11-8-2011

Scaling Geospatial Searches in Large Spatial Databases

Ariel Cary
Florida International University, acary001@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

SCALING GEOSPATIAL SEARCHES IN LARGE SPATIAL DATABASES

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Ariel Cary

2011

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Ariel Cary, and entitled Scaling geospatial searches in large spatial databases, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Vagelis Hristidis

Raju Rangaswami

Malek Adjouadi

Naphtali Rishe, Major Professor

Date of Defense: November 8, 2011

The dissertation of Ariel Cary is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2011

DEDICATION

To my parents and grandfather.

ACKNOWLEDGMENTS

I was privileged to be advised by quite a few professors at Florida International University (FIU) during the course of my doctoral program. I am thankful to my major advisor for his encouragement on my work. Dr. Vagelis Hristidis' experienced advise helped outline my research in many ways. I am immensely thankful to Dr. Raju Rangaswami for the insightful discussions we had, for making himself available at all times when I required his thoughtful opinions, and for turning my overall research experience an excellent and memorable process. Dr. Rangaswami is an exceptional person with high ethical standards that has certainly influenced my research style and critical thinking. I am also deeply thankful to Dr. Tao Li for his generosity in sharing his research experience and valuable feedback on my research ideas. Dr. Peter Clarke and Dr. Masoud Sadjadi were very kind in providing academic support and being always willing to discuss about research topics.

Without the support of my family and friends, this work could not have been completed at all. My wholehearted thank you to my parents and family for their emotional support that motivated me to hang in there. A very special appreciation to Gonzalo Argote and Rene Bueno, long time friends like no other, who have been with me in the ups and downs. I am also grateful to my lab mates for exchanging thoughts and the laughter that made the doctoral experience very lively.

Finally, I want to acknowledge the financial support that I received from the School of Computing and Information Sciences at FIU, The National Science Foundation (NSF), and a FIU Dissertation Year Fellowship awarded in the last year of my doctoral program.

ABSTRACT OF THE DISSERTATION

SCALING GEOSPATIAL SEARCHES IN LARGE SPATIAL DATABASES

by

Ariel Cary

Florida International University, 2011

Miami, Florida

Professor Naphtali Rishe, Major Professor

Modern geographical databases, which are at the core of geographic information systems (GIS), store a rich set of aspatial attributes in addition to geographic data. Typically, aspatial information comes in textual and numeric format. Retrieving information constrained on spatial and aspatial data from geodatabases provides GIS users the ability to perform more interesting spatial analyses, and for applications to support composite location-aware searches; for example, in a real estate database: "Find the nearest homes for sale to my current location that have backyard and whose prices are between $50,000 and $80,000". Efficient processing of such queries require combined indexing strategies of multiple types of data. Existing spatial query engines commonly apply a two-filter approach (spatial filter followed by non-spatial filter, or viceversa), which can incur large performance overheads. On the other hand, more recently, the amount of geolocation data has grown rapidly in databases due in part to advances in geolocation technologies (e.g., GPS-enabled smartphones) that allow users to associate location data to objects or events. The latter poses potential data ingestion challenges of large data volumes for practical GIS databases.

In this dissertation, we first show how indexing spatial data with R-trees (a typical data pre-processing task) can be scaled in MapReduce – a widely-adopted parallel programming model for data intensive problems. The evaluation of our

algorithms in a Hadoop cluster showed close to linear scalability in building R-tree indexes. Subsequently, we develop efficient algorithms for processing spatial queries with aspatial conditions. Novel techniques for simultaneously indexing spatial with textual and numeric data are developed to that end. Experimental evaluations with real-world, large spatial datasets measured query response times within the sub-second range for most cases, and up to a few seconds for a small number of cases, which is reasonable for interactive applications. Overall, the previous results show that the MapReduce parallel model is suitable for indexing tasks in spatial databases, and the adequate combination of spatial and aspatial attribute indexes can attain acceptable response times for interactive spatial queries with constraints on aspatial data.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

Geographic information systems (GIS) integrate and store complex and large volumes of geographically related data. The types of data managed by GIS include: raster data (satellite and aerial digital images) and vector data (points, lines, polygons). Spatial data is periodically generated via specialized sensors, satellites or aircraft-mounted cameras sampling geographical regions into digital images, or GPS devices generating geographical coordinates of real-world objects. Objects in current geospatial databases contain a rich set of attribute data, typically as textual descriptions and numeric values, stored in aspatial attributes, in addition to geographical information. GIS databases need to be optimized for various operations, such as data retrieval, spatial analysis, image processing, and visualization of geographic images. In particular, structured and semi-structured geographic data needs to be first loaded into GIS database schemas before users may start posting queries for performing their analysis. Due to the constant growth of modern spatial databases and the complexity of pre-processing loading stages, timely data ingestion may become a challenge for traditional sequential computing models.

In the unstructured data domain, today's Internet applications typically offer users the ability to associate geographical information to Web content, a process known as "geotagging". For example, Wikipedia has standardized geotagging of their encyclopedia articles and images via templates [oGC10]. Furthermore, technological advances in digital cameras and mobile phones allow users to acquire and associate geospatial coordinates, via built-in GPS devices or Wi-Fi triangulation, to media resources. Additionally, Web content can be automatically paired with geographical coordinates exploiting content features, such as place names or street addresses, in combination with gazetteers. Thus, the powerful combination of In-

ternet applications, GPS-enabled devices, and automatic geotagging can potentially generate large amounts of georeferenced content.

Search is one of the fundamental operations in GIS applications. Traditional spatial queries in GIS databases include spatial relations, such as nearby, intersects, contained-in, and distance threshold. For example, in a real estate database, users may be interested in finding houses for sale nearby a reference point. Efficient methods exist for processing spatial searches. Existing methods typically organize objects in tree-based indexes, such as R-tress, Quadtrees, or Grid-based indexes, which allow reducing substantially the search space to only objects in the vicinity of the query location. Then, a search method needs only to scan a few percent of the database to answer a spatial query.

In this dissertation we explore two related problem in spatial databases. First, we study how MapReduce [DG08] – a widely-adopted parallel computing model developed at Google – can be leveraged in spatial data indexing tasks. Second, we study composite indexing strategies to efficiently process spatial queries constrained on aspatial attributes.

CHAPTER 2

RESEARCH PROBLEM

2.1 Research Statement

In this thesis, we study and propose solutions to two related research problems in geospatial databases:

1. Parallel construction of R-trees on large spatial databases.
2. Efficient processing of nearest neighbor queries constrained on aspatial attributes.

2.2 Motivation

Large-scale databases in general are becoming the norm in today's information era. Spatial datasets with hundreds of millions of objects are fairly common nowadays. Nearly every object or event recorded by modern mobile devices, e.g. smartphones equiped with a GPS, can be geolocated, or specialized companies collect geographical data on ever increasing number of real-world objects. For example, an Internet-based yellow pages database reports to have more than 27 million busines listings[1] while a database of United States parcels has more than 110 million records [Sol10]. Thus, spatial information is being rapidly disseminated.

The MapReduce parallel programming model [DG08] is an emerging parallel computing model that facilitates writing distributed applications on large computer clusters. It has been widely adopted by the industry, and it has also found utility in scientific environments for solving data-intensive problems. Thus, MapReduce

[1] http://www.yellowpagesgoesgreen.org/

provides a potential alternative for scaling data processing tasks in spatial databases [CSHR09] [BCR11].

On the other hand, records in geospatial databases generally represent real-world objects, such as a house in a particular neighborhood, or events, such as a social gathering in a restaurant. Thus, records have a rich set of aspatial attributes associated to them in addition to their geolocation information. Typically, aspatial attributes are stored in textual and numeric format. In a real estate database scenario, house objects might have description, street address, number of bedrooms, and price attributes associated to them, in addition to the location of the houses. In this scenario, more interesting analyses can be performed by constraining spatial record retrievals on aspatial attributes. For instance, the following query may be posted to a real estate database:

"Find the *nearest* homes for sale to my *current location*
that have *swimming pool* and whose *prices*
are between $50,000 and $80,000"

Supporting efficiently spatial queries with aspatial constraints is challenging for existing query processing methods that use spatial-only indexes or attribute-only indexes. Thus, a suitable combination of both types of access methods, on spatial and aspatial data, and query execution plans are required.

2.3 Geospatial Data Model

In general, the geolocation of real-world objects can be approximated by geometrical figures, such as lines for road segments, or polygons for land parcels. Without loss of generality, the location of objects can be represented by a two-dimensional point, for example the middle point of a line, or the center of mass of a polygon.

On the other hand, the data model includes two sets (possibly empty) of textual and numerical attributes. Formally, the data model we adopt in this work is defined as follows. A spatial database is a set of objects $D = \{o_1, o_2, ..., o_N\}$ such that each $o \in D$ has a the following attributes:

- o_p – A two-dimensional point that represents the location of object o
- $\{o_{T_1}, o_{T_2}, \ldots\}$ – A set of attributes of *string* type.
- $\{o_{V_1}, o_{V_2}, \ldots\}$ – A set of attributes of *number* type.

2.4 Assumptions

In this work, we make the following assumptions:

- We assume that objects' geolocations in spatial databases are represented by GPS latitude and longitude coordinates as a two-dimensional point. The methods we describe in this work are not only limited to 2D points, but they can also be extended to support lines and polygons.
- In information retrieval, results are typically sorted by a ranking function that puts the most relevant document to the query in the first position of the query result, the second most relevant document in second position, and so on. In this work, we assume spatial distance as the ranking function. That is, the spatially closest object to the query location is placed first in the result list, and similarly the remaining objects are sorted by distance to the query location in non-decreasing order.

CHAPTER 3

SCALING SPATIAL DATA INDEXING

3.1 Introduction

This chapter presents our experiences from using the MapReduce parallel model to tackle the construction of R-trees [CSHR09] [CYAR10]. R-trees – tree data structures for organizing multi-dimensional data [Gut84] – are widely used in Geographic Information System (GIS) databases to accomplish fast data retrieval for queries with spatial relations; for example, "retrieve all objects that fall within the boundaries of a city". MapReduce is a parallel programming model developed by Google to simplify the writing of parallel applications [DG08]. The model was inspired by the map and reduce functions commonly used in functional programming.

In this chapter, we show how the ordering of multi-dimensional data via space-filling curves can be used to create a MapReduce algorithm for the problem of building R-trees in parallel. The proposed MapReduce algorithms were implemented in MapReduce Hadoop [Pro11] – an open source implementation of the Google's MapReduce model. Hadoop applications are submitted to a Hadoop cluster as jobs, where they are scheduled and executed. In our experiments we used a Google&IBM Hadoop cluster supplied to us via the NSF Cluster Exploratory (CluE) program [Pro08] [Ini07]. We experimentally evaluated our algorithms in terms of execution time, scalability and quality of the constructed R-tree.

The rest of the chapter is organized as follows. Section 3.2 gives an overview of the R-tree index data structure and MapReduce parallel programming model. It also presents physical configuration details of the Google&IBM's Hadoop MapReduce cluster used in our experiments. Sections 3.3 presents our MapReduce approach for building R-trees. Section 3.4 presents experimental results of running

implementations of our MapReduce algorithms under different settings. Section 3.5 discusses how our solution is different from previous approaches on parallelizing the construction of an R-tree. Finally, in Section 3.6 we summarize the chapter.

3.2 Preliminaries

3.2.1 R-tree Index Overview

R-tree indexes [Gut84] are a generalization of B-tree indexes for multi-dimensional data [Com79]. Without loss of generality, let us consider a dataset of two-dimensional points. An R-tree groups nearby points within a rectangle of minimal size, called minimum bounding rectangle (MBR). Subsequently, MBRs are recursively grouped in larger rectangles of minimal size. MBRs are oganized in a tree structure in such a way that every node stores MBRs larger than its child nodes. The root node contains entries storing the largest MBRs defined on the dataset. Nodes have minimum m and maximum M capacities. Node entries store two pieces of data: a pointer to either a spatial object or a lower level node, and the MBR of the pointed element. Leaf nodes contain pointers to database objects while inner nodes point to child nodes. Figure 3.1 shows a sample spatial database $D = \{o_1, o_2, ..., o_{12}\}$ (a), and an R-tree index with $m = 2$ and $M = 3$ constructed on D (b). Objects o_1 and o_2 are grouped by the rectangle N_1, which is further enclosed by its parent rectangle N_6.

R-trees provide efficient algorithms to retrieve objects contained within a spatial region [Gut84]. Intuitively, MBRs of the R-tree that do not intersect with the search region do not need to be explored. There also exist algorithms to efficiently retrieve nearest neighbor objects [HS99]. In addition, there has been a number of research works that aimed at improving the retrieval efficiency of R-trees by minimizing the

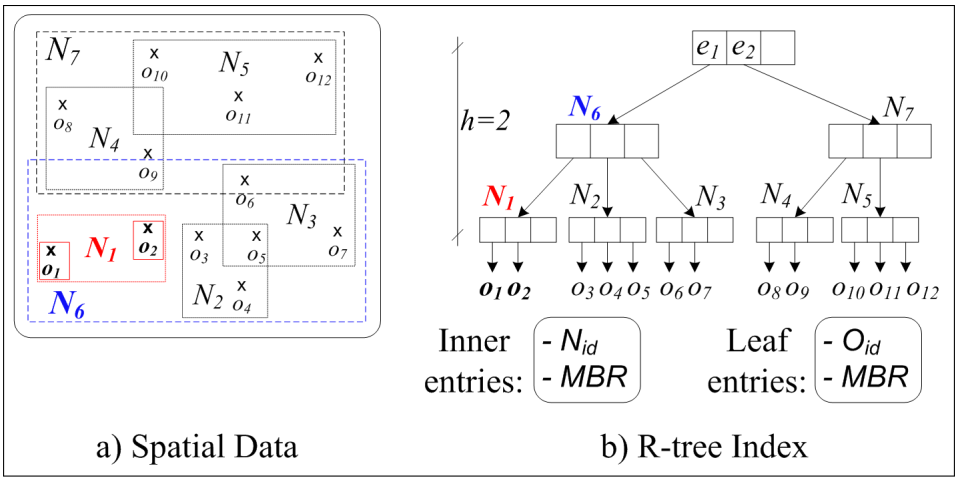


Figure 3.1: R-tree index example on a spatial database.

overall MBR overlap of the tree [BKSS90] [SRF87] [KF94], or guaranteeing optimal worst-case retrieval cost [AdBHY04].

3.2.2 MapReduce Overview

MapReduce is a simplified model for expressing distributed computations on large scale datasets [DG08]. Developed inside Google, the original motivation was the need to separate the complex details of writing distributed applications, such as fault tolerance, data distribution and load balancing, from the actual computations that needed to be performed on raw data. Although their computations were relatively simply, e.g. counting word frequencies in documents during inverted index constructions, often times that simplicity was obscured by the additional logic required to take care of distributing computing issues.

The model uses two primitives, *map* and *reduce*, to express the logic of the computations. These primitives are inspired in the *map* and *reduce* functions commonly found in functional programming languages. Input and output of these functions are expressed as *key/value* pairs. Underneath, MapReduce implements a runtime

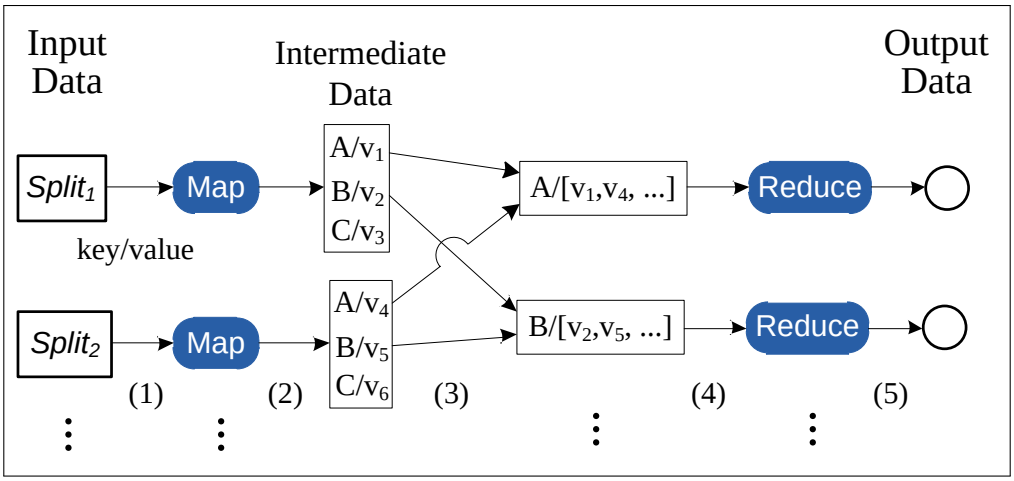


Figure 3.2: MapReduce job execution overview.

framework that relies on cluster management systems and distributed file systems that deal with the complexities of paralellizing computations. The main characteristics of the model are as follows.

- It separates the *what* needs to be done, which is specified in the *map* and *reduce* functions, from the *how* the distributed computations are carried out, which is taken care by the MapReduce framework. As a result, application developers focus their attention on the data processing logic only.
- It scales reliably to clusters with thousands of computers. Should a node running a mapper or reducer fail, only the data portion being processed by the faulty node is lost.
- It implements a shared-nothing cluster architecture (each node has its separate storage, CPU and memory) with data replication.

Figure 3.2 shows an overview of the execution of a MapReduce compound. The input data is stored in the Google's distributed file system (GFS) [GGL03], which is optimized for large files. Files are divided into splits of predefined size; typical sizes are 64MiB and 128MiB. A MapReduce job starts with mappers reading individual

splits of the input data and transforming them into *key/value* pairs for processing by the *map* function. Mappers generate intermediate data also in *key/value* pair format. In general, a mapper generates records of one or several types of *keys* as seen in Figure 3.2 where mappers generate records with A, B and C keys. The map phase finish when all mappers process their input records in their entirety. In a second phase, all the values associated to the same *key* are copied by one reducers – this is known as the shuffle phase. For instance, the list of values $[v_1, v_4, ...]$ associated to the A key are transferred to one reducer in Figure 3.2. A reducer may receive several types of *keys*, but they are processed in sorted order. Finally, each reducer produces an output after applying the *reduce* function on their input values. The MapReduce output is the union of the individual reduce outputs, which are generally kept partitioned in the distributed file system rather than combining them in a single file.

3.2.3 MapReduce in Practice

Google's implementation of MapReduce is not available outside the company. The cluster used in this dissertation was provided by the NSF Cluster Exploratory (CluE) program, and the cluster operation was maintened by Google and IBM [Pro08] [Ini07].

The CluE cluster was shared among a dozen educational institutions. At the time we ran out experiments the cluster contained around 480 compute nodes running the Linux operating system, XEN hypervisor, and Apache Hadoop [Pro11], the open source implementation of the MapReduce programming model. Each node had half terabytes storage capacity summing up to about 240 terabytes in total. Access to the cluster was provided through the Internet by a SOCKS proxy server.

SOCKS is an Internet protocol that secures client-server communications over a non-secure network. There are three main steps in interacting with the cluster. First, input data is uploaded into the cluster by users. The Hadoop distributed file system (HDFS) provides file system shell scripts to upload to and download from the cluster; HDFS is an integral part of the Apache Hadoop project, and its implementation is based on the Google file system (GFS) [GGL03]. Second, users develop a Hadoop application and submit it as a job to the cluster via a Hadoop command where the job is scheduled for execution. Application in Hadoop are developed in Java, but other languages are supported, like C++ and Python. Third, once the execution of Hadoop jobs is completed, the output is downloaded to the users' local site via Hadoop file system shell scripts.

3.3 Building R-trees with MapReduce

This section presents a MapReduce-based algorithm for building an R-tree index on a given spatial dataset in a parallel fashion. Let us start our description by formalizing the problem.

3.3.1 Problem Statement

Let D be a spatial dataset composed of objects $o_i, i = 1, .., |D|$, where $|D|$ represents the number of objects in the database. Each object o has two attributes $< o.id, o.P >$, where $o.id$ is the object's unique identifier and $o.P$ is the object's location in some spatial domain (other attributes are possible, but we concentrate on only these ones for the R-tree construction purpose). The problem is to construct in MapReduce an R-tree index as defined in [Gut84].

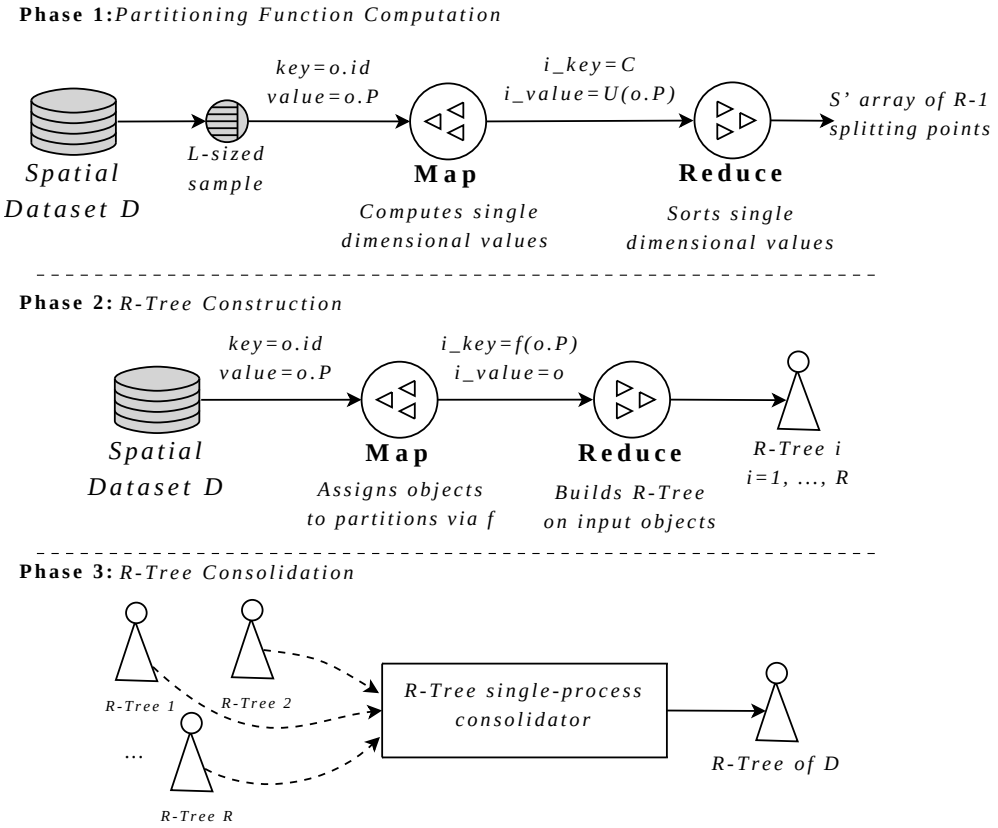


Figure 3.3: Main phases involved in building an R-tree index for a spatial dataset D in MapReduce.

