Vaid et al., 2005). The index can only provide BRQ, as no simple extension of the indexing scheme is better than using a normal inverted file (Chen et al., 2013).

The index is quite simple in the fact that the textual part of the document is indexed with an inverted index. The spatial domain is divided into a grid, and each inverted file is assigned the best matching grid cell. In the evaluation by Chen et al. (2013), TS was one of the better text primary indexes for space requirements, using 12GB to generate the index for the biggest dataset. However grid-based indexes are performing worse compared to the other indexing strategies (Chen et al., 2013).

Figure 3.5 Example of the TS index

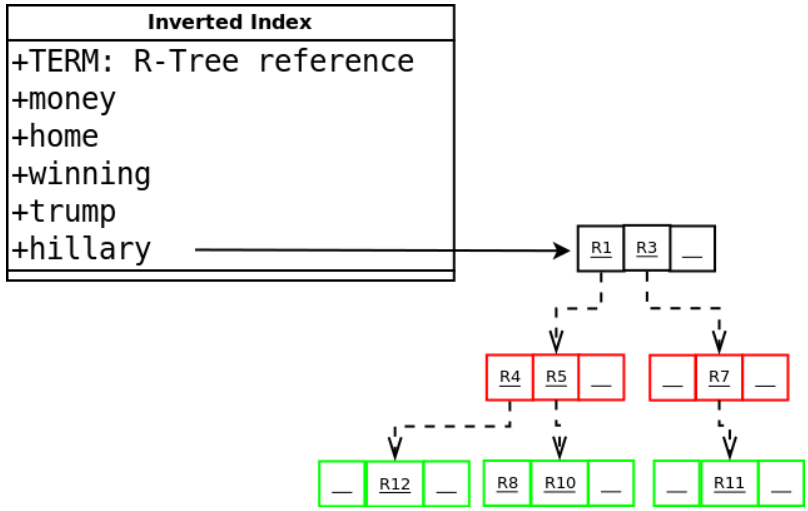


Inverted File R*-tree

IF-R* (Zhou et al., 2005) utilizes a text-first combination scheme with an inverted file as the top level index. Each distinct term t in D has a R*-tree which is used to keep the objects o that contains the term t (Chen et al., 2013).

The performance of the index is tied to the number of query terms. An increase above 1 affects the performance negatively. There also exists variations that replaces the inverted index with bitmaps, but these perform worse overall (Chen et al., 2013).

Figure 3.6 Example of the IF-R* index



3.4.2 Spatial-First Loose Combination Scheme

Spatial-first combination indexes are using a spatial index as primary index, and a textual index as leaf node.

Index	Spatial Part	Textual Part	BkQ	TkQ	BRQ
ST	Grid	inverted file			✓
R*-IF	R*-tree	inverted file		Δ	✓
SF2I(Chen et al., 2006)	SFC	inverted file			✓
SKI (Cary et al., 2010)	R-tree	bitmaps	✓		✓

Table 3.3: Comparison of existing spatial-first spatio-textual indexes (Chen et al., 2013)

Spatial Primary Index (ST)

ST (Vaid et al., 2005) is very similar to TS, combining a grid based spatial index with an inverted file. However in ST, the spatial index is at the top level, and an inverted index is constructed at each cell that contains all the documents that has a footprint in the corresponding cell (Vaid et al., 2005). Compared to TS, ST happens to perform consistently worse (Chen et al., 2013).

R*-IF

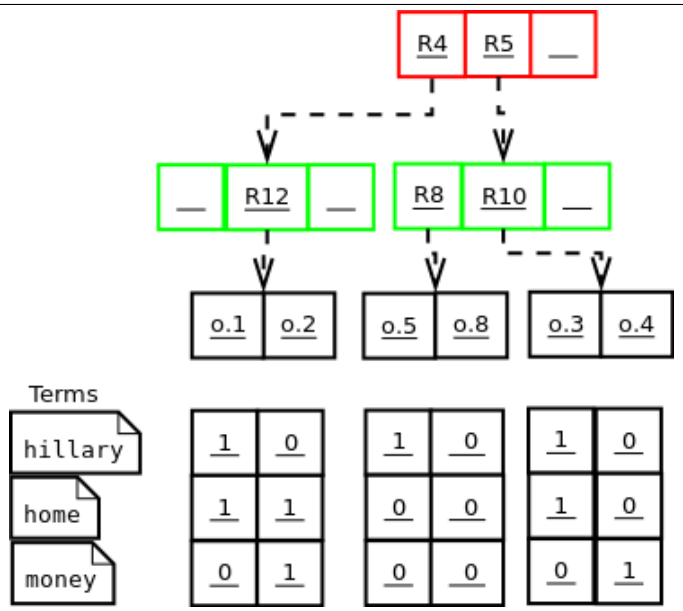
The R*-IF (Zhou et al., 2005) index is the counterpart of the IF-R* index, and uses a R*-tree to store all the documents based on the spatial data, an inverted file is created at each leaf node. R*-IF is therefore considered as a spatial-first index. The objects $o \in D$ are stored as a R*-tree. At each leaf node an inverted file is used to consider the textual properties $o.t$ of the objects (Chen et al., 2013).

It is shown that IF-R* performs better than R*-IF for BRQ(Chen et al., 2013), and as such IF-R* should be preferred over R*-IF.

Hybrid Spatial-Keyword Indexing (SKI)

SKI (Cary et al., 2010) uses an extended R-tree for the spatial part, and a bitmap version of a inverted file. The R-tree is extended in the sense that the non-leaf nodes contains a bitmap of what terms are contained in the leaf nodes in the subtree (Chen et al., 2013). Specifically every term $t \in D$ has a bitmap with length of the documents located at the leaf nodes, with the value of 1 if the corresponding object contains the term t, and 0 otherwise.

Figure 3.7 Example of the SKI index



3.4.3 Tight Combinations

Tight combination focuses on using both the textual and spatial index to prune the query area simultaneously. There have been used 2 different strategies in the following indexes, the first which integrates a text summary into each node of a spatial index, the second with spatial information injected in each textual index (Chen et al., 2013). SFC-QUAD is an example of the latter.

Index	Spatial Part	Textual Part	BkQ	TkQ	BRQ
KR*-tree(Hariharan et al., 2007)	R*-tree	inverted file	Δ		✓
IR^2 (De Felipe et al., 2008)	R-tree	bitmaps	✓		Δ
IR-tree (Cong et al., 2009)	R-tree	inverted file	Δ	✓	Δ
IRLi-tree (Li et al., 2011)	R-tree	inverted file		✓	
SKIF (Khodaei et al., 2010)	Grid	inverted file			✓
WIBR-tree (Wu et al., 2012)	R-tree	iverted bitmaps	✓		Δ
S2I	aR-Tree	inverted file	Δ	✓	Δ
SFC-QUAD	SFC	inverted file			✓

Table 3.4: Comparison of existing tightly combined spatio-textual indexes (Chen et al., 2013)

SFC-QUAD

SFC-QUAD uses an inverted index at the top level, and sorts the document IDs according to their Z-curve position (Chen et al., 2013). The use of this space filling curve accurately sorts the documents based on the spatial data. SFC-QUAD also uses a quad-tree structure to enable skipping. For a query, the quad-tree is traversed and m document ranges are found which contains all the documents that satisfy the query location. The query processer, which is an inverted file, then only accesses these ranges, and can skip the rest of the documents (Christoforaki et al., 2011). Finally objects containing all the query keywords are found through document-at-a-time processing along with forward skip optimization. SFC-QUAD is only suitable for BRQ (Chen et al., 2013).

S2I

S2I (Rocha-Junior et al., 2011) is an index that maps each term to an aggregated R-tree or a block based on the frequency of the index. Using a inverted file as primary index, each term t points either to a block, or an aR-tree. There are three main components to the S2I index: **the vocabulary** which describes the frequency for each term, a flag indicating whether the storage method is a block or a aR-tree, and a pointer to the aR-tree or block. **Blocks** are used to store objects that have a low frequency. For each object, the id, the impact of the term in the corresponding document and spatial location is stored. **Trees** The aR-tree use the same structure as a normal R-tree, but in addition keeps a aggregated value in the form of the maximum impact of the corresponding term.

