DISTRIBUTED APPROACHES TO SPATIAL PIVOT INDEXING

by

Kevin Tyler
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Master of Science
Geography and Geoinformation Science

Committee:

____________________________________	Dr. Chaowei Yang, Thesis Director
____________________________________	Dr. Arie Croitoru, Committee Member
____________________________________	Dr. Dieter Pfoser, Committee Member
____________________________________	Dr. Anthony Stefanidis, Department Chairperson
____________________________________	Dr. Donna M. Fox, Associate Dean, Office of Student Affairs & Special Programs, College of Science
____________________________________	Dr. Peggy Agouris, Dean, College of Science
Date: ______________________________	Spring Semester 2016 George Mason University Fairfax, VA

Distributed Approaches to Spatial Pivot Indexing

A Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

by

Kevin Tyler
Bachelor of Science
University of Mary Washington, 2012

Director: Chaowei Yang, Professor
Department of Geography and Geoinformation Science

Spring Semester 2016
George Mason University
Fairfax, VA

This work is licensed under a creative commons attribution-noderivs 3.0 unported license.

DEDICATION

This document is dedicated to my grandmother, Gertrude Quigley. You are as beautiful and gracious as they come.

ACKNOWLEDGEMENTS

To Dr. Arnold Boedihardjo; thank you for the inspiration and direction of this research, and for your continued guidance as a supervisor and mentor.

To my coworkers, Dr. Ray Dos Santos, Matt Renner, and Harland Yu: thank you for tolerating my many intrusions into your workspaces over the past six months. I greatly appreciate your willingness to answer questions and reflect the ideas bounced off of you.

And to my wife, Kayla: thank you for your endless support, and for always understanding my replies of “I can’t, I have homework.” Only five more years to go!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF EQUATIONS

ABSTACT

DISTRIBUTED APPROACHES TO SPATIAL PIVOT INDEXING

Kevin Tyler, MS

George Mason University, 2016

Thesis Director: Dr. Chaowei Yang

Spatial indexing is critical for retrieving data efficiently from geospatial databases, and has been a long-standing research direction of GIS. In a departure from recent spatial indexing paradigms, this study leverages pivot indexing. Pivot indexing begins by selecting a small number of points from the dataset. Then, the distances from the points in this subset- the *pivots*- to every point in the database are stored in secondary memory. During query time, these pre-computed values are used to evaluate candidates for range or nearest neighbor search. This approach offers a substantial reduction in the number of distance computations necessary to evaluate objects in the spatial plane.

While previous spatial pivot indexing research leverages graphical processing units, this study utilizes an alternative parallelization mechanism- distributed computing. The Hadoop file system is used to accelerate index creation and querying in a scalable fashion. The results are then compared to an existing distributed solution. I ultimately discovered that my implementation of the pivot index at the distributed level

underperformed existing methods by a small margin; however, my implementation offered an improved kNN query performance. These results affirm the legitimacy of the pivot indexing approach, and suggest that similar approaches deserve further investigation.

Ultimately, this research does not assert that pivot indexing is superior to other approaches. Rather, it is an exploration of pivot indices in a specific computing context (the Hadoop File System). Thus, the contribution to the research community is in application. The outcome of this study is to demonstrate the utility of this particular indexing paradigm in the Hadoop software environment.

CHAPTER 1: INTRODUCTION

The demand to accelerate search and query performance within the computational realm is substantial. Indeed, while searching for a particular passage within a voluminous textbook, an individual will hope to avoid reading every page from beginning to end- and this simple premise extends to search within a computer’s storage for a particular object. This obvious realization has given rise to an independent subdomain within the database community, and *indexing*- whether it be of text, images, and numerous primitive data storage types- is a non-trivial problem that has been the subject of numerous research publications.

While computer indexing is a well-defined topic within the information retrieval domain, the capacity to interrogate collections of multidimensional objects within the metric space can yet be improved. The crucial distinction is that traditional computer indices can function only in one-dimensional spaces. This begs the question- how can long-standing solutions to the indexing problem be extended to data that is inherently spatial? Does the answer lie in dimensionality reduction techniques, such as principle component analysis? If so, what measures must be taken to ensure that a given index for spatial data is capable of sufficiently describing a multi-dimensional object in a one-dimensional space without significant loss of data? Can these conceptual constructs truly produce computing solutions that accelerate querying of spatial objects in a manner that is competitive with traditional information retrieval?

Fortuitously, as social media platforms have become increasingly integral pieces of modern culture, technological innovation has provided massive amounts of locational data with which we can explore the answers to these questions. As such, this study revolves around a singular question: can spatial indexing strategies be improved in a way that both optimize query performance *and* scale well to massive datasets (i.e.: “Big Data”)?