3.3.2 Approach Overview

The proposed method consists of three phases executed in sequence, as can be seen in Figure 3.3. First, spatial objects are partitioned into groups. Then, each group is processed to create a small R-tree. Finally, the small R-trees are combined into the final R-tree. The first two phases, namely data partitioning and simultaneous construction of small R-trees, are executed in MapReduce. The third phase, namely combination of several R-tree root nodes under a common root node, does not require high computational power, thus it is executed sequentially outside of the cluster. The three main phases of the algorithm are discussed next.

Phase 1: Partitioning Function Computation

The inputs for this phase are the dataset D and a positive number R, which represents the number of partitions. The purpose of the partitioning function f is to assign each object $o \in D$ into one of the R possible partitions. The function is computed in such a way that after applying f on all objects of D, we should (ideally) obtain R equal-sized partitions. In practice, minimal variance in sizes is acceptable. At the same time, f attempts to put objects that are close in the spatial domain in the same partition. The output of this phase is a function: $f : D \rightarrow \{1, 2, \ldots, R\}$. In actuality, the function f uses object's location $o.P$ to assign it to a partition number. Note that no actual data partitioning or move happens at this point. More details of this step are presented in Section 3.3.3.

Phase 2: R-tree Construction

During this phase, the function f calculated in the first phase is used by mappers to divide D into R partitions which are passed to reducers. Then, R reducers build simultaneously R independent (small) R-tree indexes on their input partitions. The output of this phase is a set of R R-tree root nodes. Details of this step are presented in Section 3.3.4.

Phase 3: R-tree Consolidation

This phase combines the R individual R-trees built in the second phase under a single root node to form the final R-tree index of D. This phase can be as simple as making the R R-trees children of a single root node, or it may require adding a few extra levels (at most one in practice) if R exceeds the capacity of a single R-tree node. Since this phase is not computationally intensive for R under a few hundreds

or even thousands, it is executed by a single process outside the cluster. The logic to run this phase is fairly simple, and it is summarized in Algorithm 3.1.

Algorithm 3.1 R-tree Consolidation

Input: *roots* - List of individual R-tree root nodes.
Output: Root node of the consolidated R-tree.

```
1: rTree ← initialize empty Rtree
2: for all r ∈ roots do
3:     rTree.insert(r.id, r.MBR)
4: end for
5: return rTree
```

The algorithm first creates a new R-tree. Then, it makes succesive insertions of the individual R-tree root nodes into the newly created R-tree in line 3. An implicit assumption of the algorithm is that all the underlying R-trees are all of the same height. In case the assumption is not met, the algorithm should first level up all the individual R-trees, for example, by inserting underflow nodes in the shorter R-trees [PM03].

3.3.3 Data Partitioning

Data partitioning is a key issue that needs to be addressed in parallel processing approaches. The choice of a partitioning scheme usually depends on how the target application intends to retrieve data.

We consider a partitioning function f whose purpose is to provide a means for assigning objects of D to one of a pre-defined number of R partitions. We use the idea of mapping multi-dimensional spaces into an ordered sequence of single-dimensional values via space-filling curves to define the partitioning function. This idea has been studied in the literature as a way to numbering objects in multi-dimensional spaces [ARR+97] [LK00]. In the present problem, we map objects' geographical locations ($o.P$) into a space-filling curve. In particular, we use the Z-

order curve [Mor66] in our experiments in Section 3.4. The partition number of an object o is determined by $f(o)$, which evaluates to a value from the set $\{1, 2, .., R\}$. By using a space-filling curve, the partitioning function f achieves two goals:

- Generate R (ideally) equal-sized partitions.
- Preserve spatial locality. That is, if two distinct objects o_1 and o_2 are close to each other in the spatial domain, then they are likely to be assigned to the same partition, i.e. $f(o_1) = f(o_2)$.

Next, we propose a MapReduce algorithm to define f.

MapReduce Algorithm

The general idea is inspired by the TeraSort Hadoop application [O'M08], which partitions an input dataset via data sampling. Given a dataset D and target number of partitions R, the MapReduce algorithm runs M mappers that collectively take L sample objects from D, and emit their single-dimensional values $S = \{U(o_i.P), i = 1, .., L\}$ for a given space-filling curve U. That is, each mapper samples $\frac{L}{M}$ objects from its $O(\frac{|D|}{M})$ input objects. Then, a single reducer sorts the values in S in ascending order, and it determines a list S' of $R - 1$ splitting points that split the sorted sequence into R equal-sized intervals. Then, an object o belongs to partition j if $S'[j-1] < U(o.P) \leq S'[j]$. That is, f utilizes the splitting points in S' to assign objects to partitions.

The specific MapReduce key/value input pairs as well as outputs are presented in Table 3.1. Mappers read in total L samples at random offsets of their input dataset D, and they compute their single-dimensional value with a space-filling curve U. The intermmediate key equals to the constant C, whose value is irrelevant, that helps in sending mappers' intermediate outputs to a single reducer. The reducer

Table 3.1: Map and Reduce inputs/outputs in computing partitioning function f.

Function	Input: (Key, Value)	Output: (Key, Value)
Map	$(o.id, o.P)$	$(C, U(o.P))$
Reduce	$(C, list(u_i, i = 1, .., L))$	S'

receives the L single-dimensional values generated by the mappers, and sorts them into an auxiliary list $[u_1, .., u_L]$, from which $R-1$ elements are taken starting at the $\lfloor \frac{L}{R} \rfloor$-$th$ element and subsequently at $\lfloor \frac{L}{R} \rfloor$ fixed-length offsets to form the list S' of splitting points.

An important observation in the sampling process is that mappers read input data from the distributed storage at block-sized amounts, which is a Hadoop distributed file system parameter specifically tuned for load balancing large files across storage nodes. Thus, all mappers, except perhaps for the last one, will read the same amount of data, equal to the file system block size. The rationale of the splitting points in S' is that they provide good enough boundaries to sub-divide D into R partitions since they come from randomly sampled objects. Experiments in Section 3.4 show fairly well balanced partitions using the Z-order space-filling curve; low standard deviation (under 1%) is observed on the number of objects per partition. Formally, the function f is defined by Equation 3.1.

$$f(o) = \begin{cases} 1 & \text{if } U(o.P) \leq S'[1] \\ j & \text{if } S'[j-1] < U(o.P) \leq S'[j], j = 2, \ldots, R-1 \\ R & \text{otherwise} \end{cases} \tag{3.1}$$

This computation is characterized by running multiple mappers (samplig data), and one reducer (sorting samples). The single reducer may become a limiting factor in scaling the solution when the number of samples L is substantially large. In such case, an approach similar to the TeraSort algorithm [O'M08] could be used to sort

Table 3.2: MapReduce functions in constructing R-trees.

Function	Input: (Key, Value)	Output: (Key, Value)
Map	$(o.id, o.P)$	$(f(o.P), o)$
Reduce	$(f(o.P), list(oi, i = 1, .., A))$	$tree.root$

S in parallel, which makes the algorithm for computing the partitioning function scalable.

3.3.4 R-tree Construction

In this phase, R individual R-tree indexes are built concurrently. Mappers partition the input dataset D into R groups using the partitioning function f. Then, objects assigned to a particular partition are received by the same reducer, which independently builds an R-tree on its input partition. Next, every reducer outputs a root node of their constructed R-trees. That is, R sub-trees are written to the file system at the end of this phase. MapReduce input and output key/value pairs are shown in Table 3.2. Mappers read their input data in its entirety and compute objects assigned partitions via $f(o)$. Then, every reducer receives a number of input objects A for which an R-tree is built and its root emitted as output.

Since f attempts to balance partition sizes, it is expected that all reducers will receive a similar number of objects ($A \sim \frac{|D|}{R}$), thus executing similar amount of work in constructing their R-trees. However, good balancing depends on two factors: a) the underlying space-filling curve U used by f, and b) the number of sampled objects L. More samples help in tuning the splitting points, but incur in larger sorting time of L elements. Another concern is the quality of the produced R-trees in relation to the parameter R. In Section 3.4, we provide some insights into this direction by measuring R-tree parameters such as area and overlap in a simplified way, and plotting their MBRs for visual analysis.

3.4 Experimental Evaluation

This section presents and discusses the experimental results we obtained by running the algorithms described in Sections 3.3.3 and 3.3.4 as Hadoop applications on the Google&IBM's cluster discussed in Section 3.2.3. The datasets used in this section are real spatial datasets supplied by the Florida International University's High Performance Database Research Center [Cen11]. At the time of our experimentation, there were jobs running in the cluster from other researchers that share this resource, thus some fluctuation in the results is expected.

3.4.1 Datasets and Setup

Experiments were executed on two real spatial datasets. Dataset descriptions are shown in Table 3.3. Location of objects in the spatial datasets are defined by geographical coordinates represented by latitude and longitude values. For the partitioning function f, we used the Z-order space-filling curve [Mor66] as U function to map the two-dimensional points into a single dimension. We set the sampling size as $L = (3\% \times |D|)$ for all datasets.

Objects in datasets are stored in semi-structured format (Tab delimited files), where each line represents an object. We used Hadoop supplied record reader functions to convert text lines from the datasets into objects. During the second phase, reducers built their individual R-trees in main memory. In the final step of reducers, R-trees were persisted on the Hadoop distributed file system.

Table 3.3: Spatial datasets used in experiments.

Dataset	Objects (millions)	Size (GB)	Description
FLD	11.4	5	Property parcels in the state of Florida.
YPD	37	5.3	Yellow pages directory of businesses mostly in the United States but also in other countries.

3.4.2 Performance Evaluation

This experiment consists of building R-tree indexes on the Google&IBM Hadoop cluster varying the number of indexes that are built in parallel, that is, the parameter R in Phase-2. R varied from 2 up to 64 in multiples of two. As R varied, job completion times were measured for mappers and reducers. In addition, quality statistics on the resulting R-trees were measured.

On the other hand, we ran a single-process R-tree construction on a dedicated local machine equiped with Intel Xeon E7340 2.4GHz processor and 8GB of RAM running Windows OS. We could not run the single process in the cluster since, as described in Section 3.2.3, the cluster was provided as a platform as a service (PaaS) cloud, thus no login access to individual nodes was possible. Therefore, cluster and single-process times are not comparable due to dissimilar hardware.

Table 3.4 shows MapReduce job completion times for R-tree construction phases 1 and 2 for each spatial dataset, as well as for a single-process builder (SP); for *YPD* we started at $R = 4$ due to memory limitations in cluster nodes for building in-memory trees with less number of reducers. We do not include phase-3 processing times since its time is negligible compared to the other phases. Phase-1 (partitioning function computation) takes very little time compared to Phase-2, which is expected

Table 3.4: MapReduce job completion times (in minutes) for the Phase 1 (MR_1), and various Reducers (R) in Phase 2 (MR_2) of building an R-tree. Also, completion times for single-process (SP) constructions ran on a local machine are shown.

		MR_1: f comp.		MR_2: R-tree		
Dataset	R	Map	Reduce	Map	Reduce	Total: $MR_1 + MR_2$
FLD	2	0.35	0.28	0.40	24.12	25.15
	4	0.28	0.23	0.40	11.07	11.98
	8	0.47	0.22	1.73	5.62	8.03
	16	0.30	0.22	0.40	3.05	3.97
	32	0.48	0.23	0.40	1.95	3.07
	64	0.28	0.33	0.45	1.60	2.67
	SP	-	-	-	-	27.34
IYP	4	0.47	0.38	0.47	52.57	53.88
	8	0.22	0.45	0.72	25.42	26.80
	16	0.40	0.43	0.38	8.93	10.15
	32	0.40	0.43	0.42	4.65	5.90
	64	0.40	0.42	0.88	2.55	4.25
	SP	-	-	-	-	63.98

since $L = (3\% \times |D|)$ number of elements require little main memory to store them in an array, and the sorting can be done quickly with a $O(L \log L)$ algorithm. For our largest dataset *YPD*, about 1 million elements are sampled. Our Z-order values are 8-byte sized elements, so around 8MB of RAM is needed to execute the sort operation, which is much less than the available main memory of cluster nodes. Likewise, mappers in Phase-2 read data sequentially and execute inexpensive Z-order value computations on their inputs. The most computationally intensive part is performed by reducers in Phase-2, where the actual R-tree constructions occur. The fewer the number of reducers, the longer the R-tree construction takes because each task receives larger number of objects.

Figure 3.4 shows job completion times as stacked bars of the map and reduce execution times. In this figure, almost linear scalability is observed as more parallelism is induced by increasing R in Phase-2. Performance improvement rate is high

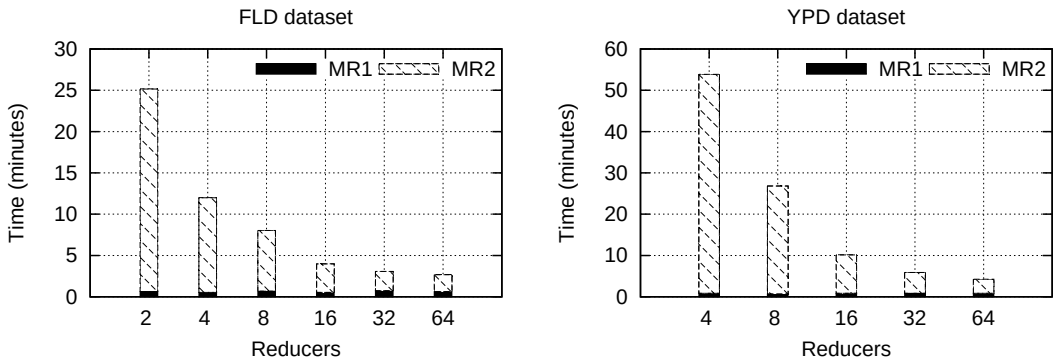


Figure 3.4: MapReduce job completion times for *FLD* and *YPD* datasets.

for few reducers, but soon improvements drop as the number of reducers increases since partitioning overheads in Phase-1 (MR_1) start becoming significant compared to R-tree build times in Phase-2 (MR_2). In fact, for larger values of R, the dominating time component is given by MR_1, which, as can be seen in Table 3.4, is almost constant for a given dataset. Thus, much less improvements are expected as R is increased beyond 64.

Although we cannot compare the cluster and single process (SP) times due to mismatches in hardware configuration, the MapReduce parallelization certainly yields performance benefits for large-scale datasets. For example, it takes more than an hour to sequentially build the *YPD* R-tree, while the task can be achieved in parallel in less than 5 minutes with 64 Reducers. On the other hand, the resulting R-trees are different due to differences in object insertion sequences. Later in this section we measure and discuss R-tree quality parameters for both cases.

Figure 3.5 presents percentages of performance gains in job completion times in relation to subsequent increases of the number of reducers in the second phase of the algorithm. For example, in the *YPD* dataset, going from 4 to 8 Reducers we observe 50% decrease in job completion time, which represents linear scalability. On the other hand, going from 8 to 16 reducers shows super-linear scalability (62%).

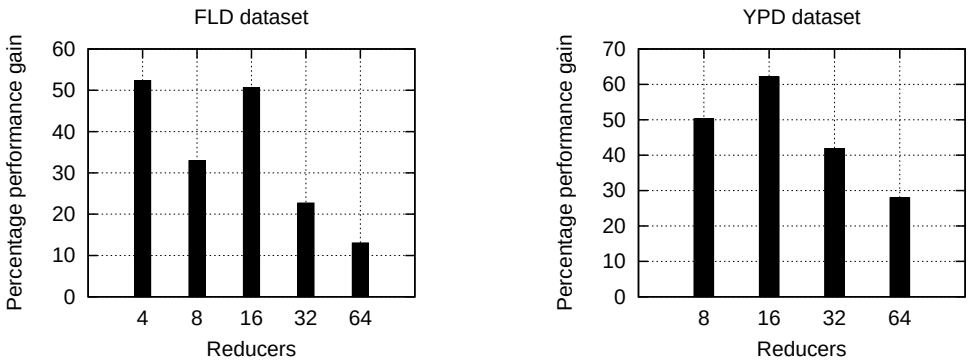


Figure 3.5: MapReduce job percentage of performance gains as the number of reducers is increased.

We pressume this may be due to heterogeneous nodes in the cluster (eventually, the Hadoop job with $R = 16$ was scheduled to be executed on faster nodes), or it may be the cluster resources were idler during that period. As discussed, as we further increase the number of reducers, performance gains are less significant because the execution time for the first phase, which has a sequential component (single reducer), stays almost constant.

R-tree Construction Phase (MR_2): Performance Analysis

As the number of reducers increase beyond 16 in Figure 3.4, MapReduce completion times do not decrease linearly. In fact, for large number of reducers, diminishing returns are observed. We performed additional experiments to further analyze the causes of the previous behavior. In particular, we focused on the reduce task of the second MapReduce job (MR_2), where R-tree constructions take place, which is effected by the number of reducers. The first MapReduce is executed with the same number of mappers and a single reducer, thus its completion time is nearly constant.

(We originally ran the experiments for the results in Figure 3.4 in 2009 while the additional experiments in this section were run in 2011. There were two main

Table 3.5: Reduce task three primary phases.

Phase	Operation	Description
1	Shuffle	Intermediate data generated by mappers is copied over the network by reducers.
2	Sort	Reducers sort their local data by key.
3	Reduce	The user-provided function $reduce(key, list(.))$ is executed.

differences in the cluster environment within this period of time. First, the cluster has been expanded in capacity and newer hardware was added – 900 nodes versus 400 nodes in 2009. Second, a newer version of MapReduce Hadoop was running – Hadoop 0.20 versus 0.17 in 2009. Thus, faster completion times are expected in the experiments of this section. However, regardless of the cluster improvements, the performance analysis of this section is still statistically significant.)

Reduce Task Phases

According to the Hadoop documentation [Red11], a reduce task has three main phases as described in Table 3.5. In the shuffle phase, reduce tasks copy their data from the mappers over the network as intermediate data becomes available. The sort phase is executed simultaneouly with the shuffle phase. That is, data is merge sorted by *key* as it is copied. Finally, the reduce phase starts after all the data from mappers has been copied and sorted. In this last phase, the $reduce(.)$ function is invoked for every *key* and list of *values* found in the sorted input.

The completion time of the whole reduce phase is given by the longest running reduce task. In practice, the reduce phase includes setup and cleanup operations that happen at the beginning and end of the reduce phase, respectively. Although

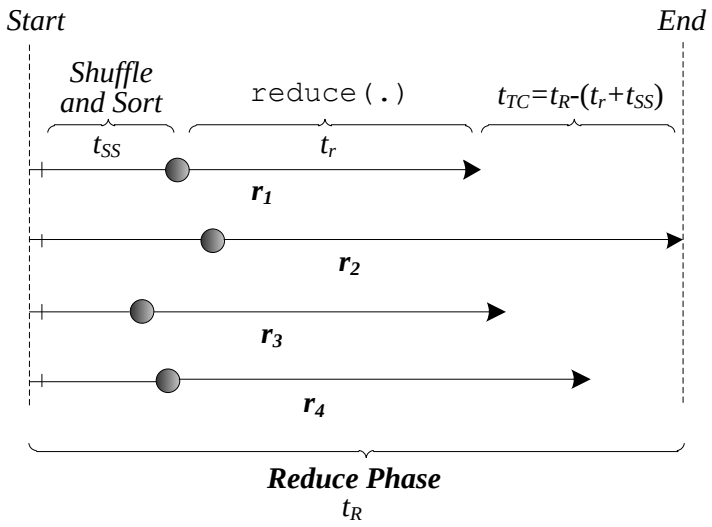


Figure 3.6: Reduce phase time components with four reduce tasks.

startup/cleanup time costs fluctuate, they show slight variance accross different runs that can be considered constant cost. So, we disregard these times in our analysis.

Since the cluster is composed by heterogeneous hardware, it is expected that some reducers would take longer to complete even when the input size is comparable accross all reducers. The backup task mechanism of the MapReduce framework aims at alleviating such situation by running backup reducers of those that are taking longer to finish than the rest. Figure 3.6 illustrates the time components discussed in a reduce phase with four reduce tasks. We summarize the notation used in our experiments in Table 3.6.

Reduce Task Performance Experiments

We re-ran the R-tree construction MapReduce jobs for the FLD dataset and measured the times of Table 3.6 in the reduce phase of the jobs.

Figure 3.7 shows average times in the reduce phase varying the number of reducers from 2 up to 128. In the figure, we can observe change trends of the time components as the number of reducers increase. First, the reduce time cost (t_r) de-

Table 3.6: Notation summary of reduce task time components.

Variable	Description
t_{SS}	Shuffle and sort combined time.
t_r	Time spent in the $reduce(.)$ function execution.
t_R	Total time of the reduce phase.
t_{TC}	Time to completion of reduce phase. $t_{TC} = t_R - (t_{SS} + t_r)$.

creases almost proportionally as the number of reducers doubles, which is expected since the input size per reducer halves at each reduce increase in the X-axis.

Second, the shuffle and sort cost increase slightly as the number of reducers becomes larger, but it is kept around 40 seconds most of the time. However, for number of reducers approaching 128, shuffle and sort times become significant compared to the total reduce phase time. Hence, shuffle and sort times are one reason that precludes linear scalability for large number of reducers. That is, although reduce times decrease proportionally to the number of reducers, the shuffle and sort cost does not.