Single-keyword algorithm (SKA) and Multiple-keyword algorithm (MKA) (Rocha-Junior et al., 2011) are two top-k spatial keyword queries that where developed to exploit the S2I index. The approach showed to outperform the DIR-tree (Cong et al., 2009) in both query and update.

Chapter 4

Related Work

In this chapter a overview of different papers in the related field of spatio-textual indexing will be presented. Different approaches to the current problem will be explored, including other related research.

4.1 Methodology

Both for this chapter and the next, articles where gathered through research document aggregators on the Internet. For this chapter, concepts describing both spatio-textual and spatial indexes where used to search for research papers as (Oates, 2005) suggests. As for the next chapter concepts describing Big Data platforms, and spatial indexing where used. After obtaining a set of papers, the papers where processed by skimming through and selecting the most important documents to be read carefully through. Another pass of this method yielded a set of research papers that describe a part of the current state of the art.

4.2 Review

When processing all of the different research papers, the amount of different index strategies are clearly showing. There exists a lot of work in indexing textual and spatial data, both in their own individual state, but also as spatio-textual objects. The most common approach when constructing spatio-textual indexes has been to combine the most used indexes from each domain. Inverted index from the textual side, and R-tree from the spatial side.

State of the art spatio-textual indexes combine these indexes but by constructing a custom combination scheme are able to improve performance significantly over the rest. By utilizing properties of term frequencies in documents and utilizing aR-trees, Rocha-Junior et al. (2011) created a spatio-textual index that outperforms the previous state of the art

indexes in the field. The S2I index, maps each occurring term in a document to either an aggregate R-tree or a block depending on the frequency of the term. Efficient algorithms to exploit this index is also part of the contribution, and provides top-k spatial keyword query.

As an overview and comparison between a multitude of different types of spatio-textual indexes Chen et al. (2013) found through extensive evaluation that if the target query method is BRQ, SFC-QUAD beats all the other indexes in both query execution time and index size. Grid based indexes where also reported to be unfavorable, when dealing with BRQ. IF-R* created the largest index in size, using 173GB to create a index on a dataset called TWITTER containing 20M objects. Consistently it was showed that R-tree based text-first require much more index space as objects will be replicated many times.

There exists a lot of different approaches when indexing spatio-texutal data however, and Christoforaki et al. (2011) explores new and efficient algorithms for improving query processing by utilizing space filling curves and inverted indexes. The paper has a focus on web searches and puts higher priority on the textual part of spatio-texutal queries. Instead of focusing on the fine grained attributes of some spatial structures, more coarse grained approaches improves performance. This approach can lead to improvement in both run time and index size, as the overhead associated with structures like R-trees are reduced.

As for addressing the question whether to utilize a spatial-first or textual-first index, Christoforaki et al. (2011) created a naive textual first approach that improved performance compared to other spatial indexes such as R-tree. They argued that as there is significantly more textual than spatial data in current geo search engines, it would be important to focus more on the textual part of the problem. The Text-first index was created using an inverted index, complimented by state of the art IR query processing techinques, such as compressing the document ids and frequencies using the OPT-PFD(Optimized PForDelta) algorithm which is an efficient way of compressing very large collections (Yan et al., 2009). The Text-first baseline was extended to include a coarse grade spatial structure based on kd-trees and space filling curves (Christoforaki et al., 2011). This approach outperformed the baseline CPU cost by an order of two magnitudes. In conclusion the paper showed that for several million pages, R-tree based methods do not appear to perform as good as a carefully implemented trivial baseline. This is also not only restricted to R(*)-trees, as all deep spatial structures spend significant time navigating the structure and performing lookups in the leaf nodes (Christoforaki et al., 2011).

Skovsgaard et al. (2014) explores techniques to support a top-k function for frequent terms in a spatio-temporal tagged text content. They argue that index structures based on R-trees are not well suited for rapid content stream and spatio-temporal aggregate queries. While not exactly within our scope, streaming spatio-temporal data, which includes microblogs and Twitter, are a huge contribution to the amount of geo-tagged data out there.

Tsatsanifos and Vlachou (2015) focused on including non-spatial information into spatio-textual queries. E.g. a hotel would be ranked higher if there are nearby restaurants and entertainment facilities. The paper then aims to not only retrieve objects that have high

spatio-textual similarity, but also include information about facilities in the nearby vicinity. The index proposed here is called ”SRT” and uses a bitmap to represent the occurance of terms in the global vocaulary. Each bitmap is then mapped to a hilbert curve. The order of the objects is defined in a way such that if the distance of the vectors are 1, the objects have only 1 different term. The index uses a R-tree built on both the spatial location, the hilbert value and a non-spatial score.

Lee et al. (2014) presents a lightweight index on top of HBase and Hadoop. The index is created by using a type of space filling curve called geohash to create bounding boxes from a latitude/longitude pair as a hash string. This string has unique properties that can be utilized to specify how much precision the records in the index should have, and can be used to define the size of the bounding boxes. The paper mentions pruning, but does not detail in any form how the pruning process happens.

In (Fox et al., 2013) the researchers have also facilitated the use of Geohash as a spatial indexing structure, but on top of Apache Accumolo[1], which is a distributed key-value store . The contribution of the paper is mostly focused around spatio-temporal queries, and constructs the rowID (key) to be a combination of a temporal value and a geohash character. The rest of the record is then the more specific spatial location and time. This approach only uses 1 character of the 7 calculated to distribute the records, as it was noted as an explosion of unique rowID ranges if the whole geohash value was used as key. However the most significant variable under the evaluation was the number of geohash characters in rowID (Fox et al., 2013). As one geohash character is able to cover 1/32 of the earth, by using two would cover 1/1024 which is much more specific. The study also lists many results in tuning Accumulo, which is not particularly useful.

To sum up there exists a lot of different approaches to spatio-textual indexing. We have in the previous chapter introduced a couple of different indexes and combination schemes, and we can see that there exists a lot of different approaches in the related work. Many usages of R-tree has been observed, but a lot of work has also been done on using space filling curves as a spatial index. Inverted index is however much more dominant when spatial and textual are combined, but there are exceptions to this as well.

[1] https://accumulo.apache.org/

Chapter 5

Current Spatial Systems

In this chapter multiple currently available approaches to solving spatial processing on Big Data platforms will be presented. Initially a overview of SpatialHadoop will be detailed, a further extension called GISQF will follow. A alternative variant which also extends Hadoop is called Hadoop-GIS and will be explained before rounding up with 2 implementations of Spatial processing on Apache Spark.

5.1 SpatialHadoop

SpatialHadoop is a MapReduce framework based on standard Hadoop, but developed to incorporate native support for spatial data.

5.1.1 Architecture

SpatialHadoop is an extention of the Hadoop framework, and injects spatial data awareness into each layer of the architecture. Specifically it extends Hadoop's first layer, the language layer, with a expressive high level language for performing spatial operations and describing spatial types. The storage layer is supplemented by adding spatial index structures like Grid file, R- and R+-trees. In the map-reduce layer, a SpatialFileSplitter and SpatialRecordReader is added to improve spatial data processing efficiency.

Challenges with index creation

Index structures are optimized for procedural programming, where a program executes statements sequentially. Hadoop utilizes functional programming, where map and reduce functions are executed by slave nodes. This means traditional solutions for constructing R-trees are not applicable (Eldawy and Mokbel, 2015).

Language Layer

In the language layer SpatialHadoop adds a high level language for expressing spatial operations and data types. This layer provides Pigeon, which is an extension of Pig Latin that adds spatial data types, functions and operations (Eldawy and Mokbel, 2015). Standard spatial data types are supported, such as Point and Polygon for example. Extending Pig Latin with more spatial awareness creates support for filter and join functions as well.