Higher Dimensionality Applications of Pivot Indices

Previous work has extended the pivot indexing structure into spaces of higher dimensionality. The most seminal research [11] transformed a database of photographs of human beings into multi-dimensional objects in the feature space and performed normal pivot indexing operations on them. However, the extension of traditional pivot indexing principles (such as Sparse Spatial Selection) into non-spatial datasets is an imperfect one. This particular example includes dimensions that are not implicitly spatial, and this blending of data types is potentially problematic when applying operations designed to run on inherently spatial observations.

To truly apply higher dimensional data operations to pivot index schemes, it is necessary to control for the disparate nature of data distribution. This may be accomplished through a kind of two-step process, where the data is first normalized (through a technique such as z-order normalization), then reduced in dimensionality (via a transformation technique such as principle component analysis).

For this Geoinformatics master thesis using the pivot indexing technique, I will exclusively refer to spatial data. While the complications of extending pivot indexing to higher dimensions are important to note, further discussion on this topic is highly theoretical and beyond the scope of this thesis.

Objectives

Recent technological trends- such as social media- has created datasets of increasing size. In response, "NoSQL" technologies (databases that deviate from traditional relational data storage) have become increasingly popular. While pivot indexing has been well-explored via GPU technology, I believe that there is utility in implementing a NoSQL approach. Thus, my objective is to implement a spatial indexing using the concepts put forth from various literatures, and to evaluate the performance in accessing several disparate spatial datasets in a distributed environment.

For prototyping, the initial implementations will be "single-node" (i.e.: designed to be run on a single computer and not leveraging parallelization and storage capabilities from multiple machines). My expectation will be that query performance will be fairly satisfactory, especially with smaller datasets. I expect this largely because of the ability to hold a dataset in Random Access Memory during index creation and querying, as this does not incur the I/O costs associated with loading a record from secondary memory (such as a magnetic hard disk).

To leverage synchronous computing technology, this study will implement a pivot indexing solution based on a distributed platform. This software implementation based on the Apache Hadoop File System (a framework for storing data across a server "cluster," comprised of multiple physical machines), Apache Accumulo (a wide-column data store built atop of the Hadoop File System), Apache Storm (a suite that provides distributed processing of real-time data for high throughput) and Apache Kafka (a distributed queuing system). The “pivot table” described in previous work will be stored in Accumulo, with several customizations to leverage the opportunities for efficiency within the Accumulo structure. This indexing scheme will be tested in a distributed environment that includes 19 machines functioning as a cluster.

This serves as a departure from existing work in spatial pivot indexing research, as previous efforts parallelize index creation and querying processes via GPU technology. To my knowledge, neither the Hadoop File System nor any distributed computing framework have been used for pivot indexing. While the research detailed in [15] uses an external server for processing data, there is a key distinction- in a Hadoop environment each server holds a subset of the data and processes only that data during runtime.

After deployment of this framework, a number of datasets will be tested for index building and query performance (range and KNN). The test datasets are a blend of synthetic data (i.e.: created pseudo-randomly) and recordings of actual observations, ranging from 5,000 points to approximately 1.1 million. The intent is to observe how the indexing performance scales as input datasets grow in volume.

Finally, the results will be compared against an open-source distributed indexing package, Geomesa. Geomesa is architecturally similar to the software package proposed here (it also relies on the storage and iteration framework provided by Accumulo), and is self-described as an attempt to create a relationship between itself and Accumulo similar to the one between PostgreSQL and PostGIS. Said otherwise, support for abstract spatial data types is not included within the default Accumulo packaging.

Geomesa was chosen as it takes a hashing index approach to its spatial indexing, and this afforded an opportunity for a reasonable baseline. Performance evaluations will be performed on the same hardware, identical datasets, and comparable query parameters (i.e.: the input will be logically equivalent, but certain adjustments will be made to account for differences in API implementation).

Hypothesis

While my hope is to create an implementation that yields superior performance during range querying, I expect to find that the numbers of the pivot index implementation will be roughly comparable to the Geomesa performance. Accumulo is designed in such a way that enables users to create a number of custom “iterator” objects that run server-side and accelerate query performance. While my implementation will leverage server-side iteration, developing custom Accumulo code is not the chief objective of this study, and I do not expect to outperform the existing Geomesa framework.

However, I believe that it will be feasible to obtain superior query performance during the kNN querying, as pivot indices enjoy architectural advantages over the grid-based indexing structure (geohashing) employed by Geomesa. In summary, I expect to find comparable results for range querying and improved performance for kNN querying. Provided the desired results are found, I believe this will conform to the guidelines presented in [1] regarding simplicity, scalability, and support of a broad range of data structures. The intent of this study is to demonstrate that superior kNN querying can be achieved using distributed pivot indexing as a result of using an alternative to existing methods for distributed spatial indexing- and that if these results are achieved, an equivalent range querying capacity may be achieved via further exploration of the tools within the Accumulo suite.