Third, the difference in execution time among the longest running reduce task and the rest of the tasks is significant for four reducers or more as it can be infered by the t_{TC} times of Figure 3.7. A possible reason is that the more reducers, the higher the probability to execute them in machines with dissimilar hardware characteristics. Thus, the reduce task scheduled in the slowest machine will take longer to finish. On the other hand, even if all the machines in the cluster had similar hardware characteristics, another reason may be that the slowest reduce task was scheduled in a machine that was busy processing other MapReduce tasks. Therefore, the effect of the longest running reducer also makes the MapReduce job less linearly scalable,

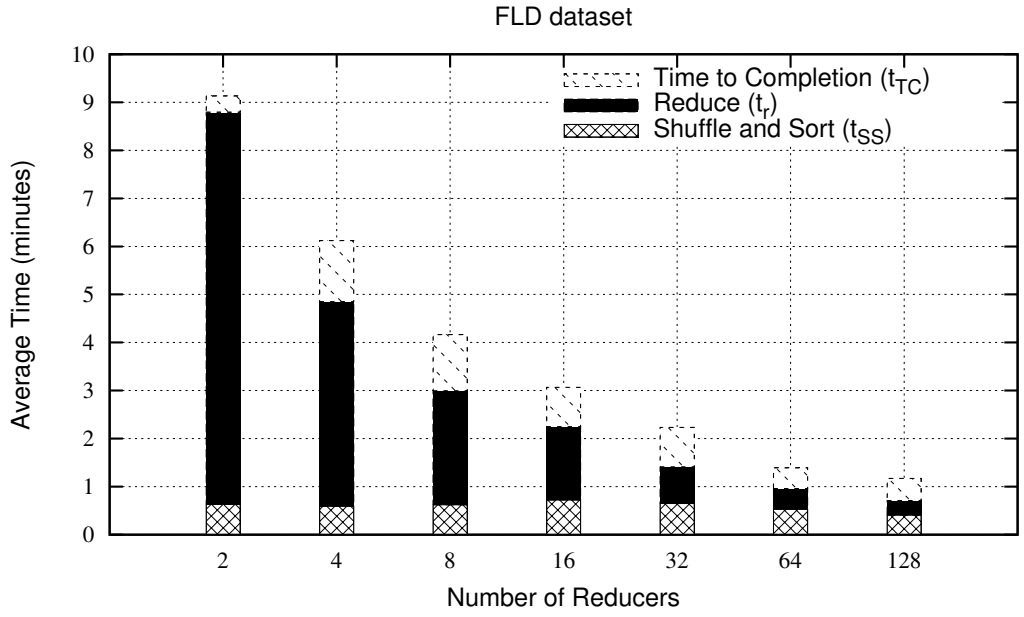


Figure 3.7: Reduce phase time components in MR_2. Times are the average of five MapReduce job executions.

especially for larger number of reducers.

3.4.3 Quality of Generated R-trees

We use Equations 3.2 and 3.3 to compute the area and overlap metrics, respectively, for a given consolidated R-tree with root T:

$$Area(T) = \sum_{i=1}^{n} Area(T_i.MBR) \tag{3.2}$$

$$Overlap(T) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} Area(T_i.MBR \bigcap T_j.MBR) \tag{3.3}$$

where n is the number of children (small R-trees generated by reducers) of T, and T_i is the i-th entry in the T node. Note that other metrics of R-tree quality could be considered as well; for example, include all the nodes of the R-tree in computing the metrics instead of just the top level.

Table 3.7: Statistics on consolidated R-trees built by various number of reducers (R), and single process (SP) construction.

		Objects per Reducer		Consolidated R-tree			
Dataset	R	Average	Stdev	Nodes	Overlap (sq.mi)	Area (sq.mi)	Height
FLD	2	5,690,419	12,183	172,776	132,333.9	304.4	4
	4	2,845,210	6,347	172,624	106,230.4	4,307.9	4
	8	1,422,605	2,235	173,141	103,885.8	17,261.9	4
	16	711,379	2,533	162,518	96,443.1	21,586.3	4
	32	355,651	2,379	173,273	140,028.7	80,389.1	3
	64	177,826	1,816	173,445	152,664.2	96,857.7	3
	SP	11,382,185	0	172,681	746,145.0	1,344,836.8	4
YPD	4	9,257,188	22,137	568,854	26.511M	21.574M	4
	8	4,628,594	9,413	568,716	23.160M	20.480M	4
	16	2,314,297	7,634	568,232	67.260M	54.582M	4
	32	1,157,149	6,043	567,550	68.627M	54.008M	4
	64	578,574	2,982	566,199	69.791M	55.064M	4
	SP	37,034,126	0	587,353	164.967M	658.583M	5

Table 3.7 shows quality metrics on the consolidated R-trees built for various number of reducers and single process (SP). As reference, the U.S. Census Bureau reports Florida state land area roughly as 54,000 square miles in the 2000 census [SQ08]. We observe that the total MBR area and the overlap (measured both in square miles) increase as the parallelism (R) increases because the construction of each small R-tree is unaware of the rest of the dataset. The latter lowers the chance of co-locating neighbor objects within the same R-tree. Smaller MBR area and less overlap are known to improve search performance since they increase path pruning abilities of R-tree navigation algorithms [BKSS90]. This means that we degrade the R-tree quality, marginally gaining in execution time. The latter can adversely effect performance of search algorithms, such as nearest neighbor type of queries due to extra I/O operations incurred in traversing multiple sub-trees.

For a sequential construction (SP), we observe that the area and overlap metrics

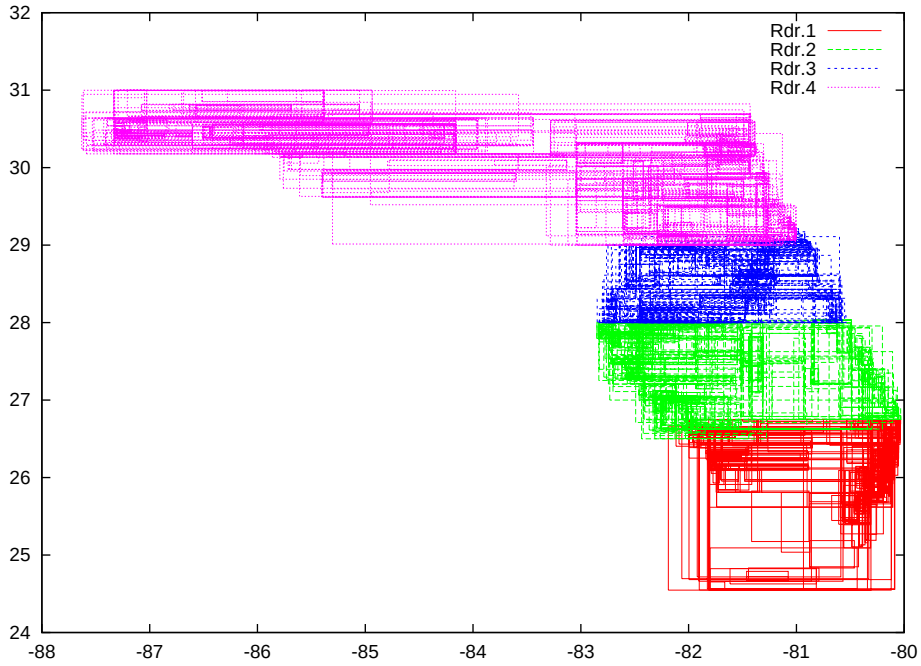


Figure 3.8: MBR plotting of the R-tree created for *FLD* by MapReduce with $R = 4$.

are much worse, especially the overlap factor, since objects are not spatially shuffled but rather inserted in the dataset original sequence. Thus, higher performance penalties are expected in *SP* constructed R-trees. On the other hand, the tree height slightly decreases for *FLD* for R beyond 32 because more small trees means that each one of them may be shorter, while for *YPD* the height increases by one level for the *SP* case. In general, small variations in tree height is less significant from a performance standpoint.

To visually study the effect of increasing R over the MBR distribution, we have plotted the MBRs of the resulting R-trees for the case of 4 and 8 reducers in Figures 3.8 and 3.9, respectively, for the Florida state dataset (*FLD*). The same type of plotting is shown in Figure 3.10 for the SP R-tree. In neither case is the root MBR plotted since it is common for all trees.

A few observations can be made from the MBR plottings. First, the partitioning

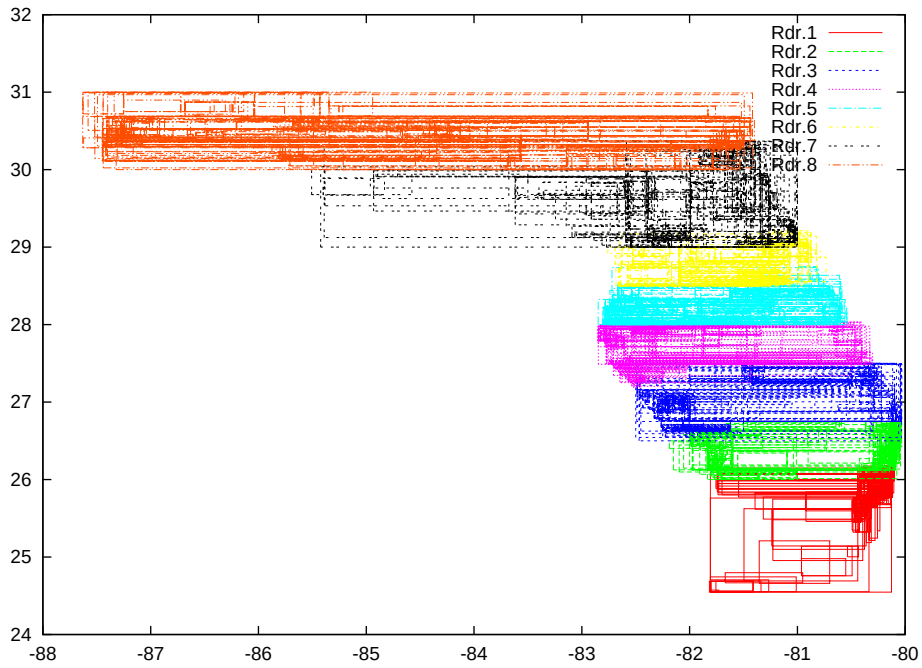


Figure 3.9: MBR plotting of the R-tree created for *FLD* by MapReduce with $R = 8$.

mechanism employed in our algorithms seems to be effective in preserving spatial locality. This results in individual reducers indexing highly localized objects; their boundaries, however, result in multiple overlappings, which are inevitable. Second, as the number of reducers is increased from 4 to 8, the plotting shape resembles more the actual shape of the Florida state map; that is, $R = 8$ reduces wasted areas (where no actual objects are located) as the area statistic confirms in Table 3.7. In fact, Table 3.7 shows steady decrease in area from 2 to 16 reducers; after that the area keeps on increasing. Third, when the R-tree is built on the original sequence of objects (no object shuffling) in *SP* mode, large wasted areas are generated as can be observed in Figure 3.10. From a performance optimization perspective, MapReduce generated R-trees seem to be better tuned than their single-process counterpart. Therefore, we see promising performance improvements in MapReduce generated R-trees, which deserve closer verification.

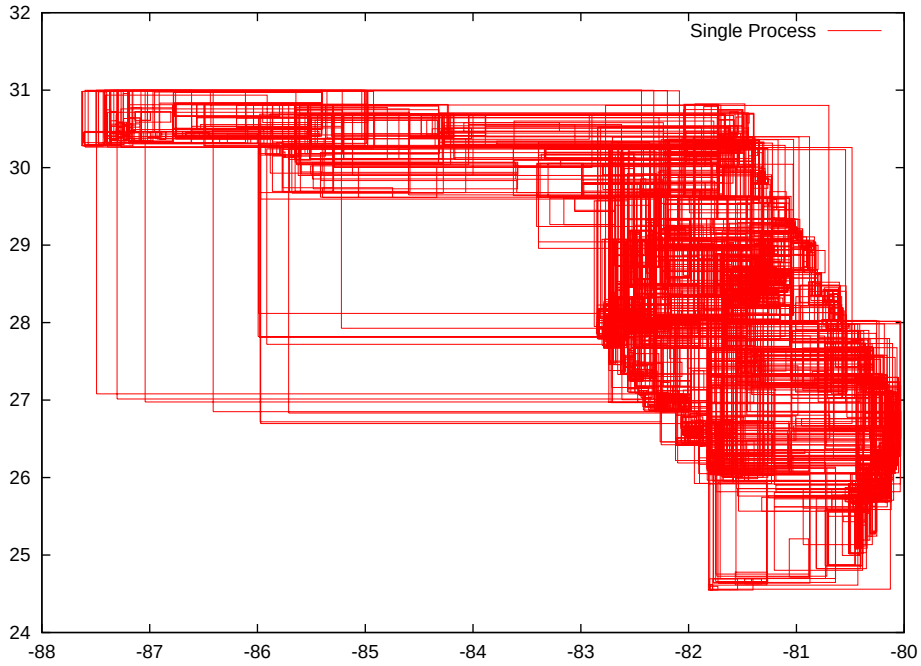


Figure 3.10: MBR plotting of the R-tree created for *FLD* by a single process.

3.5 Related Work

Space-filling Curves

The idea of using space-filling curves to map multi-dimensional spaces into a single dimension has been studied for the case of spatial databases [AM90] [LK00]. Popular space-filling curves, such as Peano and Hilbert, have been studied in great level of detail. In this work, we used the Z-order curves in our experiments. This curve showed high spatial locality preservation for our experimented real datasets. Other curves can certainly be evaluated, which goes beyond our focus on the parallelization problem of building R-trees with MapReduce.

Parallel R-tree Constructions

Previous works on R-tree parallel construction have faced several intrinsic distributed computing problems such as data load balancing, process scheduling, fault tolerance, etc., for which they elaborated special-purpose algorithms. Schnitzer and

Leutenegger [SL99] propose a Master-Client R-tree, where the dataset is first partitioned using Hilbert packing sort algorithms, then the partitions are declustered into a number of processors (via an specialized declustering algorithm), where individual trees are built. At the end, a master process combines the individual trees into the final R-tree. Another work by Papadopoulos and Manolopoulos [PM03] proposed a methodology for sampling-based space partitionining, load balancing, and partition assignment into a set of processors in parallely building R-trees. They also discuss alternatives when the global (consolidated) index has imperfections, such as different heights across individual R-trees. In the simplified MapReduce parallel model, distributed computing concerns are abstracted out from the application logic, and managed transparently as part of the MapReduce framework. Further, all nodes in the cluster access a common distributed file system, with automatic fault-tolerance and load balancing support. Data locality is employed as the base criterion to assign mappers and reducers (preferably) to nodes already containing the input data. In contrast, traditional parallel processing works assume every node has its own storage, in a shared-nothing type of architecture, where data transfer among nodes becomes an important optimization goal.

MapReduce on Spatial Data

The MapReduce framework was used to solve other spatial data problems by Wu et al. at Google [WCF+07] where they study the problem of road alignment by combining satellite and vector data. This work concentrates on the complexities of the problem, which are more challenging than the MapReduce algorithms. Schlosser et al. [SRT+08] worked on building Octrees in Hadoop for later use in earth-quake simulations at large-scale. Their approach builds a tree in a bottom up fashion. The map function in the first iteration generates leaf nodes, then the reduce function coalesces homogeneous leaf nodes into a subtree. Subsequent iterations have identity

functions in mappers, and successively use reduce functions to construct the final tree.

Relationship to MPI

Message Passing Interface (MPI) [GLS94] is a specification of a language-independent communication model targeted at writing parallel programs, and it is widely used in a variety of computer cluster platforms. MPI libraries provide primitives and functionality for communication control among a set of processes. Typically, developers need to add explicit calls to synchronize processes and move data around. The key differences between MPI and MapReduce is that MapReduce exploits its simplified model to automatically parallelize tasks (via map and reduce tasks), hiding from programmers the need to worry about process communication, fault-tolerance, and scalability, which are transparently managed by key components of the MapReduce framework, such as cluster management system and distributed file system, that the MapReduce framework is built-upon [DG08]. For example, for the R-tree case study, the Java implementation of the Map and Reduce functions of the first phase, and Map of the second phase have each less than 40 lines of code. The Reduce function in the second phase has about 70 lines of code since it includes extra code for persisting the tree on the distributed file system and collecting build statistics. These numbers do not include application-specific routines, which are needed regardless of the parallel model.

On the other hand, the underlying assumption in MapReduce is that the solution can be expressed in terms of the Map and Reduce functions, and that data can be represented as key/value pairs. In some cases this may not be natural, such as relational joins, or multi-stage processes, and can lead to inefficiencies. Then, MPI-like parallel implementations have more opportunities to address application-specific optimizations, due to its finer process control. However, high-level languages have

been proposed to address this problem in MapReduce architectures by providing efficient primitives for massive data analysis combining SQL-like declarative style with MapReduce procedural programming style [YDHP07] [ORS+08].

3.6 Summary

In this chapter, we studied how the MapReduce parallel programming model can be used to solve the problem of R-tree spatial data structure construction on large datasets. The proposed MapReduce algorithms were implemented in the open source Hadoop framework and executed on a Google&IBM Hadoop cluster. The experimental results we obtained indicate that the appropriate application of MapReduce could dramatically improve task completion times. Our experiments show close to linear scalability in R-tree index construction tasks. However, performance is not the only concern for R-tree construction, which is sensitive to the ordering of objects in its input, but also the quality of the result. MapReduce generated R-trees have improved quality in terms of MBR area and overlap measurements compared to the single-process construction counterpart. Our experience in this work shows that MapReduce has the potential to be applicable to more complex spatial problems.

CHAPTER 4

SPATIAL QUERIES WITH ASPATIAL CONSTRAINTS

4.1 Introduction

Geolocation information is becoming ubiquitous in modern databases. More often, databases store geolocation data such as street addresses, postal codes, cities or place names, in addition to the traditional descriptive fields. For instance, a real estate database may store the street address or neighborhood where a home on sale is located together with its price, number of bedrooms, and a description of the residence. In other cases, geolocation data can be derived from existing data, for example the postal code or city could be determined from the machine's IP address of a web request event. On the other hand, advances in geolocation technologies allow users to associate a location to objects or events. For example, modern smartphones have built-in GPS devices that allow users to associate geographical coordinates (with small error) to pictures or videos, or to record the place the user has been in a social event via a Web 2.0 application. Therefore, nearly everthing can be tagged with a geolocation on the map.

In the recent past years, several research works have tackled the problem of efficiently answering geospatial queries, such as k-nearest neighbor (k-NN) or window queries, with exact and approximate keyword constraints [ABL10] [CJW09] [DFHR08] [HHLM07] [PK03]. For example, in the real estate database, a user may be interested in finding houses for sale nearby *Miami Beach* (spatial constraint) that have *backyard* in their description (textual constraint) and their streets addresses contain *Collins Avenue* (textual constraints). Existing works typically assume that query keywords are conjunctively connected. That is, objects containing all query terms are retrieved. In the general case, it is desirable to include multiple constraints

on textual and numeric attributes, and to connect constraints with different logical operators (beyond the conjunctive semantics) [CWR10]. For instance, in a database of property parcels, fire fighters traveling in a truck may want to quickly determine the *nearest* parcels that have *swimming pool* and are not located in *buildings* for water replenishment purposes in an emergency.

On the other hand, spatial databases also store a variety of numeric attributes. Despite numerous research works on keyword searches in geospatial databases, little research was done in efficiently supporting location-aware queries with numeric constraints. In particular, numeric range constraints are useful for several domain-specific application, such as real estate listings or product sales, and Web mapping services (like Google Maps [mGM] or Bing Maps [mBM]), when users have only a rough idea of the value they expect to find, or simply when finding an exact value is not relevant. For example, in the real estate database a buyer may post the following query:

"*Find the nearest homes for sale with prices between $50,000 and $80,000*"

As geospatial databases increase in size, the requirement of efficient processing of spatial queries with textual and numeric constraints becomes more important.

In this chapter, we propose methods for efficiently processing k-nearest neighbor (k-NN) queries with textual and numeric constraints. We consider the case where constraints are combined with the three basic Boolean operators: AND, OR, and NOT. We assume the database follows the data model presented in Section 2.3 of Chapter 2. Intuitively, the result of k-NN spatial Boolean selection query is a list of the k nearest objects in the database, sorted by distance to the query location, that satisfy a Boolean selection criteria.

The proposed method combines two tree-based indexes: R-trees to organize objects by their spatial coordinates, and B^+trees to store textual and numeric data

with references to the R-trees[1]. The combined data structures result in a novel hybrid index of spatial, textual, and numeric data.

The specific contributions of this chapter are:

1. We define a k-NN spatial Boolean query (k-SB) that finds the k-nearest neighbor objects satisfying a Boolean selection criteria on aspatial database attributes. Constraints can be combined with conjunctive ($\wedge$), disjunctive ($\vee$), and complement ($\neg$) logical connectives.

2. We propose a novel hybrid index to efficiently process k-*SB* queries. A salient feature of our query processing algorithm is that it only searches spatial regions that do contain objects satisfying the query Boolean selection criteria.

3. We execute extensive experimentation on an implementation of our methods over large spatial databases. Experimental results show that the proposed methods have excellent performance and scale to multi-million sized datasets compared to alternate methods.

In the rest of the chapter, we separate the presentation of the proposed hybrid indexing approach and query processing algorithms in two sections for clarity. Section 4.2.2 presents the hybrid indexing approach of spatial with textual data while the hybrid index spatial and numeric data is discussed in Section 4.3.3. Experimental study is performed with a prototype search system that implements our hybrid indexes in Sections 4.2.4 and 4.3.5. Section 4.5 discusses related work to our research. Finally, the chapter is summarized in Section 4.6.

[1]In the rest of the chapter, whenever we mention B-tree, we actually mean B$^+$tree unless explicitly stated otherwise.

4.2 Indexing Spatial and Textual Data

4.2.1 Problem Definition

A spatial database $D = \{o_1, o_2, ..., o_N\}$ is a set of objects such that each $o \in D$ has a pair of attributes $< o_p, o_T >$, where: o_p is a two-dimensional point that represents the location of object o, and $o_T = \{t_1, t_2, ...\}$ is a set of terms that provide a textual description of the object. The spatial distance of two objects a and b is given by the function $dist(ap, b_p)$.