Storage Layer

SpatialHadopp uses a two layer index structure to overcome the challenges with traditional index creation. The top layer is a *global* index which indexes all of the partitions across the nodes of the cluster. A *local* index further indexes the records within each node. SpatialHadoop stores the spatial indexing structures within HDFS, and in combination with the global index file are able to retrieve the input files without having to scan all the data (Eldawy and Mokbel, 2015).

The index is created by completing three phases:
1) Partitioning: This phase partitions the input file into *n* partitions. To get the number of partitions the equation $n = \lceil \frac{S(1+\alpha)}{B} \rceil$ (Eldawy and Mokbel, 2015) is used where *S* is the input files, B is the HDFS block capacity (default = 64 MB) and α is an "overhead ratio" which accounts for the overhead when storing the local indexes.

Next a minimum bounding rectangle (MBR) is created for each partition to decide the spatial area of each partition. All of the MBRs makes up the entire space domain.

Finally each record *r* is assigned to a partition *p* that covers the record's geographical space, and the map function writes an intermediate pair $< p, r >$.
2) Local Indexing: The requested index structure is now created as a *local index* on each partition. Each local index has to fit in one HDFS block *B*, as this allows spatial operations to access each local index as one map task. It also ensures that the index is *load balanced* by Hadoop when it relocates the blocks across nodes.
3) Global Indexing: To build the global index, as the requested index structure, all local index files are concatenated into one file and a global index is built by using the bounding rectangles as the index key. This global index is kept in memory at the master node.

MapReduce Layer

Similar to standard Hadoop, the MapReduce layer is where the query processing runs MapReduce tasks. SpatialHadoop however supports spatially indexed input files, and enriches the Hadoop system by using a *SpatialFileSplitter*. This splitter exploites the global index to prune the file blocks so that only blocks that can contain the answer is retrieved. A *SpatialRecordReader* is used to read the resulting splits and uses the local index when processing the data.

Operations Layer

In addition to the previously mentioned layers, the Operation layer make up the core of SpatialHadoop, and with the resulting spatial functionality the system can perform operations such as range query, k-nearest neighbor and spatial join. **As an example**, a range query takes a set of spatial records *R* and a query area *A* and returns $r \subseteq R$ where *r* is located in *A*.

Query Work flow

1) Global Filter: By using the global index, SpatialHadoop checks whether the block *b* has its bounding rectangles inside *A*, $b.bound \subset A$, if this is true the whole block is returned as all records inside the block is inside the query area. If the block is not in A at all it is discarded, as no record could be located inside A. However if the bounds are partially overlapping A, the block is sent to further processing in step 2.

2) Local filter: The local filter uses the local index to extract the matching records. This means for example traversing the R-tree to extract the proper records. **With Replication:** With Grid-file and R+trees the global filter has to further process all blocks that are partially or completely inside *A*, as there may be duplicates records . The local filter has an extra step, which is called *duplicate avoidance* (Eldawy and Mokbel, 2015).

5.2 SpatialSpark

SpatialSpark is a implementation of spatial operations and indexing on top of Apache Spark. The technical contributions consists of two spatial-join algorithms, spatial indexing and range query (You et al., 2015). SpatialSpark is build with the Spark specific RDD(Resilient Distributed Dataset) in mind. The programming language used is Scala.

The index is created by sampling the dataset and creating minimum bounding rectangles based on the sample data. The remainder is then scanned and each record is placed in the best matching partition based on the partition's MBR and the record MBR. The index is saved as a textfile, and by querying the index with a geometry it is possible to prune away records that cannot match the query parameter. Scanning through the index, each partition that is partly covered, of fully covered by the query area is returned. This approach is very similar to SpatialHadoop, as there exists a "global" index, which can prune partitions before filtering the remainders.

Some problems related to efficiency when computing on multiple nodes have been documented (You et al., 2015), and a preliminary investigation showed that overhead when shuffling small jobs where high enough to affect the overall performance. Suggested further research on the trade off between number og partitions and size of partitions.

5.3 GeoSpark

GeoSpark is an in-memory cluster computing system for processing large spatial datasets. It is implemented on top of core Apache Spark to provide support for spatial data types, indexes and spatial operations (Yu et al., 2015). GeoSpark thereby extends the standard Resilient Data Set (RDD) to support spatial data, which is called Spatial Resilient Data Set (SRDD).

GeoSpark consists of 3 layers:

Apache Spark Layer which is the normal Apache Spark operations and data types.

Spatial Resilient Distributed Dataset (SRDD) Layer Which extends the normal RDD with spatial data types and operations to partition the SRDD across the cluster. To partition the data, GeoSpark utilizes a global grid file. The spatial space is split into equal geographical cells, and each element in the SRDD is assigned to the overlapping cell. If an element intersects with multiple cells, a duplicate is made for each. (Yu et al., 2015) Two types of indexes are supported in this layer, R-Tree and Quadtree.

Spatial Query Processing Layer provides support for spatial queries such as Spatial range query, spatial join query and spatial KNN query.

5.4 GISQF

(Al Naami et al., 2014) has developed a Geographic Information System Query Framework (GISQF) on top of SpatialHadoop. This system extends the normal operations of SpatialHadoop to include indexing, decoding and querying a specific dataset, the Global Data of Events, Langugage and Tone (GDELT [1]). GISQF consists of 3 layers, the bottom layer being responsible for preprocessing the datasets into shape objects so that SpatialHadoop can accept the input. The second layer communicates with SpatialHadoop to index the dataset, the index is saved in HDFS. The third layer is responsible for query processing.

When creating the index, GISQF creates a ”PreMaster” file that contains a record describing the MBR of the input dataset. The input is then divided into partitions based on the specified index, for example R-tree, Grid or R+-tree. A local index is then created on each partition containing meta data for some blocks and their MBRs. The local index is stored in HDFS. All local indexes are concatenated into one global index to be stored into the main memory. The process of creating the index is bottom-up, as the global index is created first when all the local indexes have been completed. Querying is top-down, as the system first query the global index and then the local indexes.

All in all Al Naami et al. (2014) extends SpatialHadoop to perform indexing on a specific dataset, and utilizes a lot of the same methods as SpatialHadoop but adapts the input to the system. The paper only presented comparisons to standard Hadoop, which it outperformed. The paper reinforces the need for MapReduce when dealing with big datasets, and utilizes the same methods as SpatialHadoop when constructing the index.

[1] http://www.gdeltproject.org/ [Accessed 4-May-2017]

5.5 Hadoop-GIS

A scalable spatial data warehousing system, designed to run large scale spatial queries by utilizing the MapReduce paradigm. Hadoop-GIS uses global partitioning and an on demand local index to support efficient queries. The system is also integrated into Hive to support declarative queries. The spatial objects are distributed into buckets called tiles, and each object are given a tile UID which is then stored on HDFS. Each tile is then spatially processed individually, boundary objects are handled. Aji et al. (2013) aims to provide a spatial query engine that can *(1)*support a variety of spatial queries, and that can be extended.*(2)* Parallelized on a cluster, and *(3)* can leverage existing indexing and query methods. This resulted in Real-time Spatial Query Engine(RESQUE). On spatial partitioning Aji et al. (2013) provided 2 main arguments on why it is very important, first of all, partitioning the 2 dimensional data creates a set of tiles which becomes the processing unit for the query tasks. The partitions are independent and can be run in parallel, therefore the partitioning scheme determines the computational parallelization.

Second, data skew can be quite substantial in spatial data (Aji et al., 2013), by experimenting with the numbers of partitions, the goal is to create partitions that contains similar number of records. The example used in the paper was OpenStreetMap, where by partitioning the dataset on 4Kx4K, resulting in partitions with 20k records when the average number was 4,291. Optimizing the partitioning scheme to normalize the number of records in the partitions would then improve performance. As a conclusion, Hadoop-GIS utilizes spatial partitioning and partition based parallel processing to create a solution that combines the scalable and cost-effective nature of MapReduce with efficient spatial query processing and access methods.

5.6 Summary

Most current systems reviewed focused on partitioning the data in an efficient way. Aji et al. (2013) showed that there are two main reasons for doing this, to increase parallel processing efficiency and to reduce record skew in the dataset. There where multiple ways of achieving this, Eldawy and Mokbel (2015) with SpatialHadoop created a global index based on local indexes as a way to prune away partitions. You et al. (2015) with SpatialSpark sampled the input dataset and created minimum bounding rectangles and fitted the remaining dataset into the best matching partition. GeoSpark (Yu et al., 2015) constructed a global grid file, and placed records into the corresponding cell.

As well as partitioning, in most of the systems there was a need for a local index. Variations of R-tree was most commonly used, as well as quad-tree or grid file.

SpatialHadoop seems to be the common denominator for the rest of the systems, as the structure of the indexes are based on the idea of having one global index keeping track of the MBR of the local indexes to enable pruning.

Chapter 6

Approach

6.1 Intro

This chapter will be a presentation of the approach for extending a spatial index with a textual property. An exploration of selecting a combination scheme and deciding adoptions that would need to be made implementing the index on top of Apache Spark will be presented. Finally the resulting system will be introduced.

6.2 Extending spatial indexes

The most intuitive method of extending the existing spatial indexing systems with a textual property would be to extend the spatial index to a spatial-first spatio-textual index as we have seen in Chapter 3 and 4.

For example SpatialHadoop's R-trees can be fitted with an inverted file at their leaf nodes which can describe the textual properties resulting in something similar to a R-IF index. Chen et al. (2013) showed that IF-R performed better than R-IF when querying, so another alternative could be to construct a inverted index first, and use R-trees for each posting in the inverted file.

We should consider a few criteria when selecting indexes to extend. The index needs to perform well, with quick run times when querying. In addition to this, the size of the index should be considered, as we are working with large datasets and the result in both cases should not grow more than linearly. The index build time should be considered as well, as we want the system to generate the index in a reasonable time.

Using space filling curves to index geographical data seems to be the most space conserving strategy. Both in Christoforaki et al. (2011) and Lee et al. (2014) the space filling curve strategy showed good results in space considerations and run time.

Fox et al. (2013) and Lee et al. (2014) have utilized geohashes to represent the spatial data, both on top of distributed platforms. This approach complements the MapReduce aspect of Spark, as we can freely compute the geohash for each record without having to

pay attention to a global structure like a R-tree.

When establishing Geohash as our spatial structure, an important aspect of how to partition the data arises as we do not have any global structure that can be used. We will discuss this in then next section.

After accepting Geohash as our primary index, the choice of textual index are in question. The state of the art in spatio-textual indexing are using an inverted index at the top level, and the index itself is the most common textual index in the field. By choosing to use an inverted index, we only need a binding from each word to the ID of the record.

Combining the spatial and textual indexes was first done by creating the baseline, and is utilizing the textual-first combination scheme. Later a spatial-first variation was used that included partitioning.

By using the spatial-first combination scheme, the index have a way of querying the spatial region, obtain the results and find the term we are querying for. The remaining process is then simply to retrieve the record from the dataset.

6.2.1 Partitioning

This system should be able to perform similar functions as SpatialHadoop, which uses a global index to allow pruning among the partitions. An index in each partition is used to further improve performance, but in our case we want to utilize a textual index at this level. When working with Apache Spark we want to utilize all of the system's most efficient functionality. By using the partitioning tools that exists within Spark, we can automate the process of pruning partitions when querying. By selecting the proper columns to partition the data on when creating the index, we are able to prune partitions that certainly does not contain the query area.

The partitioning function in Spark is based around selecting a column or multiple columns to partition the data. Records will be distributed into partitions based on values in the columns if we select some carefully chosen values. We use this in our evaluation to show test whether an index partitioned on the geohash performs better than the baseline of using a unpartitioned text-first index.

Spatial-first

As the partitioning function in Spark is column based, we need to select a proper partitioning scheme. Fox et al. (2013) has previously proved that using a small prefix of the full geohash have proved to increase performance when pruning. By placing the first character of the geohash in a separate column, we can first partition the data into 32 folders based on the first character. This enables us to immediately select only the partition that our geohash is located within.

It is important to note that while using more characters to partition the data would possibly improve query run time as we could prune increasingly more data, the partitioning function in Spark specifically indicates that the function works best when creating less than tens of thousands partitions. Using 3 geohash characters would create 32'768 partitions, and in addition to this, HDFS does not work well with many small files. By using 1 character we can balance the need to prune data with the restrictions our system provides.

After pruning the irrelevant partitions, a pass to filter the remaining geohashes are performed by comparing the prefixes to the geohash computed when querying.

Using geohash to represent spatial information provides a ”good enough” precision if we use an appropriate amount of characters in the hash. As default this precision value is set to 12, giving the geohash a worst case precision of 3.7cm x 1.9cm ($7cm^2$).

For each record that is encoded with a geohash, we should also utilize a textual index. A simple but effective index is the record-level inverted index, which maps each word to the corresponding document ID. To utilize this in a distributive manner we create a new record for each word, attach the ID to the word while keeping the geohash value. Performing filtering based on the geohash string first will prune the data, resulting in a dataset with all of the words that exists within our query area. From there finding the tweets that contain the right queryterm is done by filtering the remaining rows and remove words that is not our search term. This will result in a final dataset containing a set of IDs that can be efficiently retrieved by querying the ID'ed input dataset. The dataset is partitioned by ID in Spark, resulting in 32 partitions. A join to extract the IDs is then performed as we collect the correct tweets from the input dataset. The final result are printed to disk.

6.2.2 Variations

The baseline of the index was first created, Textual-First and later extended to a Spatial-First variation. Textual-First uses a similar structure as Spatial-first, but are not partitioned by the geohash. In addition to this Textual-First queries the index initially by filtering the dataset based on the query term, before filtering the geohashes based on the query area.

6.3 Overall Pseudo code

Algorithm 1 Overview of program flow for the spatio-textual index

1: $ReDistributed \leftarrow ProcessInput()$
2: $AssignId()$
3: **for** $Word$ in $Tweet.TextContent$ **do**
4: $\quad RemoveStopwords()$
5: $\quad ComputeGeoHash()$
6: $\quad$ Create $Row(Geohash, Word, ID)$
7: $RePartition()$

6.4 Spark Specifics

ReDistributed The first step is to read the input and redistribute the data as it may not be in a parallelizable form. This step reads the data, redistributes the input into a suitable number of partitions.

AssignId After the data has been parallelized, an ID is assigned to each record so we can refer to this ID in the inverted index. After every record is given an ID, the dataset is partitioned by the ID using a hash function to partition the data across 32 partitions.

$$ZippedDataset = Dataset[ID, Tweet]$$

RemoveStopwords A naive and simple IR method to remove the most common stop words. This includes removing symbols and urls.

ComputeGeoHash Computes a geohash from the latitude, logitude pair that is associated with the tweet.

Create Row For each remaining important term, a row of the term, geohash and the ID is created. The first character in each geohash is given its own column, as to partition the data in the RePartition stage.

$$GeohashedWords = Dataset[Geohash, Word, ID, Char - 1]$$

RePartition The partitioning function partitions the records based on $Char - 1$, which is the first character of the calculated geohash. This function also saves the index to disk.

6.5 Querying

For a query $q\{g, t\}$ where $q.g$ is a geohash of the query location and $q.t$ is the query term, the spatio-textual query execution is completed in three steps:

6.5.1 Spatial Query

When querying the spatial index, the first geohash character in the query geohash $q.g$ is retrieved. Spark then reads the stored index's metadata to retrieve only the partitions that have the correct geohash prefix.