A k-NN spatial Boolean query (k-SB) Q is a triple $< Q_l, Q_k, Q_B >$, where: Q_l is a two-dimensional point that represents the query location (spatial constraint), $Q_k > 0$ is the desired query output size, and Q_B is a conjunctive Boolean selection predicate (textual constraint). More specifically, Q_B is a set of terms prefixed with the Boolean operators $\{\wedge, \vee, \neg\}$, and conjunctively connected as follows:

$$B = \Big[\wedge(A = \{a_1, a_2, ...\}) \bigwedge \vee(C = \{c_1, c_2, ...\}) \bigwedge \neg(G = \{g_1, g_2, ...\})\Big] \tag{4.1}$$

Where, the (possibly empty) subsets represent:

A - the AND-semantics subset of terms prefixed with the $\wedge$ (and) connector.

C - the OR-semantics subset of terms prefixed with the $\vee$ (or) connector.

G - the NOT-semantics subset of terms prefixed with the $\neg$ (not) connector.

An object $o \in D$ *satisfies* B if:

$$[(\forall a \in A : o_T \cap a \neq \emptyset) \wedge (\exists c \in C : o_T \cap c \neq \emptyset) \wedge (\forall g \in G : o_T \cap g = \emptyset)] \tag{4.2}$$

The result of the k-SB query Q is the list:

$$L = \{o_i \in D | o_i \; satisfies \; B\}, \text{ such that:} \tag{4.3}$$

$$\forall o \in (D \setminus L) \quad : \quad [dist(o_p, Q_l) \geq arg\,max_{r \in L} dist(r_p, Q_l) \vee \neg(o \; satisfies \; B)]$$

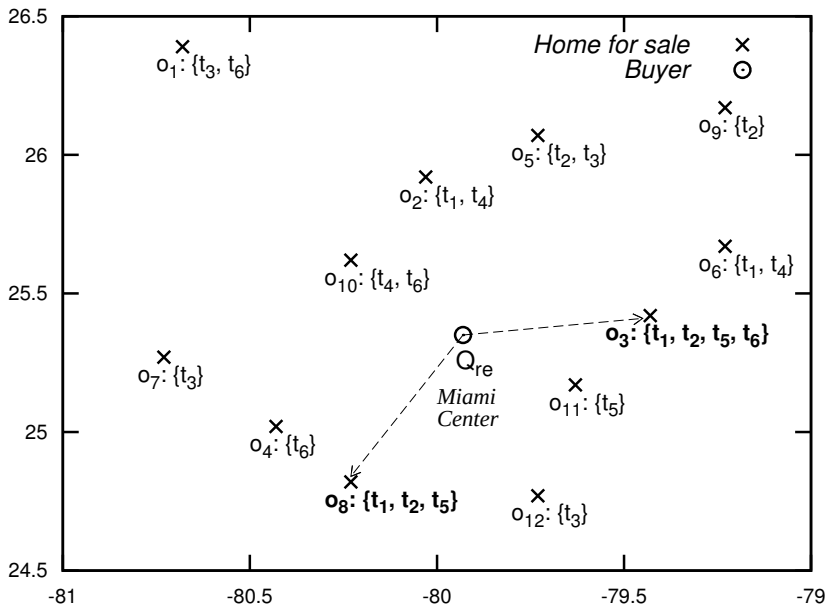


Figure 4.1: k-SB query Q_{re} on the real estate database $D_{re} = \{o_1, o_2, ..., o_{12}\}$. Terms t_i in objects' textual descriptions are shown in Table 4.1 in an inverted file format. The result of the query is $L_{re} = \{o_3, o_8\}$.

Objects in the result set L are sorted by distance to the query location Q_l in non-decreasing order. In other words, a k-SB query Q returns the k-NN objects to the query location Q_l that satisfy the conjunctive Boolean predicate B. We assume the distance function $dist()$ is the Euclidean distance. The problem is how to efficiently compute L.

Example: In the real estate database D_{re} in Figure 4.1 the query:

> "Find *10*-NN houses for sale nearby Miami that have masterbed with bathtub, have a pool or backyard, and are not located in a building"
>
> translates to the following k-SB query:
> $Q_{re} = \{Miami, 10, [\wedge(masterbed, bathtub) \bigwedge \vee(pool, backyard) \bigwedge \neg(building)]\}$
> and retrieves objects $L_{re} = \{o_3, o_8\}$.

Table 4.1: Terms in database D_{re} of Figure 4.1. For every term t_i, the list of objects containing t_i is shown.

Term	**Object List**
backyard (t_1)	$\{o_2, o_3, o_6, o_8\}$
pool (t_2)	$\{o_5, o_8, o_9\}$
building (t_3)	$\{o_1, o_5, o_7, o_{12}\}$
collins (t_4)	$\{o_2, o_6, o_{10}\}$
single-family (t_5)	$\{o_3, o_8, o_{11}\}$
miami (t_6)	$\{o_1, o_3, o_4, o_{10}\}$

In designing the hybrid index, we pursue the following objectives. First, we want to attain fast retrieval even when the matching objects are located far away from the query location. Second, we want to efficiently filter objects not satisfying the query Boolean selection constraints on objects' terms. A key challenge is to perform a small number of computations to eliminate as many non-candidate objects as possible. In particular, the *NOT*-semantics constraints may generate opportunities to substantially shrink the search space, i.e. eliminate spatial regions that contain no candidates; for example, if the *not* terms are frequent in the database, the set of candidate objects is expected to be small. Third, we want to maintain low storage requirements for the index data structure while keeping high query performance.

With the previous objectives in mind, our indexing approach leverages the strengths of R-trees [Gut84] in spatial searches, and organizes textual data in a modified inverted file [ZM06] for efficient processing of Boolean constraints on textual attributes. The combination of indexing techniques yields the hybrid data structure *Spatial-Keyword Index* (*SKI*). We next introduce two important definitions in the *SKI* data structure.

Definition 4.2.1 *Given an R-tree R with node capacity m, a Super Node s is the list of m leaf nodes (level 1) that share the same parent node. Super nodes are identified by unique numbers assigned in a depth-first order visit sequence. The*

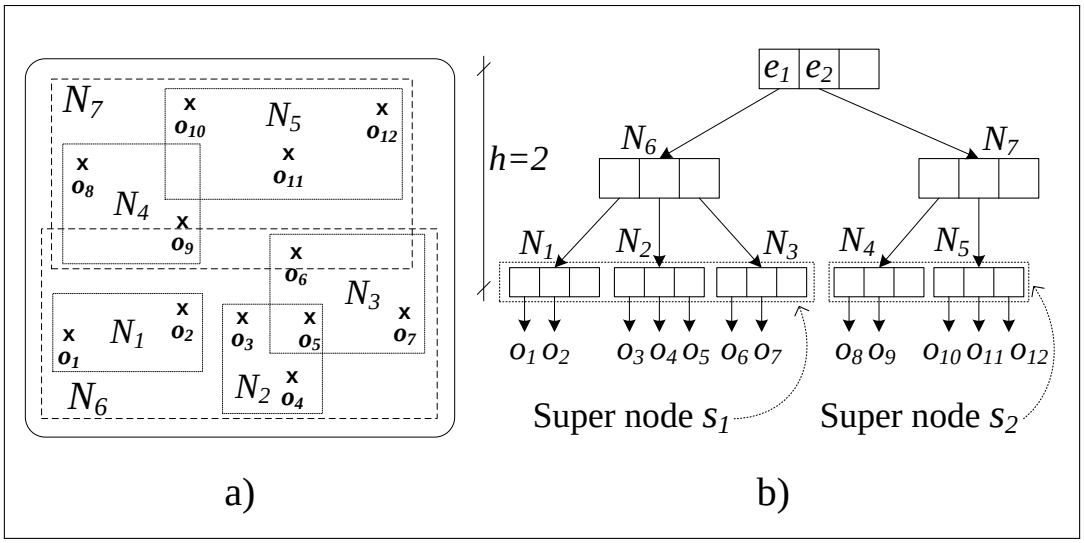


Figure 4.2: a) A sample spatial database. b) Super nodes $[s_1, s_2]$ in the R-tree index of the sample database in (a).

universe of super nodes in a given tree R is $S_R = [s_1, s_2, ...]$, where s_1 is the left most super node in the R-tree.

A *super node* s contains $O(m)$ leaf nodes, or equivalently $O(m^2)$ object pointers. An R-tree of height $h > 0$ has $O(m^{h-1})$ super nodes[2]. (A single-level R-tree has no super nodes.) Figure 4.2b shows an R-tree of height $h = 2$ with node capacity $m = 3$ and its super nodes on the sample spatial database in Figure 4.2a.

Definition 4.2.2 *The bitmap of a term t at super node s is a fixed-length bit array $I(t, s)$ of size m^2, where the i-th bit is computed as follows:*

$$I(t,s)[i] = \begin{cases} 1 & \text{if } s[i] \text{ points to object } o : t \in o_T \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

Figure 4.3 shows the term bitmap for the keyword "miami" at super node s_1 of an R-tree built on the spatial database D_{re} (Figure 4.1).

[2]We consider the height of an R-tree as the number of levels below the root node.

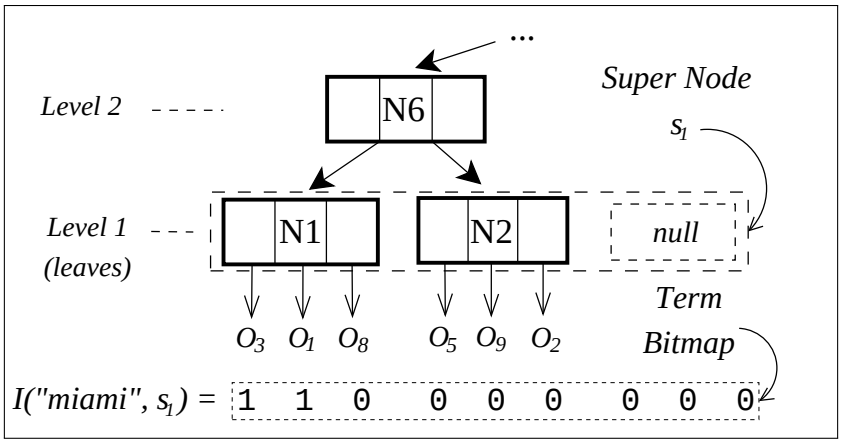


Figure 4.3: Term bitmap for keyword *"miami"* at super node s_1 composed of two leaf nodes $[N_1, N_2]$.

4.2.2 Spatial Keyword Index

The spatial-keyword index (SKI) is composed of two building blocks: a) A modified R-tree index R, and b) A spatial text store T_s, backed up by a B-tree. Figure 4.4 shows the internal data structures of the spatial keyword index. Both tree-based data structures are stored in secondary memory since our objective is to be to index large databases, and we want to avoid hitting main memory capacity limits.

R-tree Index (R)

A modified R-tree built on the spatial attributes of database D. The modification is as follows. Each entry e in inner nodes is augmented with a range $[a, b], a \leq b$ of super node identifiers, where s_a is the left-most super node in the sub-tree pointed by entry e, and similarly s_b is the right-most super node in the sub-tree rooted at e. Ranges in leaf node entries contain a single value, that is, the identifier of the super node that contains the leaf node.

Table 4.2: Key and value definitions of the B-tree that backs up the spatial text store.

	Attribute	Description
Key	term	A term of the database vocabulary.
	s_i	Identifier of super node s_i.
Value	$I(term, s_i)$	Bitmap of "term" at super node s_i.

Spatial Text Store (T_s)

The vocabulary of a database D is defined by $V_D = \{\bigcup_{o \in D} o_T\}$. Terms in the vocabulary and their bitmaps form records that are organized in a text store T_s, which is backed up by a B-tree. Specifically, records are identified by terms and super node identifiers (B-tree keys) in the R-tree R, which link terms to where they occur in the spatial dimension. Keys in the B-tree as well as their values are shown in Table 4.2.

The selection of a B-tree to organize the textual data of the database is justified by the way we intend to retrieve values. B-trees guarantee logarithmic time $O(\log n)$, where n is the number of records stored in the B-tree, for finding records by key [Com79]. In addition, after finding the B-tree leaf node that contains a key $< term, s_i >$, the next p elements in the sorted sequence (that is, range retrievals) can be found with $O(p)$ additional cost, i.e. $O(\log n + p)$ total cost. Both random and range retrievals are operations that our query algorithm performs, which makes B-trees a good candidate for organizing text-related data.

On the other hand, in order to reduce storage space requirements, we compress term bitmaps $I(t, s_i)$ before storing them in the B-tree. In our experiments we employed the Word-Aligned Hybrid (WAH) bitmap compression method [WOS06]. An advantage of the WAH compression method is that it allows fast bitwise com-

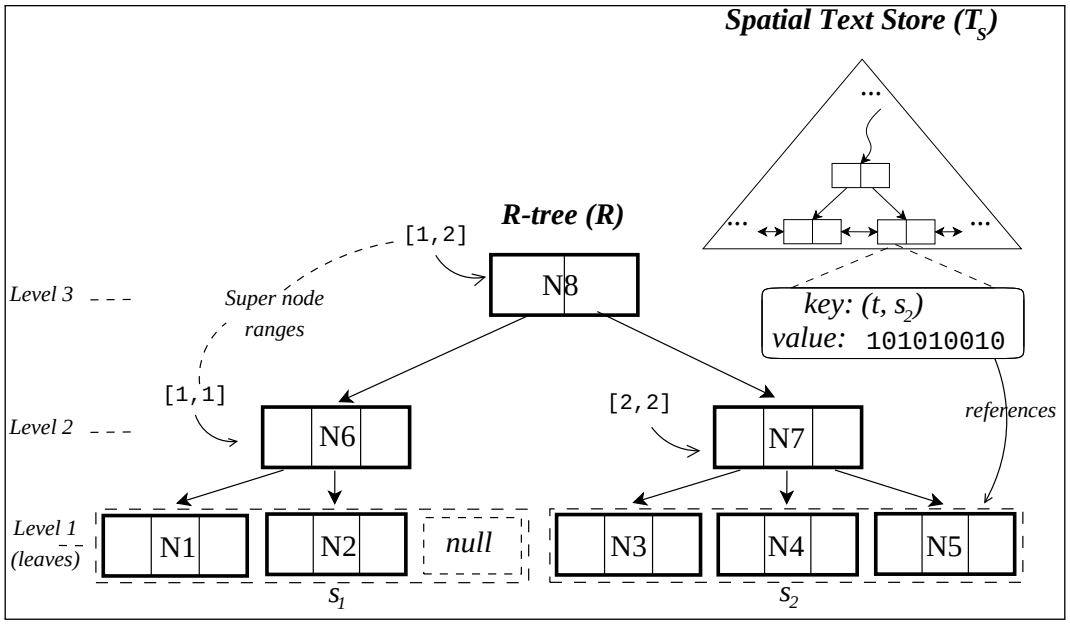


Figure 4.4: Spatial-Keyword Index internal data structures.

putations with logical operators *AND*, *OR*, and *NOT* on uncompressed bitmaps, which help improve efficiency of the query processing algorithm.

4.2.3 Processing k-SB Queries

In order to process a query $Q =< Q_l, Q_k, Q_B >$, the R-tree index R is traversed starting from the root node down to lower levels until the leaf node that contains the nearest object to the query location Q_L is found. Node entries are visited in order of proximity of their minimum bounding rectangles (MBR) to the query location Q_l. However, before traversing a sub-tree rooted at entry e, the text store T_s is used to determine whether there exists at least one element within such sub-tree that satisfies the Boolean selection criteria Q_B. The algorithm retrieves candidate objects using an incremental nearest neighbor approach. That is, after retrieving i objects, the $(i+1)$-th object can be retrieved without re-computing the i previously retrieved objects. Our R-tree traversal algorithm is inspired by the incremental

nearest neighbor strategy proposed by Hjaltason and Samet [HS99], but proper filters are added during R-tree traversal to ensure that no unfruitful traversals are performed. Indeed, a salient feature of our algorithm is that unnecessary sub-tree traversals are eliminated altogether.

Algorithm 4.1 Process k-SB Query

Input: $Q =< Q_l, Q_k, Q_B >$ - A k-SB query.
Output: L - A list of objects with the result of Q (see Equation 4.3).

```
 1: pQueue ← R.root // initialize a priority queue with R's root pointer
 2: L ← ∅ // list of retrieved objects
 3: while (pQueue ≠ ∅ and |L| < Q_k) do
 4:    Entry e ← pQueue.pop()
 5:    if (e points to inner node) then
 6:       if (T_s.isSubtreeCandidate(e, Q_B)) then
 7:          Node n ← R.getNode(e) // retrieve inner node
 8:          for all (Entry e ∈ n) do
 9:             pQueue.push(e, dist(e.MBR, Q_l))
10:          end for
11:       else
12:          // sub-tree pointed by e has no candidates, prune it
13:       end if
14:    else
15:       L.add(getObject(D, e)) // candidate object found
16:    end if
17: end while
18: return L
```

Algorithm 4.1 shows the steps involved in processing *k-SB* queries using the *SKI* data structure. The algorithm starts by initializing a priority queue with the R-tree root pointer in line 1. The priority queue maintains entry nodes sorted by distance to the query location Q_l. An entry e can point to either an R-tree inner node or an object in the database D. In the former case, the entry distance to the query location is computed as the distance from the query location point to the entry's minimum bounding rectangle. In the latter case, the distance is the regular distance between two points, the query location and the object's location. At the head of

the priority queue $pQueue$ is the next nearest entry to the query location, which guarantees that entries are visited in order of proximity to the query location.

The loop between lines 3 and 17 is executed while there are not enough objects in the result list to meet the requested query output size Q_k, or $pQueue$ gets depleated, which means no more potential candidate objects exist in the database. The next nearest entry e is removed from the queue in line 4. In case e points to an internal node, it is evaluated if the sub-tree pointed by e contains at least one candidate object that satisfy the query Boolean selection predicate Q_B. The latter is accomplished by the function call $isSubtreeCandidate()$ (which uses the spatial text store T_s) in line 6. We will explain the logic of this function in the next paragraphs. For now, let us assume that $isSubtreeCandidate()$ return true only if it is worthwhile to traverse the sub-tree rooted at e. When the function $isSubtreeCandidate()$ returns true, the root node n of the sub-tree pointed by e is retrieved from secondary storage. Then, all entries in n are eagerly pushed into the queue even though some of them may point to sub-trees with no candidate objects whatsoever. It would be inefficient, and not necessary, to verify if such entries point to real candidate sub-trees before pushing them into the queue. Instead, the algorithm lazily verifies the sub-tree candidacy condition only when an entry is popped from the queue, i.e. the entry is the next nearest neighbor. When the function $isSubtreeCandidate()$ returns false, it means there is no object that satisfies Q_B, therefore the entire sub-tree pointed by e is pruned.

On the other hand, if the entry e popped off the queue in line 4 points to a database object o, then because of the previous filters o must satisfy Q_B, and it is the next nearest object to the query location. Hence, o is retrieved from the database, and it is added to the result list L in line 15. Finally, the collected candidate objects are returned in line 18.

Algorithm 4.2 *isSubtreeCandidate*

Input: e - An R-tree node entry.
Q_B - A k-SB query Boolean selection criteria.

Output: *true* if there exists $o \in D$ pointed by any super node in e's range $[a, b]$, i.e. $[s_a, s_{a+1}, \ldots, s_b]$, that satisfies Q_B; *false* otherwise.

1: // iterate over the range of super nodes at entry e: $[s_a, s_{a+1}, \ldots, s_b]$
2: **for** ($i = e.a$ to $e.b$) **do**
3: **for** ($j = 1$ to m^2) **do**
4: $b[j] \leftarrow 1$ // initialize a bit array with all bits set
5: **end for**
6: **if** ($Q_B.A \neq \emptyset$) **then**
7: $b \leftarrow b \wedge \left[\bigwedge_{t \in Q_B.A} T_s.getBitmap(t, s_i)\right]$ // *AND*-semantics predicates
8: **end if**
9: **if** ($cardinality(b) > 0$ and $Q_B.C \neq \emptyset$) **then**
10: $b \leftarrow b \wedge \left[\bigvee_{t \in Q_B.C} T_s.getBitmap(t, s_i)\right]$ // *OR*-semantics predicates
11: **end if**
12: **if** ($cardinality(b) > 0$ and $Q_B.G \neq \emptyset$) **then**
13: $b \leftarrow b \wedge \left[\bigwedge_{t \in Q_B.G} flip(T_s.getBitmap(t, s_i))\right]$ // *NOT*-semantics predicates
14: **end if**
15: **if** ($cardinality(b) > 0$) **then**
16: **if** (e points to inner node) **then**
17: **return true**
18: **else if** ($\exists j : b[j] = 1$ and $s_i[j] = e$) **then**
19: **return true**
20: **end if**
21: **end if**
22: **end for**
23: **return false**

Algorithm 4.2 describes the logic of the function *isSubtreeCandidate* executed during the processing of a k-SB query. Intuitively, this function determines if there is at least one super node s_i, in the range of possible super nodes $[s_a, \ldots, s_b]$ within the sub-tree rooted at e, that contains an object o satisfying the k-SB Boolean criteria.

The algorithm sequentially iterates over the range of super node identifiers between lines 2 and 22. For each super node $s_i \in [s_a, \ldots, s_b]$, term bitmaps of query terms in every predicate subset are combined according to the subset semantics. That is, bitmaps of terms in the *AND*-semantics subset are combined with the *and* bitwise operator; similar operation is done with bitmaps of terms in the *OR*-semantics subset using the *or* operand. However, for terms in the *NOT*-semantics subset, their bitmaps need to be first flipped (to implement the negation logic) before combining them, which is accomplished by the function $flip()$ in line 13. Intermediate bitwise operation results are stored in the bit array b, whose bits are initially set to 1 in line 4. At every time in the outer-most **for loop** (lines 2-22), if $b[j]$ is set to 1, it means that the object pointed by $s_i[j]$ is a potential object candidate to satisfy the k-SB selection predicates Q_B. In line 15, the algorithm checks the number of bits in b that remain 1, using the function $cardinality()$. If a bit $b[j]$ (and possible more) has 1 value, then it means that the object pointed by $s_i[j]$ definitely satisfies the k-SB Boolean selection predicate, thus the algorithm returns *true* in line 17. In the particular case when e points to a database object, the algorithms needs to additionally check that there is a super node entry $s_i[j]$ that points to the same object for some set bit $b[j]$. Otherwise, if all bits in b are off, super node s_i points to no object that satisfy Q_B, so the next super node s_{i+1} is checked. If the outer-most **for loop** is exhausted, then no candidate object pointed by the range of super nodes $[s_a, \ldots, s_b]$ satisfy the query selection predicates. In that case, the

algorithm return *false* in line 23.

Since the algorithm *isSubtreeCandidate*() is called multiple times during a query execution, starting from node entries close to the root node and down towards leaf nodes, there is potentially several redundant bitwise computations that could be performed. For instance, if a first call to *isSubtreeCandidate*() determines that in the super node range $[s_1 \dots s_4]$, s_3 has candidate objects. A second call to *isSubtreeCandidate*() may test the range $[s_2, s_3]$ which we already know that s_3 does contain a candidate object. To avoid such redundant computations, we store the super nodes s_i that do contain qualifying objects as keys in a hash table, and their respective resulted bit sequences b_i as the values associated to keys. In that way, the algorithm first checks if there is already a qualifying super node in the hash table before evaluating a range of super nodes. Similarly, following the same example, from the first call to *isSubtreeCandidate*() we know that super nodes s_1 and s_2 do not contain any candidate object. To avoid re-evaluating those super nodes in subsequent calls to *isSubtreeCandidate*(), we store non-candidate super nodes s_j in a hash set.

Time and Space Complexity

The worst case for *isSubtreeCandidate*() is when the **for loop** between lines 2 and 22 is execute over the entire super node range $[s_a, \dots, s_b]$. Since a super node contains $O(m^2)$ object pointers, where m if the R-tree fanout, a call to the function *isSubtreeCandidate*() can potentially prune $O(m^2 \times [b - a + 1])$ objects, when no candidate object exist in the super node range $[s_a, \dots, s_b]$. Considering that m is usually in the order of hundreds, the number of pruned objects can be relatively large. The cost of worst case *isSubtreeCandidate*() is remarkably low $O(|Q_B| \times [\log n + (b - a)])$, where $|Q_B|$ is the number of query terms, and n is the number

of keys stored in the B-tree. The first $key =< t, s_a >$ is retrieved in time $O(\log n)$ while subsequent keys $< t, s_{a+1} >, \ldots, < t, s_b >$ are retrieved with $O(b - a)$ cost.

The variable n (number of keys stored in the B-tree) in our analysis depends on the database vocabulary size $|V|$. In the worst case, every $term \in V$ is referenced by all the super nodes in the R-tree, which means that the distribution of terms in space is close to uniform. For the worst case, assuming an R-tree of height $h > 0$, the number n of keys in the B-tree is bounded by $O(|V| \times m^{h-1})$, where $O(m^{h-1})$ is the maximum number of super nodes in the R-tree of height h. In practice, there are location-related terms that generally occur in a constrained spatial area, like place names or street names, e.g. *Miami Downtown*. There are also generic terms, no related to any particular place, so they may appear almost anywhere on the map, such as *Avenue* or *Condominium*. On the other hand, the empirical Zipf's law suggest that the frequency of terms in a vocabulary generated from a natural language can be approximated by a Zipfian distribution – a power law probability distribution. In particular, the law says that the frequency of a term is inversely proportional to its rank in the ranking table. Thus, we can expect that a few percent of terms in the database vocabulary be highly frequent while the majority of terms would tend to appear in a few objects, thus somewhat restricted to the objects' geolocations. Because of the previous arguments, we can reasonably expect that the number n of keys in the B-tree be much less than its worst case.

4.2.4 Experimental Evaluation

We conducted a series of query performance experiments with a prototype search system implemented in Java. The search system implements both the proposed k-SB query processing algorithms, and the spatial-keyword index defined in Section 4.2.2.

Table 4.3: Spatial databases used in experiments. Database and vocabulary sizes are in millions.

Database	$\|D\|$	$\|V\|$	Description
FL	10.8	21.2	Property parcels in the Florida state.
YP	20.4	40.8	Yellow pages of businesses in the United States.
RD	23.0	64.8	Road segments in the United States.

The R-tree was implemented in Java, and we used the open source library JDBM to support the B-tree data structure[3]. Experiments were run on an Intel Xeon E7340 2.4GHz machine with 8GB of RAM, and a single disk attached directly to the host.

We used three real spatial datasets in our experimentation. The datasets are listed in Table 4.3. Objects' locations are determined by geographical coordinates in latitude and longitude format. In the case of the road segments database (RD), although segments are lines defined by a pair of points on the map, only one point was used to determine the geolocation of the segment. Databases contain between 30 and 80 text attributes, which were concatenated in a term set.

We measured two performance metrics on the execution of k-SB queries: 1) Average number of random I/Os, and 2) Query execution elapsed times. The measured metric include only access to the index data structures. Actual record retrieval is not included in the metrics. We compared the performance metrics of our query processing technique against two baselines.

Baseline 1 (*IFC*)

This is a regular inverted file of terms with posting lists containing a sequence of objects that contain the term. In addition, objects in postings are augmented with their geolocation coordinates to be able to compute spatial distances without

[3]JDBM is an open source library that provides scalable data structures, such as Hash table and B+tree, to support persistence of large object collections.

retrieving the object location from the database. Queries were processed in two phases. First, term posting lists are merged according to the semantics of each subset in the query Boolean predicate Q_B. Merging means that postings of *AND*-semantics terms are intersected, postings of *OR*-semantics terms are combined, and finally postings of *NOT*-semantics terms are substracted from the previous result. In a second phase, objects that passed the selection criteria are sorted by distance in increasing order to the query location. From the sorted list, the k nearest neighbors are returned as the result of the query.

Baseline 2 (*RIF*)

An R-tree index was built on the input database. Every node in the tree was augmented with an inverted index on the keywords found within the sub-trees rooted at that node. The difference with the *IFC* inverted files is that for a given R-tree node, term posting lists contain reference to node entries where the term occurs. This baseline is inspired by works in [CJW09] [HHLM07]. At query time, R-tree node entries are visited in nearest-first order on the basis of the distance to the query location. Posting lists of terms in the *AND*-semantics and *OR*-semantics subsets of the query predicate Q_B are merged at every R-tree node to determine which entries should be visited. Note that terms in the *NOT*-semantics cannot be applied on internal nodes because the inverted file of a term t only indicates which subtrees contain t, but there may be other objects within those subtrees that still satisfy the query predicates.

Query Workload

For each databases in the experiments, terms in its vocabulary were sorted by document frequency (*df*) in increasing order. Then the 3-quantiles were selected from

the sorted lists such that the list was divided in three equal-sized groups.

- **S** - Terms with $df < 1$–quantile (infrequent terms),
- **M** - Terms with $df < 2$–quantile,
- **L** - Terms with $df < 3$–quantile (entire vocabulary).

From each quantile, a *k-SB* query Q was composed by randomly picking between three and eight terms from the quantile group $\{S, M, L\}$. Terms were randomly included in either the *AND*-semantics, *OR*-semantics, or *NOT*-semantics subset to form the query selection predicate Q_B. The size of the output was fixed to $k = 20$. For each database, we generated 50 k-SB queries for each type of workload and executed them with our query processing method (*SKI*) and the two baseline methods.

Discussion

Figure 4.5 shows the average number of *I/O* reads executed by each query processing method over the 50 k-SB generated queries for each workload type $\{S, M, L\}$.

We observe in Figure 4.5a and Figure 4.5b that *IFC* has performance advantages over *SKI* and *RIF* methods when query terms are relatively infrequent – workloads S and M. For infrequent terms, in the S group, it is expected that their posting lists be relatively small, which can be quickly retrieved and evaluated to compute the query result. Even when terms are slightly more frequent, like in M workloads, shorter posting lists are read from secondary storage in their entirety sooner than longer postings, that may not be retrieved completely, by *IFC*. So, the merge phase can be still executed fast enough, as it can be observed in Figure 4.5b. On the contrary, *SKI* and *RIF* methods need to perform additional work in traversing R-tree nodes to locate objects in nearest neighbor order.

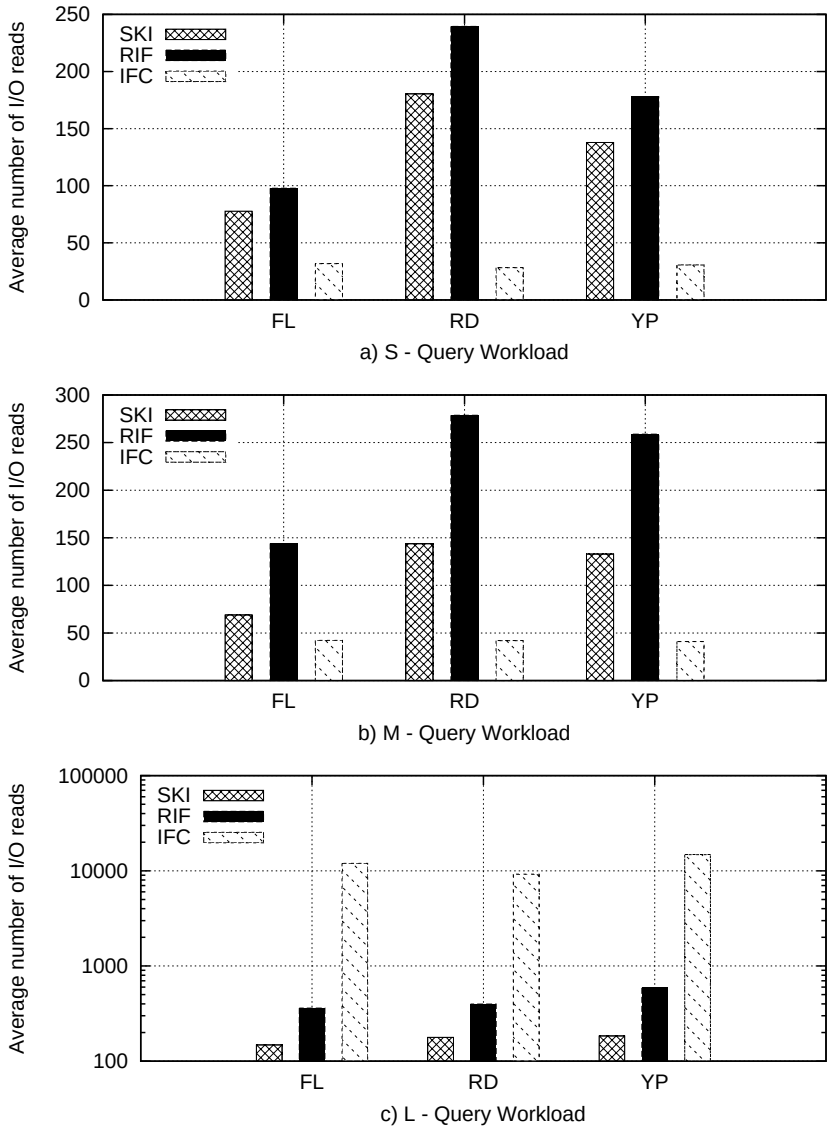


Figure 4.5: Average random *I/O* reads of 50 randomly generated k-SB queries. *Y-axis* in (c) is in logarithmic scale.

For S workloads, *SKI* and *RIF* execute comparable amount of work while still performing acceptably in terms of *I/O* cost. However, there are noticeable performance gains of *SKI* over *RIF* in M workloads. One reason for such difference is that *RIF* does not implement the NOT-semantics filter in upper level nodes of the R-tree while *SKI* applies all filters before visiting every sub-tree. In addition, it may happen in *RIF* that after reaching a leaf node level, the actual combination of query terms is not satisfied by any leaf entry. Eventually, sub-trees known (via inverted file) to contain the query terms in *RIF* are engaged in query processing. However, no single object within the sub-tree satisfy the query predicates. For instance, in an internal node, its inverted file determines that terms t_x and t_y are located within the sub-tree pointed by entry e, but at the leaf node terms occur in two different objects, $t_x \in x_T$ and $t_y \in y_T$, which makes the sub-tree traversal unfruitful and degrades *RIF*'s performance.

On the other hand, when query terms become more frequent, like in workloads M and L, IFC incurrs in expensive long posting list retrievals while performing their merging process; Figure 4.5c show high peaks for the *IFC* method as a consequence of such expensive operations. In contrast, *SKI* performs significantly better than both *RIF* and *IFC* in L workloads. The main reason for *SKI* performance advantages is twofold. First, it has the ability to prune non-candidate objects at larger granules (super node) in logarithmic time. Second, false drops are avoided altogether by guaranteeing that a sub-tree in the R-tree index contain at least one object that satisfy the query predicates before engaging in traversing the sub-tree.

Figure 4.6 shows elapsed times during query execution of the three methods. Some discrepancies with the *I/O* plots can be observed in the time plots. For example, Figure 4.6c shows substantially longer times than expected for *RIF* on S workloads over the *FL* dataset. This might be caused by overheads introduced by

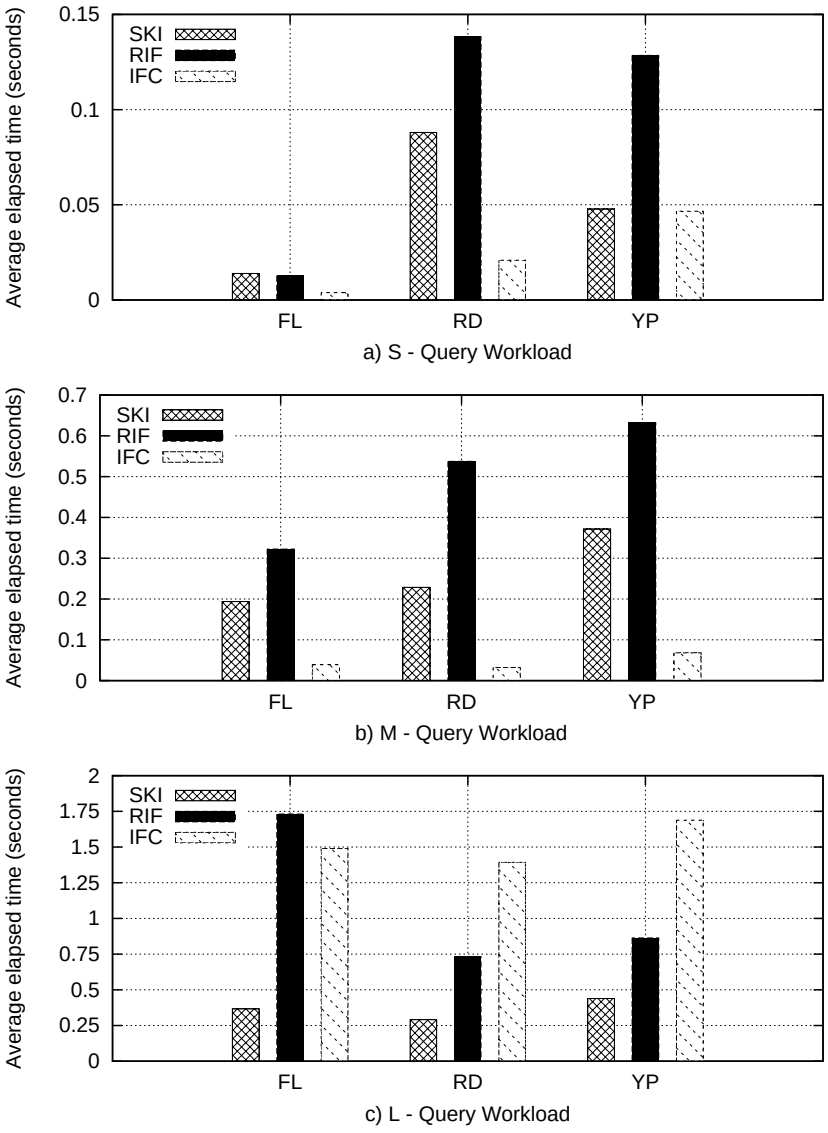


Figure 4.6: Average elapsed time of 50 randomly generated k-SB queries.

the operating system at the time we ran *RIF* experiments. Also, in Figure 4.6c, *IFC* show unexpectedly lower times compared to the amount of *I/O* work done. That might be the effect of posting lists being stored sequentially on secondary storage. During query processing, posting lists are retrieved by blocks (which contain several object identifiers) in sequence starting from the first block. So, blocks may be read semi-sequentially, reducing the total amount of disk seek time. In general for all methods, another factor that helps reduce the elapsed time is the operating system cache. In tree-based data structures, nodes in upper level are retrieved frequently, which makes them more likely to be cached. So, subsequent query executions may experience faster *I/O* reads. In summary, we observed consistent enhanced retrieval performance using the proposed hybrid spatial indexing and query processing methods.

4.3 Indexing Spatial and Numeric Data

Attributes of numeric type are common in spatial databases. Thus, it is desirable to include conditions on numeric data in spatial queries. For example, in the real estate database in Figure 4.7, a buyer may post the following k-NN query:

"*Find the nearest homes for sale with prices between $50,000 and $80,000*"

Existing works on spatial queries with keyword constraints [ABL10] [CJW09] [DFHR08] [HHLM07] [PK03], including ours [CWR10], cannot be directly applied on numeric data because they do not properly capture numeric semantics. A spatial query with a numeric range constraint "$price \leq v$", searches for no particular *price* value, like in the case of keyword matching, but can be satisfied by multiple objects so long their *price* values are at most v. A basic query processing method would retrieve all the possible objects that satisfy the numeric condition, and then apply

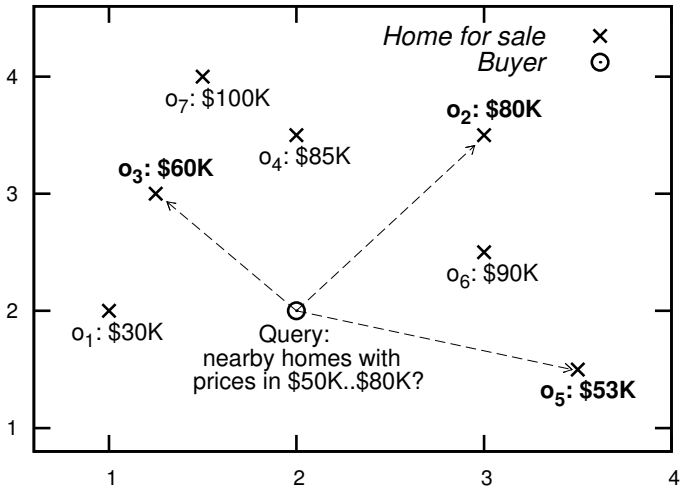


Figure 4.7: The result of a k-NN query of homes for sale with prices in the range $\$50,000..\$80,000$ is the sorted list $[o_3, o_2, o_5]$.

the required spatial relation, e.g. nearest neighbor, which can result in a very expensive operation.

The efficient execution of spatial searches with numeric constraints poses a few challenges towards achieving scalability. First, the number of objects in geospatial databases is constantly growing. It is common for today's geospatial databases to store tens of millions of objects. Second, the number of individual numeric fields stored in a database can be large; some applications may need to store a few hundreds of numeric attributes. Third, numeric attributes may have different precision requirements; integers may be adequate for storing people's ages, while decimal numbers could be required for prices and land areas. Finally, in contrast to textual data that usually comes from a (natural language) vocabulary of finite size, the amount of unique numeric values in a database may be arbitrarily large.

In this section, we focus on efficiently processing k-nearest-neighbor (k-NN) spatial queries with range conditions on numeric fields. Our approach tackles the aforementioned challenges in the following ways Numeric values are encoded into a com-

pact, byte string representation using an Interval-based tree compression method [Ris92]. A key insight of the numeric compression method is that prefixes of encoded number strings adequately represent numeric ranges that enclose the encoded number. On the other hand, geographical coordinates of database records are indexed in an R-tree [Gut84] (with a similar modification that was discussed in Section 4.2.2). Encoded numbers and encoded numeric ranges are linked to R-tree nodes where the numbers occur, and organized in a B-tree [Com79]. Both the R-tree and B-tree data structures are stored on secondary storage. Thus, our approach is not limited by the available main memory of the host system, which makes it suitable for large databases. The query processing algorithm traverses the R-tree following an incremental nearest neighbor approach [HS99] and leverages B-tree's efficient range retrieval operations for pruning R-tree branches that do not lead to candidate objects.

4.3.1 Problem Definition

We define a geospatial database $D = \{o_1, o_2, \ldots, o_N\}$ as a set of N objects. Without loss of generality, we assume that an object $o \in D$ has three fields $< o_{id}, o_p, o_v >$, where: o_{id} is the object's unique identifier, o_p is a two-dimensional point representing the object's geolocation, and o_v is a numeric attribute representing some quantity. Intuitively, a k nearest neighbor query with a user-defined numeric range constraint retrieves from database D the k nearest objects to a reference point, whose numeric values are within the requested numeric range.

Formally, a k-NN query Q with a numeric constraint (k-SBn) is defined by the tuple $Q =< Q_p, Q_k, [Q_l..Q_u] >$, where[4]:

[4]To distinguish a k-NN query with numeric constraints from the similar query with text constraints, defined in Section 4.2.1, we suffix the short name k-SB with the letter

Q_p - is the query's 2D reference point.

$Q_k > 0$ - is the number of nearest objects to retrieve.

$[Q_l..Q_u]$ - are the lower (Q_l) and upper (Q_u) bounds of the numeric range constraint. Either Q_l or Q_u can be absent, to represent open intervals $[Q_l\ ..\ +\infty)$ and $(-\infty\ ..\ Q_u]$, but not both.

The result of the query Q is a list of objects $L = [r_1, r_2, \ldots, r_{Q_k}], r_i \in D$ sorted by distance to the query reference point $dist(r_{i_p}, Q_p)$ in non-decreasing order such that $\forall r \in L : Q_l \leq r_v \leq Q_u$. We adopt the Euclidean metric as distance $dist(p_1, p_2)$ between two 2D points. The problem is how to efficiently compute L in a reasonable amount of time suitable for interactive search systems.