After retrieving the partitions, a filter is performed to find the geohashes that either start with or is equal to $q.g$. In case of border cases which have been discussed previously, every neighbor to the geohash is also collected and the records starting with or are equal to the neighbors are also returned.

6.5.2 Textual Query

When querying about a textual term $q.t$, a filter is applied to the dataset finding the records that contain the term t. The records that contain the terms collected and the ID column is selected.

6.5.3 Retrieving the results

After collecting the IDs that fit the spatio-textual query, we apply a hash function to the IDs, to find the partitions in $ZippedDataset$ we need to collect. Finally the resulting IDs and the retrieved records from $ZippedDataset$ are joined, and the results are saved to disk.

6.6 Baseline version

The baseline version of the index also follows the same procedure for constructing the index, but does not include the last step *RePartition*. When querying, the textual query is performed first, filtering to retain only the query term before performing the spatial query. The spatial query only filters the remaining records by $q.g$. The retrieval of the results are equal.

6.7 Restrictions

Some restrictions apply when working with this system, the most applicable is listed below.

6.7.1 Geohash

Finding neighbour hashes At certain locations the hash value for surrounding areas does not share the same prefix. A solution used in the current system is to find the neighbor hashes and include them in the query. For example the hash value near the "O2", North Greenwich, England is 'u10hbp', but just across the river a couple of hundred meters away, the hash is 'gcpuzz'.

Precision When creating and querying, the precision could affect the result of the index and query time. As default the indexing precision is set to 12, giving the error rate of about 7 cm^2.

BRQ This approach is only suited for boolean range queries, as the results are not ranked in any way.

6.7.2 Partitioning

This partitioning scheme can be improved, but for a naive system like this it seems acceptable. Utilizing Spark's integrated partitioning function is likely to be efficient enough for our purpose. The downside of this however is that we have less coarse grained control of the actual process when partitioning. Finding alternatives, and improving the amount of partitions we can prune is likely to increase performance in the long run.

A better partitioning scheme could then potentially increase the performance of the index by grouping together geohashes that has the same prefix. The current method partitions the data based on the first geohash character, which means that when queried with a more precise geohash does not prune more partitions.

Another probable performance enhancement is a better IR solution to retrieving the final records when we have collected the IDs of the query.

Chapter 7

Evaluation

In this chapter the results from evaluation will be presented. First setup and information about the cluster will be detailed along with some technical details about the run parameters. Data for run times, index creation times and index size then follows accompanied by a short comment.

7.1 Dataset

We have a big dataset of Tweets that we have split into varying sizes.

Name	# Geo-Tagged Records
Dataset1	10.8M
Dataset2	16.8M
Dataset3	28.4M
Dataset4	50.2M
Dataset5	86.3M
Dataset6	142.1M
Dataset7	454.1M
Dataset8	521.3M
Dataset9	2703.2M
Dataset10	3619.8M

Table 7.1: Overview of datasets

The dataset contains 3619M geo-tagged tweets collected over a period of time. The information in each record is quite substantial, as it contains data about the tweeting user and place information. The dataset is collected from all around the world, and totals to about 1.5 TB in compressed gzip files.

The most important information in the tweets are of course the geographical data and the textual content. A lot of metadata about the user, media contained in the tweets, retweets etc. are not very interesting to our system. What we are extracting from the tweets are the spatial data, that are either specific geo tagged points, or a bounding box from an associated place where the tweet was posted. We prioritize the geo tagged point as it is more precise than the bounding box, but use a point in the bounding box if the geo-point is not specified. In addition to the spatial data, we also collect the textual description of the tweet. An example of the full tweet can be found in Appendix A.

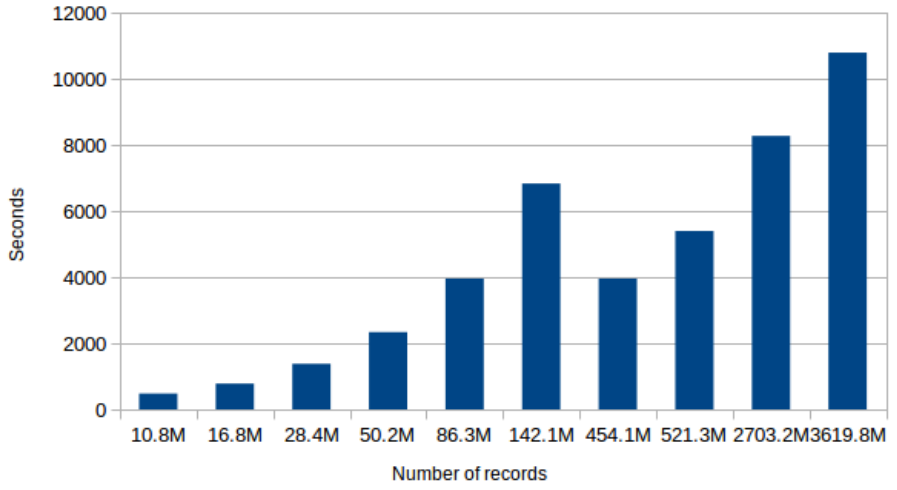
7.2 Setup

The full spatio-textual dataset consisted of 112 compressed files containing the tweets. The compression used in this case was gzip, which is not a parallelizable format, and as a result each file could only be loaded by one core at the time. This created some disproportional loading times that can be observed in Table 7.2. Note that Dataset1-6 each consisted of a single gzip file, thus could only utilize one executor and one core on that executor at a time. Dataset 7-10 consisted of multiple files which enables Spark to read each file simultaneously, however even the biggest dataset could only utilize 112 cores at a time. This means that the data load time is bound by the biggest gzip file in the directory, even if the other 111 are finished. The index itself is stored as a parallelizable format, parquet, which means we can fully utilize the cluster, and by extension all of the executors when loading the index to query.

Name	Data Load (in seconds)
Dataset1	480
Dataset2	780
Dataset3	1380
Dataset4	2340
Dataset5	3960
Dataset6	6840
Dataset7	3960
Dataset8	5400
Dataset9	8280
Dataset10	10800

Table 7.2: Data Load in Spark

Figure 7.1 Data Load Times



7.2.1 Cluster

The testing environment consists of 20 machines each running with 16 cores and 125GB memory. Each machine where running Ubuntu linux 14.04. Apache Spark had enabled dynamic allocation of resources, so the number of executors where dynamically being added as needed. The number of cores per executor, the memory available in each executor where specified at run time, as well as a driver.

Index creation:

Executor : 1 cores, 7G Memory

Driver : 2 cores, 4G Memory

Both : Extra JVM options ” -XX:MaxPermSize=256M ”

Index Query:

Executor : 3 cores, 3G Memory

Driver : 2 cores, 4G Memory

Both : Extra JVM options ” -XX:MaxPermSize=256M ”

7.2.2 Run time and Query parameters

Below is a summary of the run times for each dataset. For each dataset the index was created 6 times and the median value was chosen to represent the average performance. The performance evaluation is based on the Spatial-First variation, and an example comparison is presented afterwards.

Below is a listing of query parameters used in all queries. Figure 7.2 is a visual representation of the query area.

Location 40.754669,-73.986053 is the coordinate for Midtown in Manhattan, NY.

Precision A geohash precision of 5 characters where used. Geohashes from the immediate neighbors is also included. This means the bounds for the query includes the entire Manhattan area and more, see Figure 7.2 for an visual representation.

Query Term The Query term used in the search is ”money”.

7.3 Results

We will split the results from index creation in two parts, Dataset 1-7 in group 1 (Figure 7.3) and 8-10 in group 2 (Figure 7.4). This is done as the numbers increase substantially, making it easier to read the values from the graphs. The results from querying the index remains within a reasonable range and is presented in a single graph.

7.3.1 Index creation

As we can see from the figures 7.3 and 7.4, index creation is fairly consistent at the smallest datasets. This is probably due to overhead when working with the cluster, as it does not have enough data to utilize the resources efficiently. The dynamic allocation enabled in Spark can also affect the performance at this level, as it takes some time in each task before more executors are added. Further observations show that the run time increases at least linearly once the dataset becomes big enough.

Figure 7.2 Query Area - Collected from http://geohash.gofreerange.com/

Figure 7.3 Index Creation Dataset 1-7

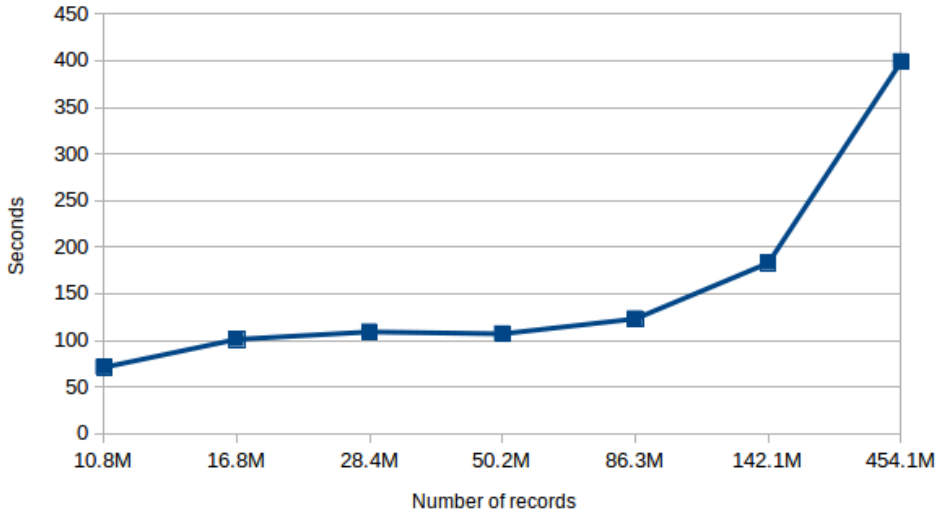
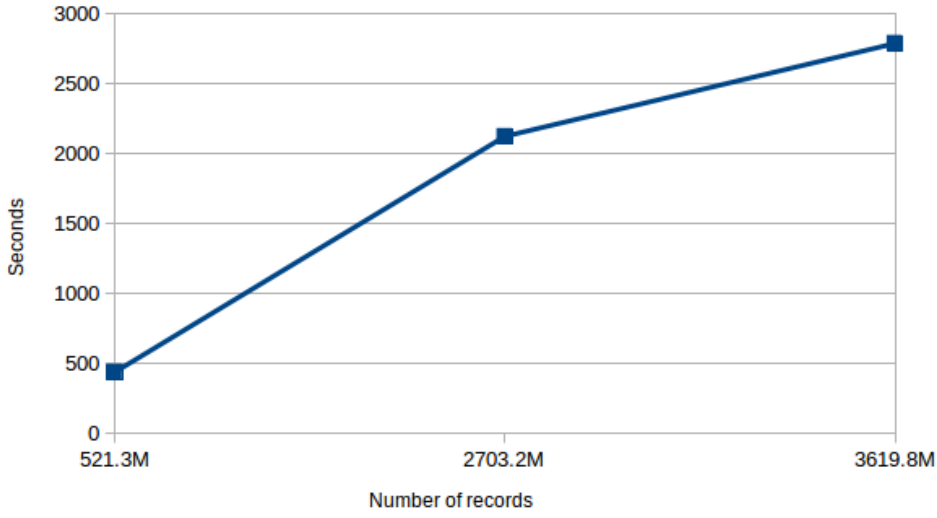


Figure 7.4 Index Creation 2 Dataset 8-10

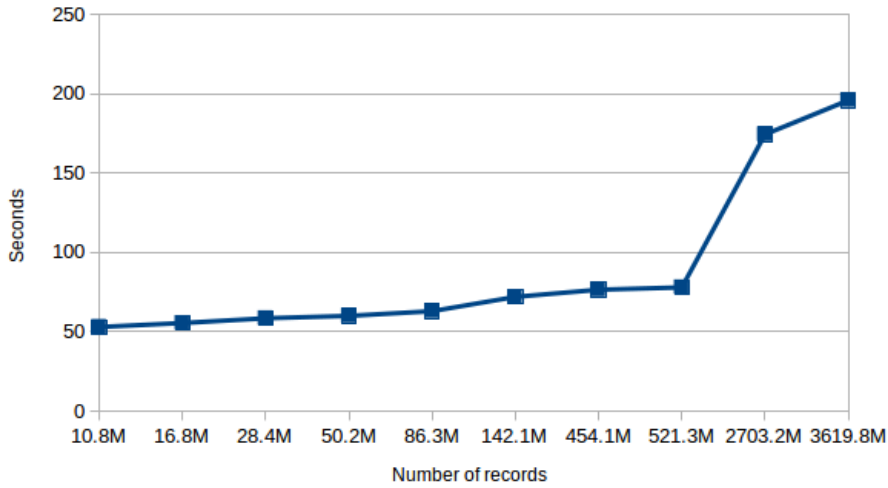


7.3.2 Index Query

Index query remains quite stable until the dataset become big enough, possibly due to spark overhead when running the query on smaller datasets. As the figure 7.5 shows, when the dataset exceeds 500M records the run time also increases.

We can observe a steep increase in run time when querying the larger datasets. Note that the 9th dataset contains over 5x as many records as dataset 8. The total time for querying even the largest dataset does not take more than 200 seconds.

Figure 7.5 Index Query



7.3.3 Size of index

The amount of space the index uses is defined as $zippedDataset + GeohashedDataset$. The size of the zippedDataset would be very similar to the input dataset as we only add an ID to the records. The most interesting value to examine in this case is the size of $GeohashedDataset$.

The indexes are compressed to the file type ”parquet”(Apache, 2017), and is designed to retain the partitioning of index. The compression codec used is ”snappy”.

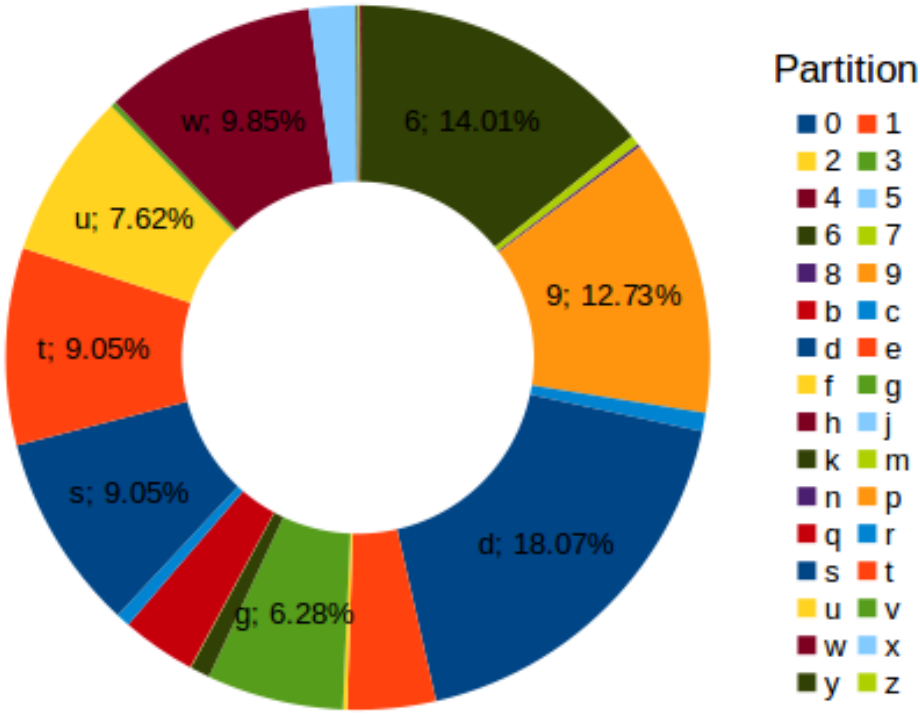
The following Table 7.3 is a listing of the index sizes for each dataset. The two sizes describe the total size including HDFS replication (3x replication) and when unreplicated.