4.3.2 Numeric Data Encoding

As part of our data indexing approach, we encode numeric data using the compression method described in [Ris92]. The encoding method uses a tree of intervals in which each node has exactly 128 children. The root node is the open interval $(-\infty, \infty)$ of real numbers. The root interval is divided into 128 non-overlapping sub-intervals $[a_i, a_{i+1}), i = 0, \ldots, 127$, closed on the left and open on the right, except for the first sub-interval. (Different methods can be used for sub-dividing intervals; for example, an arithmetic partitioning sub-divides an interval in 128 equal-sized sub-intervals. A suggested set of partitioning methods suitable for databases is found in [Ris92], which we also use during our experimentation.) Each child interval is recursively sub-divided into 128 sub-intervals. The tree of intervals is infinite, but the numbers in a database are represented by a concrete tree instance with finite height.

"n", as in k-SBn

For a number v, the encoding algorithm starts traversing the tree of intervals from the root by choosing the sub-interval where v is contained. Then tree branch of the chosen sub-interval is recursively visited to identify which child sub-interval contains v. The algorithm continues in that fashion until a sub-interval is found such that its left boundary is equal to v, i.e. the sub-interval is defined by $[v, v')$. Then, the encoding of the number v is represented by the sequence of sub-interval numbers visited starting from the root until the leaf sub-interval where the algorithm stopped. A sub-interval number is stored in a *byte* (of 8 bits) as follows. The seven most significant bits contain the interval number while the remaining bit indicates if more bytes follow. The least significant bit is used by the decoding algorithm. A "1" in the least significant bit indicates that the encoding continues in the next byte while "0" means that the encoded representation is complete, i.e. the sub-interval $[v, v')$ has been reached, hence the decoded value v is known.

Figure 4.8 shows the tree sub-intervals visited when encoding the number "26.1275". The resulting encoding includes the intervals 28 (root), 30 and 93 (leaf), which are stored as the byte sequence $[00111001, 00111101, 10111010]$, or equivalently in decimal numbers $[57, 61, 186]$. In general we denote the byte encoding of value v as $E(v) = [b_1, b_2, \ldots, b_e]$. A value v is encoded in time $O(\log_2 128 \times e)$, where e is the number of bytes in the encoded byte array.

The described interval-based encoding method has three main characteristics that make it attractive for tackling the scalability and performance issues of k-SBn queries. First, it provides a uniform byte-string representation of numbers, regardless of their precision (integer, decimal). Second, the lexicographic ordering of encoded strings matches with the ordering of numeric values[5]. Third, assuming adequate interval partitioning schemes, the encoding method yields space-efficient

[5]A formal proof can be found in [Ris92].

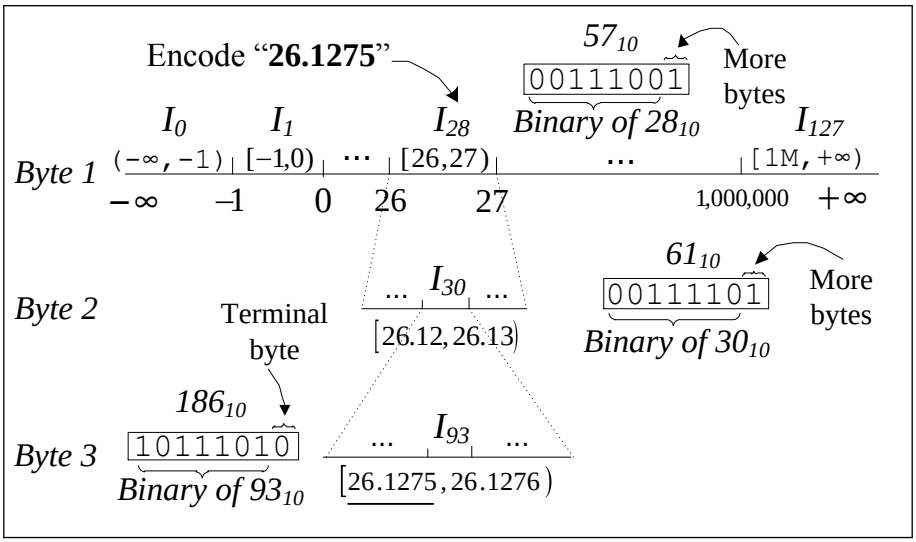


Figure 4.8: Tree of intervals generated during the encoding of number "26.1275".

byte representations for numeric values frequently encountered in databases. Such characteristics allow us to store encoded numbers in data structures that support efficient key range retrieval, like the ubiquitous B-trees [Com79], which we do as part of our indexing technique in Section 4.3.3, while still maitaining low storage consumption.

Furthermore, an interesting insight of the interval tree is that partial interval sequences from the root to intermediate sub-intervals adequately capture numeric ranges. That is, byte string prefixes of an encoded number represent any number in the range defined by the prefix intervals. For example, in Figure 4.8, the byte prefixes $[00111001, 00111101]$ and $[00111001]$ would be shared by any encoded number in the ranges $[26.12, 26.13)$ and $[26, 27)$, respectively. Such insight is capitalized by our indexing and query processing methods described in Sections 4.3.3 and 4.3.4, respectively.

4.3.3 Spatial Number Index

Indexing locational and numeric data requires a combination of adequate indexes for each data type. We leverage the hybrid data structure that we proposed in Section 4.2.2 for indexing locational and textual data simultaneously. That is, spatial attributes in the database are organized in a modified R-tree. However, it would not be efficient to store numeric data directly in the spatial text store T_s for two main reasons. First, it may take substantially large store space to store all possible numbers in the database. Since some numbers tend to be frequent, like some integer numbers (e.g., the number of bedrooms attribute in a real estate database may contain integers from a small range), there are opportunities to compress numeric data to alleviate storage requirements. Second, processing k-SB queries with numeric range constraints may take excessive amount of time, especially when the query range intersects with a large percentage of numbers in the database. For instance, if the query in Figure 4.7 had instead the range constraints "$price \geq 0$", most homes for sale are potentially candidates, which would make our k-SB query processing algorithm retrieve many R-tree and B-tree nodes.

To address the aforementioned problems in storing raw numeric data, numbers in the database are encoded using the compression method described in Section 4.3.2. Additionally, numeric ranges are pre-computed on the basis of encoded numbers. Finally, encoded numbers and encoded ranges are stored in a *Spatial Number Store* (N_s), backed up by a B-tree, similar to the *Spatial Text Store* (T_s) defined in the *SKI* index.

The systematic combination of R-trees and B-trees yields the *spatial-number index* (*SNI*). The *SNI* 's underlying tree-based data structures – R-tree and B-tree – are constructed in sequence, similarly to the *SKI* index in Section 4.2.2. However, the *SKI*'s *Spatial Text Store* (T_s) is replaced by a *Spatial Number Store* (N_s). The

characteristics of N_s are discussed next in the section.

Encoding Numbers and Ranges

Numbers in the database are first encoded using the interval-based numeric data encoding described in Section 4.3.2. The result is a set of compact byte arrays, one for each numeric value v in the database, in the format $E(v) = [b_1, b_2, \ldots, b_e]$.

An interesting insight of the numeric encoding algorithm is that prefixes of the byte array of an encoded number adequately capture the semantics of numeric ranges. To illustrate this idea, let us consider the encoding of two numbers v_1 and v_2 contained in the top sub-interval $[a_i, a_{i+1}($, where their leaf sub-intervals (the last byte of their encodings) are j and k, respectively, i.e. $E(v_1) = [\#i, \ldots, \#j]$ and $E(v_2) = [\#i, \ldots, \#k]$. The lowest common ancestor of j and k is a sub-interval $[a'_l, a'_{l+1}($ that contains both numbers v_1 and v_2. For example, in Figure 4.8 the encodings "$E(26.1275) = [\#28, \#30, \#93]$" and "$E(26.1276) = [\#28, \#30, \#94]$" have sub-interval "$\#30 = [26.12, 26.13($" as lowest common ancestor, and clearly both numbers 26.1275 and 26.1276 are included in that sub-interval. By construction of the tree of intervals, all upper level sub-intervals that contain the lowest common ancestor until the top sub-interval $[a_i, a_{i+1}($ do also contain the original values v_1 and v_2. In other words, the prefixes of a byte array encoding $E(v)$ can represent numeric ranges that contain v. Furthermore, such prefixes can also be encoded as if they were numbers by turning off the terminal bit in the last byte of the prefix arrays.

With that observation in mind, encoded numeric ranges are generated on the basis of the already encoded numbers by extracting prefixes of their encoded byte arrays. For every encoded byte array $E(v) = [b_1, b_2, \ldots, b_e]$, a single pass on the array can be made to generate all its prefixes: $\{[b_1], [b_1, b_2], \ldots, [b_1, b_2, \ldots, b_{e-1}]\}$.

Algorithm 4.3 Derive Numeric Ranges

Input: $E(v) = [b_1, b_2, \ldots, b_e]$ - a byte array representing the encoded value v.
Output: A list of byte arrays representing all the upper level sub-intervals of $E(v)$ in encoded format.

```
1: encodedRanges ← ∅ // initialize a list of byte arrays
2: for (i = 1 to e) do
3:    range ← subString(E(v), 1, i) // get prefix up to the i-th byte
4:    range[i] ← and(range[i], "11111110") // make last byte terminal
5:    encodedRanges.add(range)
6: end for
7: return encodedRanges
```

Algorithm 4.3 shows the steps involved in deriving numeric ranges from a given number encoding $E(v)$. As mentioned, to conform to the numeric encoding scheme, the last byte of encoded ranges is made terminal. The latter is accomplished by the *and* bitwise operation in line 4. For a given number v, the total number of number encodings, including v and all its numeric ranges (i.e., the array prefixes generated by the Algorithm 4.3), is $O(e^2)$.[6]

Number Bitmaps

We consider a similar concept to the term bitmaps, as defined in Definition 4.2.2, in the *SKI* text store T_s, but for numbers.

Definition 4.3.1 *The bitmap of a number v at super node s is a fixed-length bit array $I(v, s)$ of size m^2, where the i-th bit is computed as follows:*

$$I(v,s)[i] = \begin{cases} 1 & \text{if } s[i] \text{ points to object } o : o_v = v \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

[6]This might be a large number of encodings when e is large. However, in the experimental study, we observed that more than 80% of the numbers were encoded with $e = 3$ bytes or less.

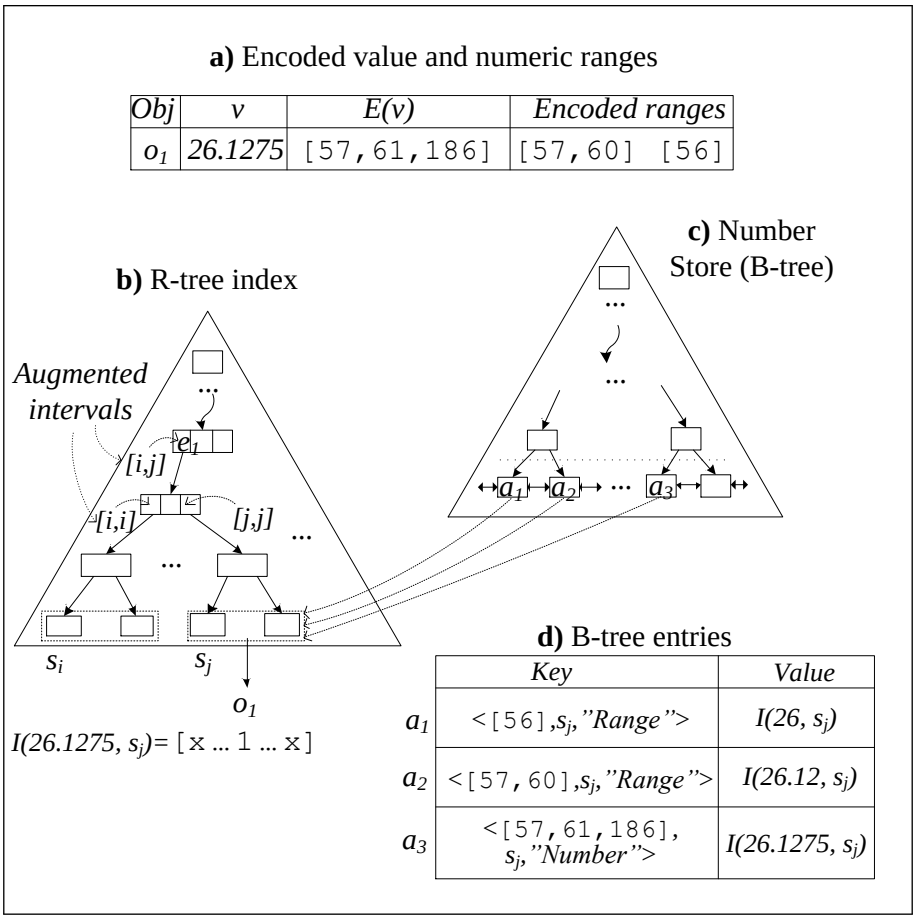


Figure 4.9: Underlying data structures in the Spatial-Number Index (SNI).

Spatial Number Store (N_S)

Encoded numeric values v and their encoded ranges are computed for numeric data in the database D. for every number v in the dataset Then encoded numbers and their number bitmaps are organized in *Spatial Number Store* (N_s), which is backed up by a B-tree. The definition of the keys and values in the B-tree are shown in Table 4.4. Since keys contain both numbers and ranges, an additional field is required to indicate if the encoded byte array $E(v)$ comes from a number in the database ("*Number*"), or if it was the result of numeric range generations ("*Range*") based on other value.

Table 4.4: Key and value definitions of the B-tree that backs up the spatial number store.

	Attribute	Description
Key	$E(v)$	Encoded byte array of value v.
	s_i	Identifier of super node s_i.
	type	Type of value: {"*Number*", "*Range*"}.
Value	$I(v, s_i)$	Number bitmap of value v at super node s_i.

Figure 4.9 shows the underlying data structures in the *SNI* index. In particular, Figure 4.9d shows the B-tree entries for the example encoded value and its numeric ranges in Figure 4.9a. The R-tree data structure is identical to the one in the *SKI* index, i.e. nodes are augmented with ranges of super node identifiers as can be seen in Figure 4.9b.

4.3.4 Processing k-SBn Queries

The query processing algorithm is similar to the process described in Algorithm 4.1 for processing k-SB queries with text constraints. That is, qiven a query Q, the R-tree data structure is traversed starting from its root, and entries are placed in a priority queue ordered by their distances to the query location Q_p. The main difference is in the function call to *isSubtreeCandidate* (Algorithm 4.2), which in this case uses the spatial number store N_s to evaluate the candidacy of an R-tree branch. Before traversing a sub-tree pointed by an entry e, we check in the spatial number store N_s if there is at least one encoded number or range that intersects with the query range constraint $[Q_l..Q_u]$. This logic is implemented in an analogous function, called $isSubtreeCandidate_n$, and defined in Algorithm 4.4.

Algorithm 4.4 first identifies the range of B-tree keys $[min_{key}, max_{key}]$ it needs to search for candidate objects between lines 3 and 10. Note that min_{key} and max_{key}

Algorithm 4.4 $isSubtreeCandidate_n$

Input: e - An R-tree node entry.
Q_l, Q_u - Encoded byte arrays representing the lower (Q_l) and upper (Q_u) bounds of the query range constraint.

Output: *true* if there exists $o \in D$ pointed by any super node in e's range $[a, b]$, i.e. $[s_a, s_{a+1}, \ldots, s_b]$, such that $Q_l \leq o_v \leq Q_u$; *false* otherwise.

1: $top_l \leftarrow Q_l[1] >> 1$ // get top sub-intervals
2: $top_u \leftarrow Q_u[1] >> 1$
3: **if** $(top_l < top_u)$ **then**
4: $\quad min_{key} \leftarrow < encode(top_l), e_a, \text{“}Number\text{”} >$
5: $\quad max_{key} \leftarrow < encode(top_u), e_b, \text{“}Number\text{”} >$
6: **else**
7: $\quad lca \leftarrow lowestCommonAncestor(Q_l, Q_u)$
8: $\quad min_{key} \leftarrow < encode(lca + Q_l[lca.size]), e_a, \text{“}Number\text{”} >$
9: $\quad max_{key} \leftarrow < encode(lca + Q_u[lca.size]), e_b, \text{“}Number\text{”} >$
10: **end if**
11: $iter \leftarrow N_s.getEqualOrGreater(min_{key})$ // finds min_{key} or greater in N_s
12: $key \leftarrow iter.getNext$
13: // iterate over the range of keys $[min_{key}, max_{key}]$
14: **while** $(key \neq \emptyset$ and $key \leq max_{key})$ **do**
15: $\quad$ **if** $(key.s \in [e_a, e_b])$ **then**
16: $\quad\quad$ **return true**
17: $\quad$ **else**
18: $\quad\quad key \leftarrow iter.getNext$
19: $\quad$ **end if**
20: **end while**
21: **return false**

contain encoded sub-intervals, at one particular level of the sub-interval tree, that are the most proximal to the root interval. This makes the algorithm efficient. Even if the database contains a large number of values distributed in a small range, only one upper level sub-interval will be searched for – the lowest common ancestor sub-interval. Once the proper range of minimum and maximum keys is defined, the algorithm traverse sequentially all the keys in range to find a candidate object, i.e. a super node identifier in the input R-tree entry range $[e_a, e_b]$.

Time Analysis

Every interval in the tree of intervals is sub-divided in 128 sub-intervals. Since the keys in the B-tree are associate to super node identifiers, a maximum of $128 \times (b - a + 1)$ keys will be retrieved sequentially in the *while* loop of lines 14-20. To be more precise, every *interval/supernode* pair may be of up to two possible types: "Number" and "Range". Hence, the worst case time complexity of the algorithm is $O(256 \times (b - a))$. Generally, much fewer keys are retrieved, especially for numeric intervals that have sparse distribution.

4.3.5 Experimental Evaluation

Setup and Baseline

Setup

We ran the experiments in a host machine running on Linux CentOS release 5.6 operating system with 16GB of RAM. Indexes were stored in a 3-disk RAID-5 storage array, directly attached to the host. At the time of the experiments, no other outstanding processes were running in the host machine.

Table 4.5: Spatial databases used in k-SBn query experiments

Dataset	Numeric Fields	Records	Unique Values	Description
RE	275	0.59M	1.38M	Real estate database.
BG	7	8.26M	0.12M	Demographic aggregate data from US Census 2000.

Baseline

The open source search system Apache *Solr* release 3.2 was used as baseline [Sol]. *Solr* is a widely adopted and sofisticated full free-text search engine that uses the *Apache Lucene* library[7] for indexing and retrieving documents. In addition, *Solr* incorporated in its latest releases support for geospatial queries [Sea11].

The *spatial number index* (*SNI*) and query processing method were implemented in Java. Application cache was disabled for both *SNI* and *Solr*.

Datasets and Numeric Data Compression

Datasets

We used the two real spatial databases shown in Table 4.5. The two datasets have constrasting characteristics that allow us to test different perspectives of our query processing techniques. The real estate database (*RE*) has a few hundreds of numeric fields, which can help measure the space overhead of indexing numeric data. On the other hand, the census database (*BG*) has substantially many more objects than *RE*, which can help in assessing scalability of our query processing techniques.

Numeric Data Compression

We encoded all the numbers found in numeric attributes of the spatial databases using the method described in Section 4.3.2. Figure 4.10 shows a histograms of the number of bytes used in encoding numbers per each database. Database *RE*'s

[7] http://lucene.apache.org

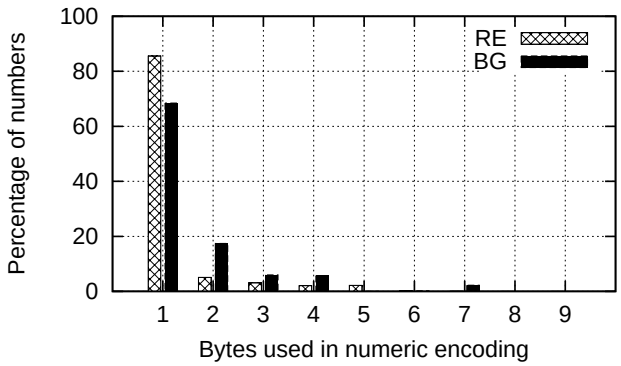


Figure 4.10: Histogram on the size of numeric data encodings.

numbers are highly compressible by the encoding method. In particular, more than 80% of values were compressed by a single byte. This is because the *RE* has a lot of small integer values, e.g. number of bedrooms or apartment number, that are suitable to be compressed by a few bytes. Database *BG* has a more balanced type of numeric values, including integers for people's ages, and decimals for average values or land areas. However, still a large percentage of values (more than 60%) were able to be compressed by as few as one byte.

Query Types

We measured query performance of two types of query range constraints on a given numeric attribute v:

- Type-1: $v \geq Q_l$ – single-bounded constraint.
- Type-2: $Q_l \leq v \leq Q_u$ – double-bounded constraint.