Name	Size replicated	Size unreplicated
Dataset1	1.5909 GB	543.01 MB
Dataset2	2.4730 GB	844.13 MB
Dataset3	4.1326 GB	1.3775 GB
Dataset4	7.3040 GB	2.4347 GB
Dataset5	12.956 GB	4.3188 GB
Dataset6	21.341 GB	7.1137 GB
Dataset7	66.714 GB	22.238 GB
Dataset8	76.822 GB	25.607 GB
Dataset9	421.44 GB	140.48 GB
Dataset10	529.67 GB	176.56 GB

Table 7.3: Index size of Spatial-First

The distribution of records in partitions is represented with the Figure 7.6.The data is collected from the spatial-first index created on Dataset10 showing values for the biggest partitions. A more detailed table containing all the results can be found in Appendix B

Figure 7.6 Distribution of records on partitions



7.4 Comparison to the baseline

A comparison to the baseline was performed on the biggest dataset, using the same query parameters as used in the previous section.

Creation Index construction took 2426 seconds on average. Which is much better than the spatial-first version. The time saved from not having to perform the partitioning makes up the big difference.

Query Query time are very similar, the baseline querying on 270 seconds, and spatial-first with 196.

Size The size of the Baseline index was calculated to 214GB before replication, versus 176GB for the Spatial-First variation .

Another comparison was made where the query geohash precision was set at 8 characters. This meant that the result only contained less than 5 records. When operating with few result the query times improved more using the spatial-first variant. This indicates that the retrieval of the documents in the last stage might be a bottleneck. This assumption can be made as no changes occurs in the spatial and textual query stage, where the amount of work remains constant. But the final retrieval stage accesses fewer partitions before returning the result.

Baseline: Query with 8 geohash characters: 146 seconds

Spatial-First Query with 8 geohash characters: 90 seconds

7.5 Summary

As we can observe from the various graphs and tables, the index are performing well both when creating the index as well as when querying. The run times seems to be growing linearly which is to be expected, and the sizes of the indexes are calculating to about 10% of the total input dataset. The query times does not vary drastically between the baseline and the spatial-first variant, however the spatial-first variant does maintain a performance boost over the baseline. The baseline does however perform on a acceptable level regardless. When working with Spark, parquet retains a lot of metadata and information about the dataset, this could be affecting the performance in a positive way. The more precise query yielded some hints that the retrieval process might be a possible bottleneck.

Chapter 8

Summary and Conclusion

In this chapter the evaluation results will be summarized, and the research questions will be accounted for. As a final comment some further research are suggested.

8.1 RQ 1: Spatial indexing on Big data platforms

In this paper several different spatial indexing systems have been explored, both systems built on top of distributed databases such as HBase and Accumulo, Big data platforms such as Hadoop, and systems built on top of Apache Spark. Several similar trends can be observed throughout the survey, where the most predominant one is the usage of a global method of pruning partitions. The usage of a global and local index can be found in systems based on SpatialHadoop, but other approaches such as using characters from geohashes are also represented. Many of the current systems today utilizes some form of spatial structure such as R-trees or Grid file, however arguments have been made in both in favor and apposing this approach. Using an inverted index for storing the textual properties are most widely used.

8.2 RQ 2: Extending spatial indexes

There are quite a number of different spatio-textual indexes in the field, however finding the right combination proved to be a challenging task. Going away from the more common usage of R-trees, we have focused more on utilizing coarse grained spatial structures such as space filling curves. To extend a SFC with a textual index, a simple inverted file has shown good result past experiments (Christoforaki et al., 2011). In the created system we utilized Geohash to represent each record's location, and this value was also used to partition the index. By utilizing this approach, which had previously been used by Fox et al. (2013) and Lee et al. (2014), we are able to prune the index area down to 1/32 of the earth when querying our index. This method achieves a similar pruning effect as SpatialHadoop, however to improve the pruning function we are restricted by limitations

in Spark and HDFS. After establishing a spatial index by utilizing Geohashes, we extend the spatial index by adding an Inverted index for each geohashed tweet. After pruning the partitions based on location, we can filter the records to obtain only the records that both are within the spatial query area, and contains the query term. We have in this paper used both the text-first combination in the baseline of the index. The second variant used a spatia-first combination scheme and improved partitioning.

8.3 RQ 3: Spatio-textual indexing on Spark

A series of experiments have shown an initial spatio-textual index built on top of Apache Spark. Combining a space-filling curve and an inverted index shows good initial results during the evaluation. Both the Geohash and inverted index are well documented index structures, and as the evaluation suggest, supports a large dataset both in index creation and at query time.

8.3.1 Run time

The creation of the index remains within a reasonable amount of time, even when the input dataset was large. Query times where also showing to grow linearly, as Spark was able to prune the partitions not containing the proper records. The limiting factor in both cases are the number of characters to partition the index on. If we use 2 characters in stead of 1 when creating the index, the query algorithm is able to prune more partitions and decreases the spatial area from 1/32 to 1/1024. This is not possible to achieve in the current system as the partitioning function is documented to only handle a partition number less than tens of thousands, in addition if two characters where used, HDFS would also be a bottleneck, as too many small files would be created. The spatial-first variant did show to outperform the baseline in querying the dataset, both when returning a large amount of results and when only returning a few.

8.3.2 Size of index

The size of the index also seems to be acceptable. Constructing to about 10% of the input dataset shows that the index is indeed not growing out of control. The baseline did however produce a bigger index than the spatial-first variation.

8.4 Conclusion

We have in this paper given a overview of different approaches to spatial indexing on Big Data platforms. A couple of different methods to extend such a spatial index to spatio-textual have been introduced. By using the information gathered from the related work, a spatio-textual index has been created on top of Apaches Spark. Evaluation based on this index is presented, and provides good results.

8.5 Further work

Suggested further work consists of exploring a better partitioning scheme when distributing the index, as we can only limit the query area to 1/32 of the earth's surface at query time. Finding a solution to the partitioning scheme so that it could utilize all of the geohash characters without causing problems for HDFS and Spark could improve query performance.

Exploring if there are any possibility to extend the index to support BkQ or TkQ is also suggested.

The baseline version of the index does not partition the index, as the number of unique terms globally are more than tens of thousands. The index performs rather well however, and exploring possible ways of improving the last retrieval stage would be an important part of improving the spatial-first index in comparison to the baseline. Another performance enhancing operation to explore further is a way of partitioning the index based on the terms.

Bibliography

Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., and Saltz, J. (2013). Hadoop GIS: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020.

Al Naami, K. M., Seker, S., and Khan, L. (2014). GISQF: An efficient spatial query processing system. In *2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, pages 681–688.

Apache, S. F. (2017). Apache parquet documentation. `https://parquet.apache.org/documentation/latest/`. [Accessed 18-May-2017].

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. (2015). Spark SQL: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394.

Balkić, Z., Šoštarić, D., and Horvat, G. (2012). GeoHash and UUID identifier for multi-agent systems. In *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, pages 290–298.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. (1990). The R*-tree: an efficient and robust access method for points and rectangles. In *ACM SIGMOD Record*, volume 19, pages 322–331.

Cary, A., Wolfson, O., and Rishe, N. (2010). Efficient and scalable method for processing top-k spatial boolean queries. In *International Conference on Scientific and Statistical Database Management*, pages 87–95.

Chen, L., Cong, G., Jensen, C. S., and Wu, D. (2013). Spatial keyword query processing: An experimental evaluation. *Proceedings of the VLDB Endowment*, 6(3):217–228.

Chen, Y.-Y., Suel, T., and Markowetz, A. (2006). Efficient query processing in geographic web search engines. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 277–288.

Christoforaki, M., He, J., Dimopoulos, C., Markowetz, A., and Suel, T. (2011). Text vs. space: efficient geo-search query processing. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 423–432.