The number of nearest neighbor queries to be retrieved was fixed to $k = 10$. We chose one numeric attribute from *RE*, and two numeric attributes from *BG* to execute k-SBn queries. The attributes where chosen in such a way that their distributions were substantially different so we could study different behaviors. Figure 4.11a shows the CDF plottings of the chosen numeric attributes. In particular,

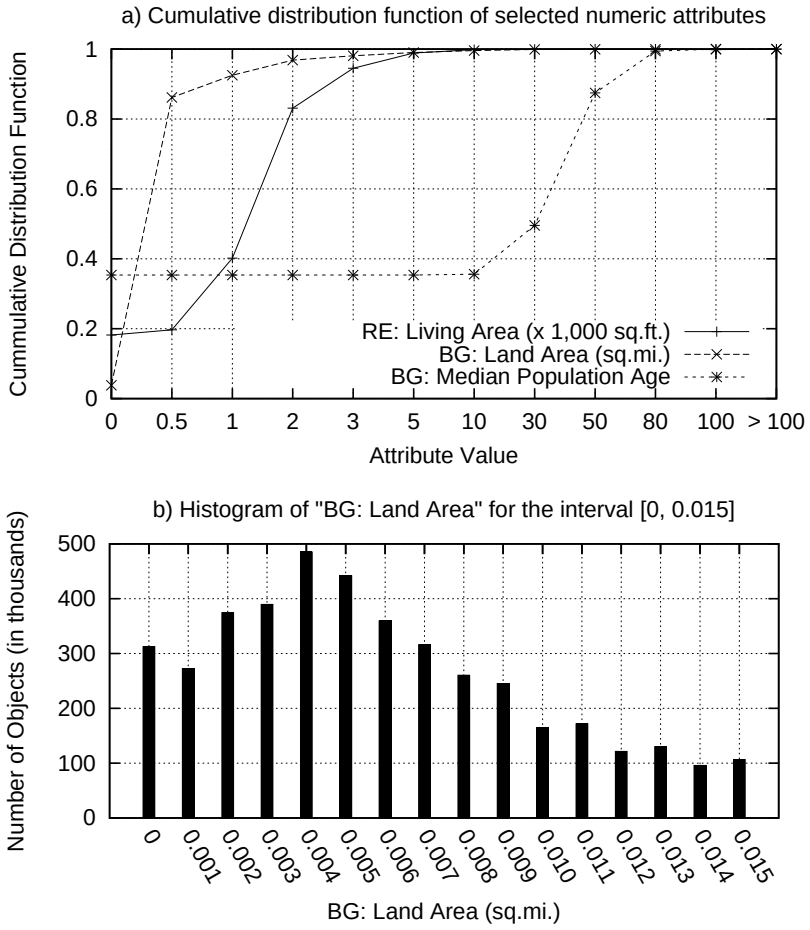


Figure 4.11: Data distribution of selected numeric attributes of databases RE and BG in Table 4.5.

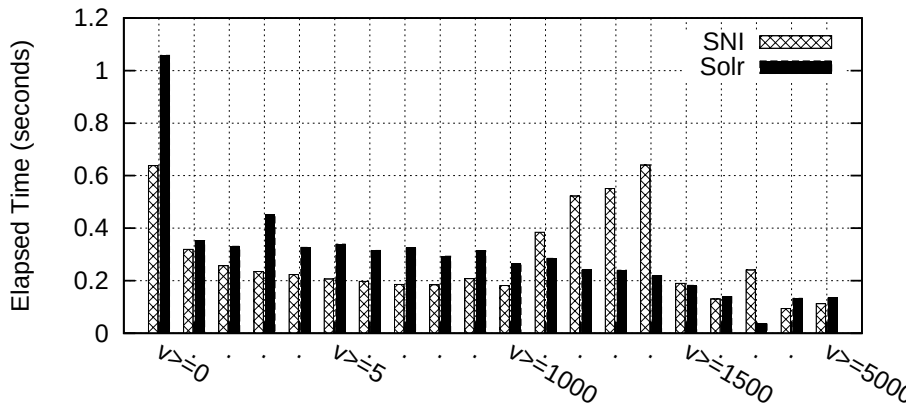
the attribute "BG: Land Area" has a large percentage of values distributed in a small interval as can be seen in the histogram of Figure 4.11b.

We measured query processing times (*QTime* in *Solr*) of k-SBn queries. Query processing times did not include record retrieval time.

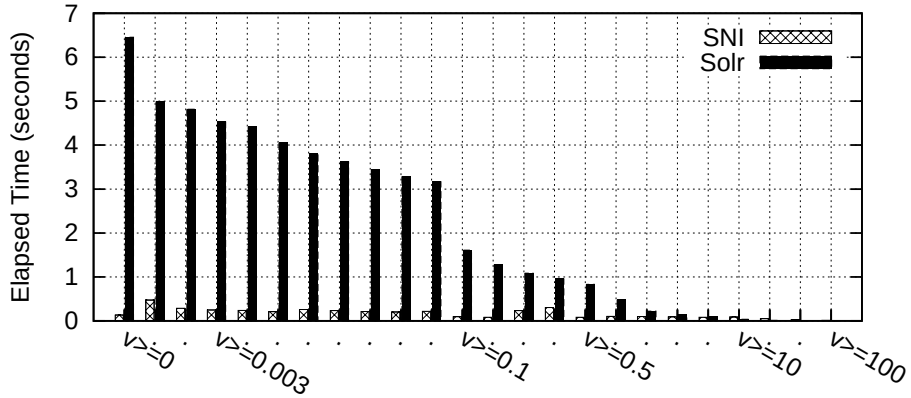
Performance Evaluation

Figure 4.12 shows multiple plottings comparing performance of executing k-SBn queries of Type-1 with both *SNI* and *Solr* methods. First, we can see in Figure 4.12a that no single method performs better than the other in all the cases for queries over dataset *RE*. However, *SNI* consistently keeps performance advantages for frequent values, i.e. $v < 1000$. For less frequent values, i.e. $v \geq 1000$, we observe some peaks for *SNI* while *Solr*'s time steadily decreases. One reason for these peaks is that the number of qualifying objects is relatively small (values are infrequent), and objects might be sparsed in the space, thus forcing *SNI* to visit additional branches in the R-tree.

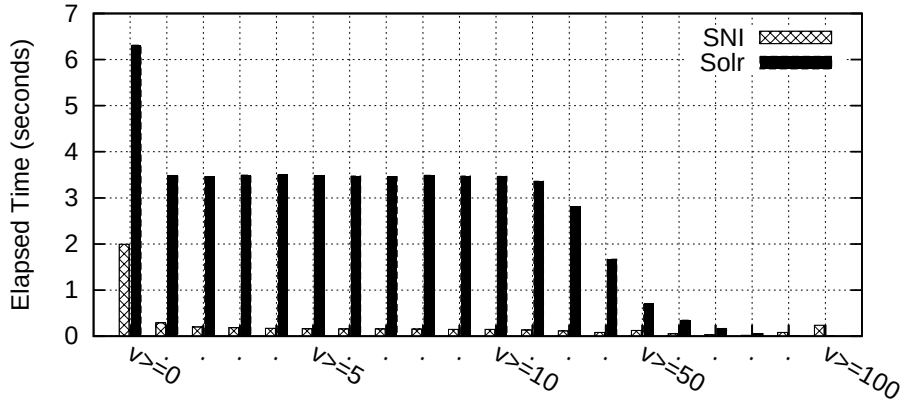
For the larger dataset *BG*, the performance gap is substantially larger with clear consistent advantages of *SNI* over *Solr* for all the large majority of the cases as can be seen in Figures 4.12b and 4.12c. Like in the database *RE* case, *Solr* struggles at the beginning when the numeric range includes very frequent values that do not help filter out objects; for example, the range constraint $v \geq 0$ on the "BG: Land Area" attribute filters no object whatsoever. In fact, query processing time for *Solr* degrades up to 6 seconds. Another factor that influences poor performance for *Solr* is the scale of the *BG* database. *BG* contains more than eight million objects, thus it appears that at larger scale performance defficiencies of *Solr*'s spatial query processor may be magnified.



a) Range constraint on "RE: Living Area" attribute.

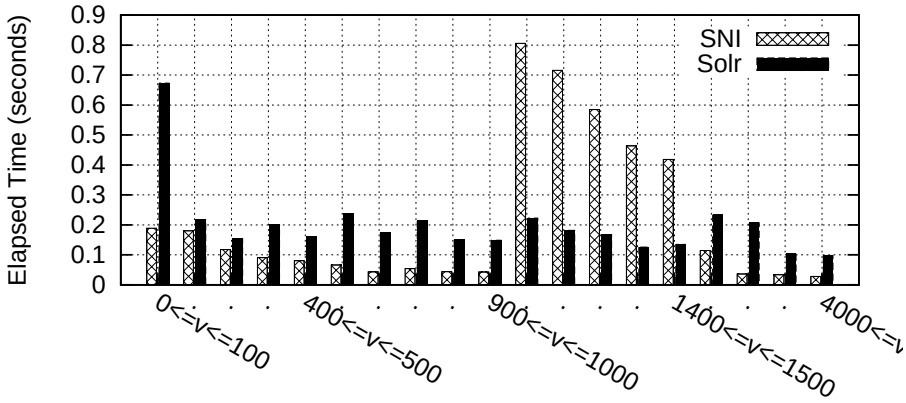


b) Range constraint on "BG: Land Area" attribute.

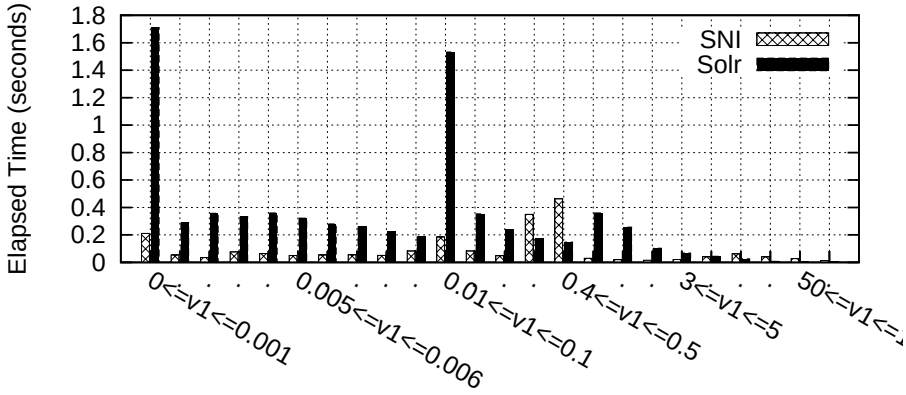


c) Range constraint on "BG: Median Population Age" attribute.

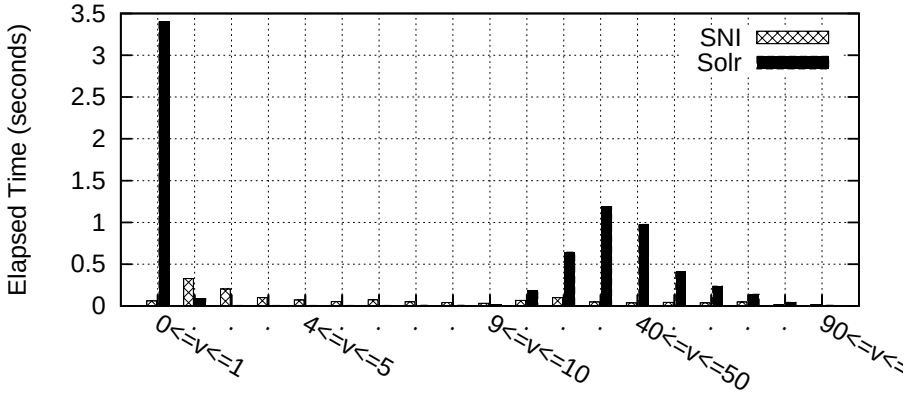
Figure 4.12: Performance of Type-1 k-SBn queries.



a) Range constraint on "RE: Living Area" attribute.



b) Range constraint on "BG: Land Area" attribute.



c) Range constraint on "BG: Median Population Age" attribute.

Figure 4.13: Performance of Type-2 k-SBn queries.

Table 4.6: Index sizes on secondary storage

Index Type	Dataset	
	RE	BG
Solr	3,685 MiB	2,037 MiB
SNI	1,978 MiB	2,032 MiB

A similar pattern is observed in the set of query experiments of Type-2 in Figure 4.13. However, since the range constraints are bounded on both sides, the universe of candidate objects is smaller than in the Type-1 queries. The latter alleviates *Solr* in processing fewer objects. In Figure 4.13c, *SNI* underperforms in some cases, but response times are still acceptable (within 100 milliseconds).

In general, *SNI* has the advantage of searching only nearby regions, while *Solr* appears to be sensitive to the number of objects that satisfy the query range condition.

Secondary Storage Consumption

Index sizes are shown in Table 4.6. *SNI* consumes substantially less space than *Solr* for the *RE* dataset, which has 275 numeric attributes. Storage savings in *SNI* are due to number compression that allows to encode even decimal numbers with two or three bytes, while, for instance in Java, a native floating point data type would require four bytes.

4.4 Indexing Large Databases

Practical GIS applications must be able to cope up with constantly growing geospatial datasets. For querying purposes, fresh data generally has to be first indexed before users can start posting queries. The spatial keyword (*SKI*) and spatial number (*SNI*) indexes presented in Sections 4.2.2 and 4.3.3 rely on two main tree-based

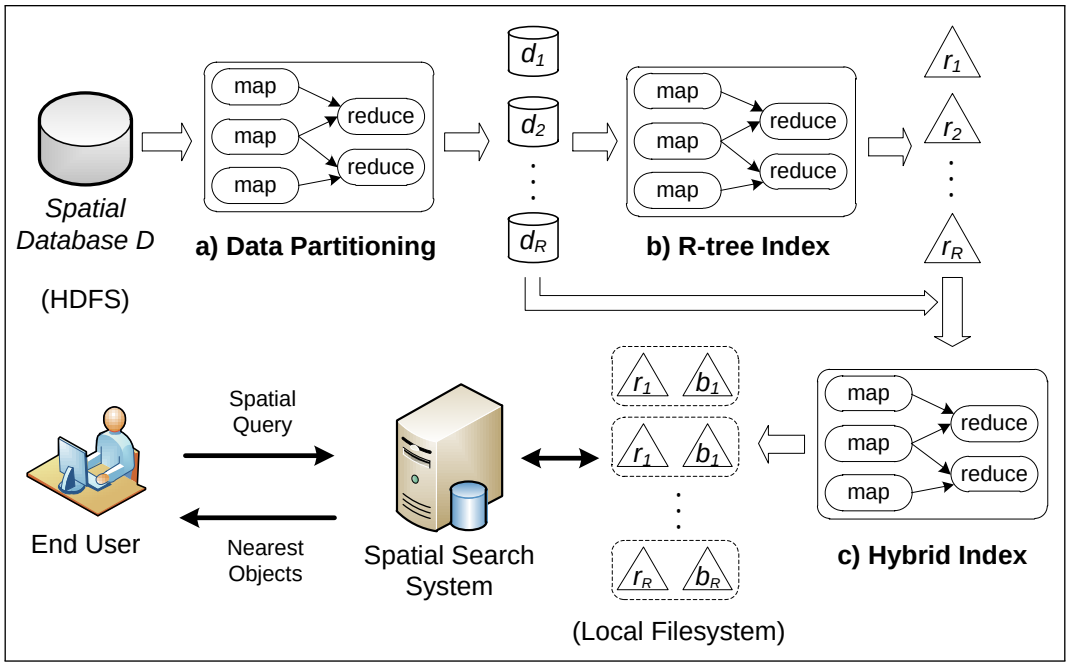


Figure 4.14: Spatial data indexing in MapReduce and local query processing.

data structures: R-trees and B-trees. as discussed in Sections 4.2.2 and 4.3.3. In this section, we leverage the parallel spatial data indexing techniques using the MapReduce programming model that were developed in Chapter 3 to construct the R-tree underlying data structure of *SKI* and *SNI* [CYAR10].

4.4.1 Architecture Overview

Figure 4.14 shows the general architectural components and data pipelines of the spatial data management system in the cloud. First, a spatial dataset D is uploaded to Hadoop's distributed file system (HDFS) for parallel processing via MapReduce jobs. Spatial data indexing is performed by three types of jobs executed in sequence:

a) Data Partitioning – These jobs prepare the data for parallel processing by partitioning D according to spatial attributes. Data partitions are non-overlapping subsets d_j such that $D = \left\{\bigcup_{j=1..R} d_j\right\}$, where R is a parameter that defines

the number of partitions. Two partitioning schemes were implemented based on: 1) Space-filling curves, and 2) X-means clustering [PM00], which automatically estimates the number of partitions (R). Section 4.4.2 empirically evaluates the performance characteristics of these schemes.

b) R-tree Index – MapReduce jobs take as input the data partitions $\{d_j\}$ R-tree constructions are executed in parallel by R reducers, one for each partition d_j, using the parallel construction method described in Chapter 3. Each reducer outputs an R-tree r_j built on its input d_j.

c) Hybrid Index – MapReduce jobs take as input the d_j small datasets and individual r_j R-trees built in previous phases. For each dataset d_j, textual data is tokenized, and numeric data is encoded. Then, term and number bitmaps are built with references to the r_j R-tree. The result is written in a B-tree b_j. The output is a set of hybrid, spatial keyword and number indexes, one for every input d_j.

Spatial Search System

After the spatial database is indexed in MapReduce, individual hybrid indexes are downloaded to a local filesystem for query processing.[8] Our spatial search system consolidates the individually built R-trees in a single, large R-tree that represents the R-tree index of spatial database D. The large R-tree is the main entry point for procesing spatial queries with textual constraints (k-SB) and numeric range constraints (k-SBn) posted by end users.

[8]At the time of our study, HDFS does not support random *I/O*, so it is not possible to execute queries directly on the distributed file system.

Data Partitioning via Clustering

As an alternate partitioning method to space-filling curves, we used X-means iterative clustering algorithm [PM00] to group the database in clusters using objects' spatial attributes. X-means extends the popular, statistical K-means clustering technique with good approximates for the parameter K – the total number of clusters. We implemented X-means clustering in MapReduce to find a clustering of spatial database D [CYAR10]. Each cluster found by X-means is considered an individual partition.

The advantage of clustering method is that an adequate number of partitions (i.e., the number of reducers R) is automatically estimated. On the other hand, partition sizes may vary considerably, and iterating on the clustering algorithm might be computationally expensive. To alleviate the first problem, we imposed a minimum cluster size before a cluster is considered for splitting to avoid generating very small partitions. On the second issue, our experimental results show that the additional clustering overhead, which is incurred only once, pays off by lowering query response times.

4.4.2 Experiments

The experimentation was divided in two parts. In the first part, two spatial keyword indexes (SKI) on a real spatial database were built on the Hadoop cluster described in Section 3.2.3. Each index used a different spatial partitioning variant based on Z-order values (ZO) and X-means clustering (XM). In the second part, query response times were measured using ZO and XM indexes running on a local machine.

Spatial Database

We used a spatial database of United States property parcels (USP) [Sol10]. Objects locations are represented by geographical coordinates of the parcels' locations. In addition, 17 other non-spatial attributes, including parcel id, owner's name, street address, and parcel type, were associated to objects. The dataset contains about 110 million objects, and its vocabulary size has around 12.8 million terms.

Spatial-Keyword Index Construction

We found a clustering of the database via X-means in MapReduce using 118 *mappers* and initial random cluster centroids. The number of clusters (*reducers*) were varied between 30 and 100. We set the minimum cluster size as one million whenever a local cluster was considered for splitting in X-means. We observed that the number of clusters consistently became stable around 60 after a few iterations. Clusterings with more clusters just marginally improved. We selected a clustering with 62 cluster as the best clustering. The number of clusters found by X-means was used as parameter for the Z-order partitioning scheme.

Figure 4.15 visualizes the clustering found by X-means after 7 iterations, and the partitions determined by Z-order values. In the figure, points are sampled object locations, and partitions are represented by their minimum bounding rectangles (MBR). Although MBR overlapping is inevitable, we can observe in Figure 4.15b that XM better approximates the distribution of the data and reduces overlapping while ZO incurrs in a lot more MBR overlapping, especially in denser areas like the eastern cost. It is a known result that excessive MBR overlapping hinders retrieval performance [BKSS90] [KF94]. In the next section, we empirically measure the effect of additional MBR overlapping incurred by ZO partitioning scheme.

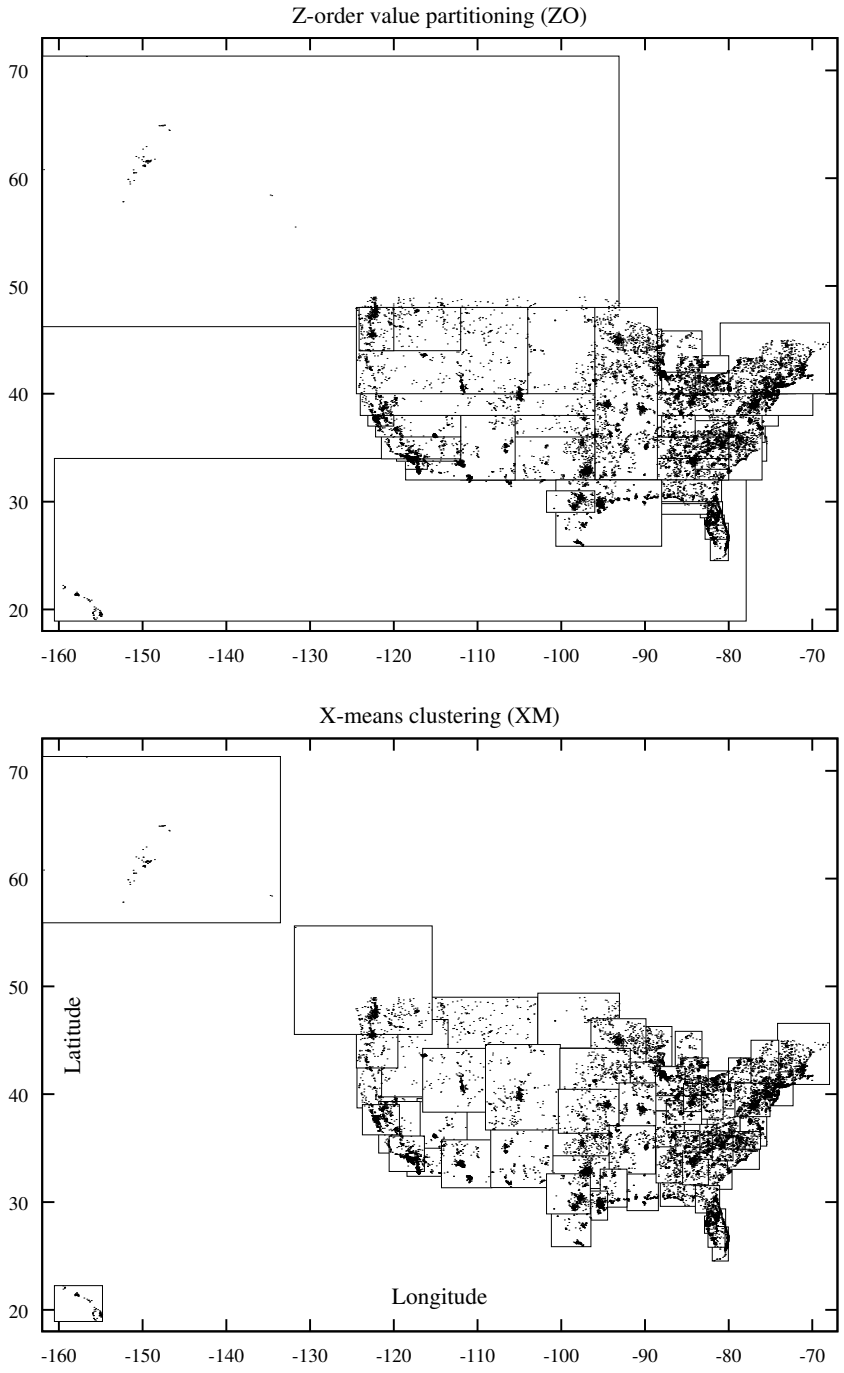


Figure 4.15: Data partitioning of United States property parcels based on Z-order values (top) and X-means clustering (bottom) with 62 partitions.