Cong, G., Jensen, C. S., and Wu, D. (2009). Efficient retrieval of the top-k most relevant spatial web objects. *Proceedings of the VLDB Endowment*, 2(1):337–348.

De Felipe, I., Hristidis, V., and Rishe, N. (2008). Keyword search on spatial databases. In *ICDE 2008. IEEE 24th International Conference on Data Engineering*, pages 656–665.

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

Duggan, M. and Brenner, J. (2013). *The demographics of social media users, 2012*, volume 14. Pew Research Center's Internet & American Life Project Washington, DC.

Eldawy, A. and Mokbel, M. F. (2015). Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1352–1363.

Fox, A., Eichelberger, C., Hughes, J., and Lyon, S. (2013). Spatio-temporal indexing in non-relational distributed databases. In *2013 IEEE, International Conference on Big Data*, pages 291–299.

Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43.

Hariharan, R., Hore, B., Li, C., and Mehrotra, S. (2007). Processing spatial-keyword (sk) queries in geographic information retrieval (gir) systems. In *SSBDM'07. 19th International Conference on Scientific and Statistical Database Management*, pages 16–16.

Khodaei, A., Shahabi, C., and Li, C. (2010). Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *Database and Expert Systems Applications*, pages 450–466.

Lee, K., Ganti, R. K., Srivatsa, M., and Liu, L. (2014). Efficient spatial query processing for big data. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 469–472.

Li, Z., Lee, K. C., Zheng, B., Lee, W.-C., Lee, D., and Wang, X. (2011). IR-tree: An efficient index for geographic document search. *IEEE Transactions on Knowledge and Data Engineering*, 23(4):585–599.

Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). MLlib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7.

Oates, B. J. (2005). *Researching information systems and computing*. Sage.

Rocha-Junior, J. B., Gkorgkas, O., Jonassen, S., and Nørvåg, K. (2011). Efficient processing of top-k spatial keyword queries. In *International Symposium on Spatial and Temporal Databases*, pages 205–222.

Skovsgaard, A., Sidlauskas, D., and Jensen, C. S. (2014). Scalable top-k spatio-temporal term querying. In *2014, IEEE 30th International Conference on Data Engineering (ICDE)*, pages 148–159.

Tan, H., Luo, W., and Ni, L. M. (2012). Clost: a hadoop-based storage system for big spatio-temporal data analytics. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2139–2143.

Tsatsanifos, G. and Vlachou, A. (2015). On Processing Top-k Spatio-Textual Preference Queries. In *EDBT*, pages 433–444.

Vaid, S., Jones, C. B., Joho, H., and Sanderson, M. (2005). Spatio-textual indexing for geographical search on the web. In *International Symposium on Spatial and Temporal Databases*, pages 218–235.

Wu, D., Yiu, M. L., Cong, G., and Jensen, C. S. (2012). Joint top-k spatial keyword query processing. *IEEE Transactions on Knowledge and Data Engineering*, 24(10):1889–1903.

Xin, R. S., Crankshaw, D., Dave, A., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2014). GraphX: Unifying data-parallel and graph-parallel analytics. *arXiv preprint arXiv:1402.2394*.

Xin, R. S., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., and Stoica, I. (2013). Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, pages 13–24.

Yan, H., Ding, S., and Suel, T. (2009). Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on World wide web*, pages 401–410.

Yang, H.-c., Dasdan, A., Hsiao, R.-L., and Parker, D. S. (2007). Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040.

You, S., Zhang, J., and Gruenwald, L. (2015). Spatial join query processing in cloud: Analyzing design choices and performance comparisons. In *2015, 44th International Conference on Parallel Processing Workshops (ICPPW)*, pages 90–97.

Yu, J., Wu, J., and Sarwat, M. (2015). Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 70.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: cluster computing with working sets. *HotCloud*, 10:10–10.

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438.

Zhou, Y., Xie, X., Wang, C., Gong, Y., and Ma, W.-Y. (2005). Hybrid index structures for location-based web search. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 155–162.

Appendix

Appendix A


```
{
"in_reply_to_status_id_str":null,
"in_reply_to_status_id":null,
"created_at":"Sat Oct 03 15:42:18 +0000 2015","
   in_reply_to_user_id_str":null,
"source":"<a href=\"http://twitter.com/download/iphone\"
   rel=\"nofollow\">Twitter for iPhone<\/a>",
"retweet_count":0,
"retweeted":false,
"geo":null,
"filter_level":"low",
"in_reply_to_screen_name":null,
"is_quote_status":false,
"id_str":"650335072202190848",
"in_reply_to_user_id":null,
"favorite_count":0,
"id":650335072202190848,
"text":"yes  there  is  i might  be renting a room lol",
"place":{
        "country_code":"US", 15
        "country":"United States",
        "full_name":"Middleton, WI",
        "bounding_box":{
                "coordinates":[[[-89.600763,43.081683],[-89
                   .600763,43.133416],[-89.452015,43.133416
                   ],[-89.452015,43.081683]]],
                "type":"Polygon"},
        "place_type":"city",
        "name":"Middleton",
        "attributes":{},
        "id":"004e2b299a1fb473",
        "url":"https://api.twitter.com/1.1/geo/id/004e2b299
           a1fb473.json"},
"lang":"en",
```

```
"favorited":false,
"coordinates":null,
"truncated":false,
"timestamp_ms":"1443886938511",
"entities":{
        "urls":[],
        "hashtags":[],
        "user_mentions":[],
        "symbols":[]},
"contributors":null,
"user":{
        "utc_offset":null,
        "friends_c      ount":284,
        "profile_image_url_https":"https://pbs.twimg.com/
           profile_images/3483706310/5e9c7801c67bdc61e472f0
           5815d22b01_normal.jpeg",
        "listed_count":0,
        "profile_background_image_url":"http://abs.twimg.
           com/images/themes/theme1/bg.png",
        "default_profile_image":false,
        "favourites_count":15,
        "description":"Ask and i will tell you",
        "created_at":"Sat Apr 06 08:17:55 +0000 2013",
        "is_translator":false,
        "profile_background_image_url_https":"https://abs.
           twimg.com/images/themes/theme1/bg.png",
        "protected":false,
        "screen_name":"_philtaz",
        "id_str":"1331025042",
        "profile_link_color":"0084B4",
        "id":1331025042,
        "geo_enabled":true,
        "profile_background_color":"C0DEED",
        "lang":"en",
        "profile_sidebar_border_color":"C0DEED",
        "profile_text_color":"333333",
        "verified":false,
        "profile_image_url":"http://pbs.twimg.com/
           profile_images/3483706310/5e9c7801c67bdc61e472f0
           5815d22b01_normal.jpeg",
        "time_zone":null,
        "url":"https://pvallejos.avonrepresentative.com/",
        "contributors_enabled":false,
        "profile_background_tile":false,
        "statuses_count":364,
```

```
69          "follow_request_sent":null,
70          "followers_count":51,
71          "profile_use_background_image":true,
72          "default_profile":true,
73          "following":null,
74          "name":"phillip  vallejos",
75          "location":null,
76          "profile_sidebar_fill_color":"DDEEF6",
77          "notifications":null }
78  }
```

Appendix B

Partition	Size (MB)	# Files
0	5.314	1072
1	124.88	1112
2	29.649	1422
3	339.85	1114
4	515.6	1126
5	4.4288	1001
6	75879	1858
7	2434.5	1797
8	760.79	1548
9	68962	1703
b	8.3595	1114
c	4580.3	1219
d	97900	1951
e	21905	1880
f	1085.3	1750
g	33992	1711
h	0.0089209	4
j	1.0057	297
k	4914.5	1285
m	187.46	383
n	0.0022939	1
p	2.1791	126
q	18467	1618
r	3429.2	703
s	49009	2133
t	49009	2133
u	41290	1693
v	1372.3	1051
w	53356	2172
x	11628	769
y	418.06	862
z	64.6	486