Spatial-keyword indexes were built individually for ZO and XM partitioning strategies and downloaded to our local site. Since the Hadoop cluster we used is shared with other researchers, it is hard to measure index construction times with accuracy. Elapsed times varied according to the cluster activity. Nonetheless, we observed that index construction times with ZO partitioning strategy were completed in about 40 minutes while indexing with XM partitioning fluctuated between 70 and 100 minutes. The dominant time factor in XM was the clustering phase (around 50%).

Query Processing

We evaluated elapsed times of queries using indexes build in parallel with both partitioning techniques ZO and XM. Queries were processed according to the algorithm in Section 4.2.3. Queries were run in an Intel Xeon E5520 machine with 16GB physical memory and two quad-core processors at 2.27GHz. Database and indexes were stored on a 6-disk RAID-5 array attached directly to the host. The main focus of this experiment was in measuring the number of R-tree nodes accessed during query processing, and the overall elapsed time.

Query Workload

k-SB queries textual constraints were generated as follows. A query location Q_l was randomly chosen from the USP space, and textual constraints included 1, 2 and 3 randomly selected terms from the USP lexicon. The special case k-SB queries with no text constraints, i.e. conventional k-NN queries, was also considered. The numbed of retrieved records was set to $k = 50$. Figure 4.16 shows the minimum, maxium, and median measurements over 100 k-SB queries with $0, 1, 2$ and 3 terms in their non-spatial constraints.

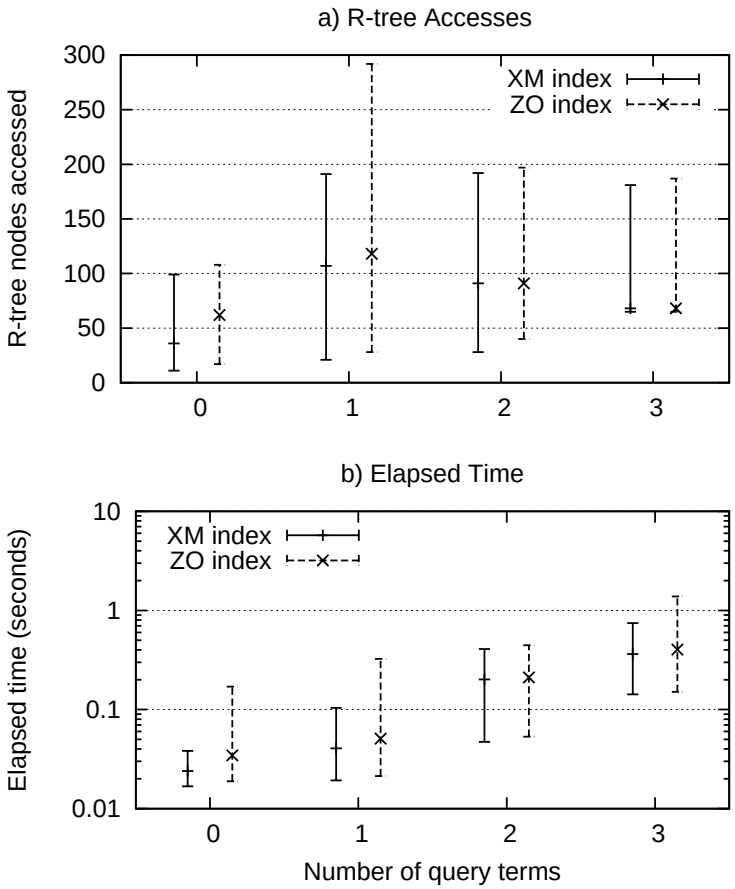


Figure 4.16: Performance metrics minimum, maximum, and median over 100 k-SB queries with $0, 1, 2$ and 3 query terms in the non-spatial constraint. Y-axis in b) (elapsed times) is in logarithmic scale.

The experimental results confirmed what we expected. For regular k-NN queries (no constraints), the median of R-tree nodes accessed by ZO index is around 70% higher than XM index as can be seen in Figure 4.16a due to increased MBR overlapping. Response times follow similar behavior for those queries in Figure 4.16b. Queries with 1-term constraints still incurr in extra R-tree node retrievals for ZO index compared to XM index. Notably, queries with ZO index generated a large gap between the minimum and maximum values on both R-tree nodes retrieved and response times. For 2-term and 3-term queries, R-tree nodes accessed are comparable between ZO and XM indexes, which indicates that candidate objects are more scattered in the search space. Response times are mostly dominated by accesses to B-tree indexes, and thus are proportional to the number of query terms.

4.5 Related Work

4.5.1 Spatial Nearest Neighbor Queries

The R-tree traversal method in our work is inspired in the Hjaltason and Samet's [HS99] incremental top-k nearest neighbor algorithm using R-trees [HS99]. Performance improvements on the original R-tree [Gut84] spatial index data structure have been proposed, e.g. R*-tree [BKSS90], R+-tree [SRF87], Hilbert R-tree [KF94] and Priority R-tree [AdBHY04]. These variants form a familiy of R-trees that can replace the R-tree index (with proper super node range augmentation) used in our proposed hybrid spatial keyword index without modifying our search algorithms.

4.5.2 Information Retrieval

In information retrieval, inverted files are arguably the most efficient index structure for free-text searches [ZM06] [ZMR98]. The processing of k-SB query Boolean predicates Q_B in Algorithm 4.2 is similar to the Document-At-A-Time style of processing in inverted files. In our algorithm, we traverse simultaneously the bitmaps of terms in AND-semantics, OR-semantics and NOT-semantics subsets until we find a super node that contains objects satisfying Q_B. Since the output of a free-text search query can be potentially large, the result list is typically sorted by a ranking function that gives a relevance score to each retrieved document. In our work, we assume that users have a location reference (the query location), and they are interested in finding the nearest database objects (sorted by proximity to them) satisfying Boolean criteria on textual and numeric data.

4.5.3 Spatial Keyword Queries

The problem of retrieving spatial objects satisfying non-spatial constraints has been studied in the recent past. Park and Kim [PK03] proposed RS-trees, a combination of R-trees and Signature trees for database attributes with controlled cardinality. Signature chopping is suggested to mitigate the *combinatorial error* [CS89] (database overrepresentation) generated by superimposing signature files, which leads to false drops – a combination of query terms may pass the membership test with a superimposed signature, but no actual record in the database has such combination of terms. Hariharan et al. [HHLM07] proposed to include inverted files in every node of an R-tree. Inverted files in a given node index terms found inside the sub-trees rooted at that node. De Felipe et al. [DFHR08] augmented signature files in R-tree nodes with similar assumptions as in [PK03]. Recently, Cong

et al. [CJW09] augmented an inverted file in every node of an R-tree, and used a combined ranking function that mixes spatial proximity and text relevancy in a single score. Our work differs in that we assume distance as ranking score, and we focus on efficiently processing Boolean selection predicates on textual and numeric data. In addition, previous works offer no efficient processing of the complement logical operator ($\neg$), which limits their applicability to the k-SB type of queries we considered in this work.

4.5.4 Spatial Queries in Database Management Systems

In current database management systems (DBMS), B-tree and its variant B+tree [Com79] are the most commonly used type of data strtuctures for indexing both numeric and textual table columns, e.g. Oracle [Ric11] and MS SQL Server [Dat11] relational databases. Current systems have incorporated extensions to manage geospatial data, as well as extensions to the SQL language to express spatial relations in queries. Database systems usually execute spatial queries with Boolean predicates by applying the Boolean filters first (e.g., on indexed columns), then the spatial relation (e.g., k-NN or window query) is applied on the results of the Boolean filter, or in the reverse order if the query optimizer determines that it is more efficient. That is, existing DBMS's perform a spatial-only filter or Boolean-only filter at a time. Thus, the retrieval cost is sensitive to the output size of the first filter. In contrast, our work strives to push both filters simultaneously during query processing.

4.5.5 Numeric Range Constraints

Several works have been done in optimizing data retrieval constrained with numeric ranges. Rishe has proposed an encoding method for numeric data based on a tree of

sub-intervals, which was successfully applied to a semantic database [Ris92]. Fontoura et al. have studied how to efficiently represent numeric ranges in inverted files in search engines to support searches with numeric range constraints [FLQZ07]. Schindler and Diepenbroek proposed a method for encoding numbers with variable precision into strings using a trie-based algorithm [SD08]. Their method was adopted by the *Apache Lucene* project – an open-source, advanced library for indexing and free-text searching[9] – for supporting efficient processing of queries with numeric range constraints.

4.5.6 Web Mapping Services and Search Systems

Modern web search engines provide local-search functionalities on business listings, e.g. Google Maps [mGM] and Bing Maps [mBM], for finding documents geolocated to a particular place, e.g. a postal code or city.. While some of these services do support spatial keyword searches, numeric conditions are not currently supported (e.g., Google Maps does not allow to use numeric range constraints in queries like "large pizza $5..$10 in Miami" in local-search queries in the same way they are supported in regular web searches). In addition, some Map services implement advanced querying options that allow users to include all terms, optionally some terms, and exclude certain terms from the search results. These constraints are similar to the k-SB Boolean selection criteria we defined in our work. Aside of that, the specific query processing techniques used by these systems are not disclosed in any detail.

In the open source domain, *Apache Solr* [Sol] is a full-text search server based on the *Apache Lucene* library. *Solr* has incorporated geospatial search support in its latest releases [Sea11]. Since spatial and numeric range querying is implemented in

[9]http://lucene.apache.org

Solr, we chose this system as the baseline to compare performance with our proposed methods for processing spatial queries with numeric constraints.

4.6 Summary

In this chapter, we studied the problem of spatial search queries with constraints on textual and numeric data. We described disk-resident, hybrid indexes for efficiently answering k-NN queries with Boolean constraints on textual and numeric content. We combined a modified version of the R-tree index to organize spatial attaributes, with a B-trees to store text and numeric data linked to the location where they occur in the R-tree. Algorithms were presented to show how the new spatial indexes are used in processing k-NN queries, achieving effective pruning of the search space. Worst case time complexity upper bounds were also discussed for the core query processing algorithms. Our experimental evaluation with large, real spatial datasets (with up to 110 millions of objects) showed increased performance and scalability over alternate query methods.

CHAPTER 5

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

This chapter presents concluding remarks of the present work, and it outlines possible direction in which this work can be extended.

5.1 Concluding Remarks

In this thesis, we addressed two related research problems in current spatial databases. First, the scalability problem of building R-tree indexes on large databases was tackled leveraging the MapReduce parallel programming model. We proposed data partitioning methods and strategies for building R-trees expressed as MapReduce compounds. R-tree construction times were significantly lowered with the parallel approach, which achieved close to linear scalability as more compute nodes were used in index construction tasks.

Second, query processing of spatial queries with Boolean selection criteria on aspatial attributes were studied and an adequate hybrid index was proposed to improve query efficiency. The proposed hybrid index resulted from a systematic combination of two widely used indexes: R-trees and B-trees. We proposed algorithms for efficiently answering spatial queries with Boolean constraints on textual and numeric database attributes. Worst case analysis was provided for the proposed algorithms. Experimental evaluation of our techniques in a prototype search system showed improved performance over alternate query processing methods.

5.2 Future Research Directions

There is a new type of problems identified as big data problems where data grows to the extend that it becomes difficult to store it or perform useful computations

on it. Presently, most of the user-generated data (e.g., pictures, videos, or text messages), and data automatically generated by machines (e.g., log entries in search engines) have explicit or implicit geographical information. Data analytics and data mining techniques are used in many industries to make inferences and identifying data patterns. Incorporating the geographical dimension in the analysis is highly desirable as it can reveal "local" insights [BCR11]. The techniques developed in this thesis may be helpful in addressing the problem of combined analysis of geographical and non-geographical data. However, due to the scale of big data problems, it becomes cumbersome to perform sophisticated data computations in big data environments. Thus, new architectures and methodologies for spatial and non-spatial big data processing are worthwhile researching.

Even more, performing data analysis in real time in big data environments is another current challenge. Updates should be done in real time as much as possible. For example, in a data analysis application we may want to update statistical metrics incrementally as new data arrives at high rates without the need to recompute everything from scratch. At the present time, the scale issue remains a challenging problem that precludes real time analysis.

Techniques like MapReduce have limitations in doing real time processing. There is currently active research interest in evolving MapReduce to make it more real time ready. In the industry, Google has developed a solution for their problem of incrementally updating the web index using an internal solution called Percolator [PD10]. Similar solutions should be studied for other types of spatial and non-spatial applications that have online requirements.

BIBLIOGRAPHY

[ABL10] Sattam Alsubaiee, Alexander Behm, and Chen Li. Supporting location-based approximate-keyword queries. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, pages 61–70, New York, NY, USA, 2010. ACM.

[AdBHY04] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority r-tree: a practically efficient and worst-case optimal r-tree. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 347–358, New York, NY, USA, 2004. ACM.

[AM90] David J. Abel and David M. Mark. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information Science*, 4:21–31, January 1990.

[ARR+97] Tetsuo Asano, Desh Ranjan, Thomas Roos, Emo Welzl, and Peter Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theor. Comput. Sci.*, 181:3–15, July 1997.

[BCR11] Jaime Ballesteros, Ariel Cary, and Napthtali Rishe. SpSJoin: Parallel spatial similarity joins (demo). In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11. (in press), 2011.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19:322–331, May 1990.

[Cen11] High Performance Database Research Center. *http://hpdrc.fiu.edu.* 2011.

[CJW09] Gao Cong, Christian S. Jensen, and Dingming Wu. Efficient retrieval of the top-k most relevant spatial web objects. *Proc. VLDB Endow.*, 2:337–348, August 2009.

[Com79] Douglas Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11:121–137, June 1979.

[CS89] W. W. Chang and H. J. Schek. A signature access method for the starburst database system. In *Proceedings of the 15th international conference on Very large data bases*, VLDB '89, pages 145–153, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[CSHR09] Ariel Cary, Zhengguo Sun, Vagelis Hristidis, and Naphtali Rishe. Experiences on processing spatial data with MapReduce. In *Proceedings of the 21st International Conference on Scientific and Statistical Database Management*, SSDBM 2009, pages 302–319, Berlin, Heidelberg, 2009. Springer-Verlag.

[CWR10] Ariel Cary, Ouri Wolfson, and Naphtali Rishe. Efficient and scalable method for processing top-k spatial boolean queries. In *Proceedings of the 22nd international conference on Scientific and statistical database management*, SSDBM'10, pages 87–95, Berlin, Heidelberg, 2010. Springer-Verlag.

[CYAR10] Ariel Cary, Yaacov Yesha, Malek Adjouadi, and Naphtali Rishe. Leveraging cloud computing in geodatabase management. In *Proceedings of the 2010 IEEE International Conference on Granular Computing*, GRC '10, pages 73–78, San Jose, CA, USA, 2010. IEEE Computer Society.

[Dat11] Microsoft SQL Server Database. *http://www.microsoft.com/sqlserver*. Microsoft Corporation, May 2011.

[DFHR08] Ian De Felipe, Vagelis Hristidis, and Naphtali Rishe. Keyword search on spatial databases. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 656–665, Washington, DC, USA, 2008. IEEE Computer Society.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[FLQZ07] Marcus Fontoura, Ronny Lempel, Runping Qi, and Jason Y. Zien. Inverted index support for numeric search. *Internet Mathematics*, 3(2):153–185, 2007.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37:29–43, October 2003.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14:47–57, June 1984.

[HHLM07] Ramaswamy Hariharan, Bijit Hore, Chen Li, and Sharad Mehrotra. Processing spatial-keyword (sk) queries in geographic information retrieval (gir) systems. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, SSDBM '07, pages 16–, Washington, DC, USA, 2007. IEEE Computer Society.

[HS99] Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24:265–318, June 1999.

[Ini07] Google&IBM Academic Cluster Computing Initiative. *http://www.google.com/intl/en/press/pressrel/20071008_ibm_univ.html*. 2007.

[KF94] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 500–509, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[LK00] Jonathan K. Lawder and Peter J. H. King. Using space-filling curves for multi-dimensional indexing. In *Proceedings of the 17th British National Conferenc on Databases: Advances in Databases*, BNCOD 17, pages 20–35, London, UK, 2000. Springer-Verlag.

[mBM] Microsoft's Bing Suite. Bing Maps. *http://www.bing.com/maps*.

[mGM] Google Mapping Services. Google Maps. *http://maps.google.com*.

[Mor66] G.M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Co., 1966.

[oGC10] WikiProject on Geographical Coordinates. *http://en.wikipedia.org/wiki/Wikipedia:WikiProject_Geographical_coordinates*. 2010.

[O'M08] Owen O'Malley. Terabyte sort on apache hadoop. Technical Report Ottawa, Ontario, Canada, May 2008.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[PD10] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.

[PK03] Dong-Joo Park and Hyoung-Joo Kim. An enhanced technique for k-nearest neighbor queries with non-spatial selection predicates. *Multimedia Tools Appl.*, 19:79–19, January 2003.

[PM00] Dan Pelleg and Andrew W. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 727–734, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[PM03] Apostolos Papadopoulos and Yannis Manolopoulos. Parallel bulk-loading of spatial data. *Parallel Comput.*, 29:1419–1444, October 2003.

[Pro08] NSF Cluster Exploratory Program. *http://www.nsf.gov/pubs/2008/nsf08560/nsf08560.htm*. 2008.

[Pro11] The Apache Hadoop Project. *http://hadoop.apache.org/*. 2011.

[Red11] Hadoop Class Reducer. *http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/mapreduce/Reducer.html*. 2011.

[Ric11] Bert Rich. *Oracle Database Reference 11g Release 2 (11.2) E25513-01*. Oracle Corporation, September 2011.

[Ris92] Naphtali Rishe. Interval-based approach to lexicographic representation and compression of numeric data. *Data and Knowledge Engineering*, 8(4):339 – 351, 1992.

[SD08] Uwe Schindler and Michael Diepenbroek. Generic XML-based framework for metadata portals. *Computers and Geosciences*, 34(12):1947–1955, December 2008.

[Sea11] Solr Wiki: Spatial Search. *http://wiki.apache.org/solr/SpatialSearch*. May 2011.

[SL99] Bernd Schnitzer and Scott T. Leutenegger. Master-client R-Trees: A new parallel R-Tree architecture. In *Proceedings of the 11th International Conference on Scientific and Statistical Database Management*, pages 68–, Washington, DC, USA, 1999. IEEE Computer Society.

[Sol] Apache Solr. *Availabel at: http://lucene.apache.org/solr/*.

[Sol10] First American Spatial Solutions. *Spatial database of United States property parcels*. 2010.

[SQ08] United States Census Bureau Florida State and County QuickFacts. *http://quickfacts.census.gov/qfd/states/12000.html*. July 2008.

[SRF87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[SRT+08] Steven W. Schlosser, Michael P. Ryan, Ricardo Taborda, Julio López, David R. O'Hallaron, and Jacobo Bielak. Materialized community ground models for large-scale earthquake simulation. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 54:1–54:12, Piscataway, NJ, USA, 2008. IEEE Press.

[WCF+07] Xiaqing Wu, Rodrigo Carceroni, Hui Fang, Steve Zelinka, and Andrew Kirmse. Automatic alignment of large-scale aerial rasters to road-maps. In *Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, GIS '07, pages 17:1–17:8, New York, NY, USA, 2007. ACM.

[WOS06] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31:1–38, March 2006.

[YDHP07] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38, July 2006.

[ZMR98] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23:453–490, December 1998.

VITA

ARIEL CARY-HUANCA

2011	Ph.D., Computer Science Florida International University Miami, Florida
2006–2011	Graduate Research Assistant Florida International University Miami, Florida
2003–2005	Database Administrator Empresa Nacional de Telecomunicaciones S.A. La Paz, Bolivia
2000	B.Sc., Systems Engineering Catholic Bolivian University Cochabamba, Bolivia

PUBLICATIONS AND PRESENTATIONS

1. Jaime Ballesteros, Ariel Cary, and Napthtali Rishe. *SpSJoin: Parallel Spatial Similarity Joins (demo).* In Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Chicago, IL, 2011 (in press).
2. Ariel Cary, Yaacov Yesha, Malek Adjouadi, and Naphtali Rishe. *Leveraging Cloud Computing in Geodatabase Management.* In Proceedings of the 2010 IEEE International Conference on Granular Computing (GrC), pp. 73–78, San Jose, CA, 2010.
3. Ariel Cary, Ouri Wolfson, and Naphtali Rishe. *Efficient and Scalable Method for Processing Top-k Spatial Boolean Queries.* In Proceedings of the 22nd International Conference on Scientific and Statistical Database Management (SSDBM), pp. 87–95, Springer-Verlag, Berlin, Heidelberg, 2010.
4. Ariel Cary, Zhengguo Sun, Vagelis Hristidis, and Naphtali Rishe. *Experiences on Processing Spatial Data with MapReduce.* In Proceedings of the 21st International Conference on Scientific and Statistical Database Management (SSDBM), pp. 302–319, Springer-Verlag, Berlin, Heidelberg, 2009.
5. Onyeka Ezenwoye, S. Masoud Sadjadi, Ariel Cary, and Michael Robinson. *Grid Service Composition in BPEL for Scientific Applications.* In Proceedings of the OTM Confederated International Conference, pp. 1304–1312, Springer-Verlag, Berlin, Heidelberg, 2007.