CHAPTER 2: LITERATURE REVIEW

Spatial Indexing Practices

Historically, spatial indexing has been divided into two subsets: point access methods and spatial access methods [1]. Point access methods (PAM) perform operations on of a set of objects, *o*. The members of *o* are comprised of both spatial and non-spatial attributes. While the members of *o* exist in two or more dimensions, they are not traditionally conceived of as “spatial” as they do not contain a spatial extent. Spatial access methods (SAM) pertain to datasets where members do indeed have a spatial extent, defined as *o.G*, where *G* is the geometry of a feature within the metric space. While PAM are clearly limited to point data, SAM enable exploration of polyhedra (e.g.: SAM are appropriate for querying polygonal data). Thus, the “point” (does an object exist at this fixed location?) and “range” (what objects are within this user-defined area of interest?) represent two broad types of spatial queries. Traditional, non-spatial datasets reveal two general truths regarding computer indices. The first is that it is computationally beneficial for features to be of uniform size. When applied to spatial datasets, it subsequently becomes prudent to abstract *o.G* to a general shape- enter the Minimum Bounding Box. Secondly, computers are capable of exploring indices in a single dimension. Thus, a function is needed to map multi-dimensional (e.g.: geospatial) data into one-dimensional representations.

A few general guidelines have been suggested in [1] for spatial indexing research. Namely, it is recommended that indices be defined by *simplicity* (in that solutions should avoid unnecessary intricacy and strive to offer robust implementations that support large-scale applications), *scalability* (in that applications should sustain substantial increases in database size), and *support of a broad range of spatial operations* (such as insertion, deletion, retrieval, etc.). Ideally, applied index solutions should avoid special-purpose data structures, instead focusing on abstractions and architectures to implement generic systems. To this end, indices should serve a variety of use cases and transfer across the application domain.

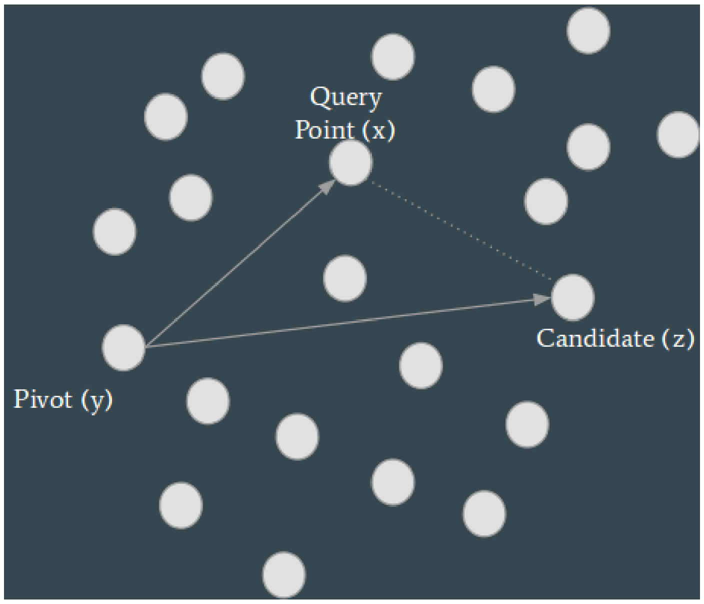


Figure 1: Sample query operation of pivot indexing scheme

Traditional indexing strategies have employed two rudimentary steps: *filter*, then *refine* [2]. Filtering consists of isolating a subset of the database that is germane to a user's query; in the refine step, exact computations are performed to exact a discrete response to the search request. For example, during the filtration step of an intersection query the algorithm may return every object that is represented by a minimum-bounding box that intersects the query window. Subsequently, exact intersection computations are executed during the refine step to establish a definitive result of which objects satisfy the criteria of the input query. The result is the computationally advantageous avoidance of intersection computations of every object in the database.

Pivot Indices

The last several decades of spatial computing indexing have focused on implementation tree-based storages of minimum bounding boxes. Extensions of these strategies include k-d trees, R+-trees, and R*-trees [1]. However, a new paradigm of spatial indexing emerged in 1986 from E. Vidal et al [3], known as the Approximating and Eliminating Search Algorithm. This approach stores pre-computed distances from a subset of the data points to every point in the database. This method introduces a terminology of pivots, where each "pivot" in the data set is used to approximate candidate solutions to nearest neighbor queries via triangle inequality. Thus, the filter step eliminates points that do not satisfy the criteria of the following inequality.

$$d(x,z) \leq d(x,y) + d(y,z)$$

Equation 1: Triangle Inequality

An example of the query process is provided in Figure 1. Upon establishing a query point, the algorithm performs a lookup of the pre-computed distance between the query point and the pivot. Next, the distance from the pivot to each remaining point in the database is retrieved; points that satisfy triangle inequality are deemed “candidates” and included in the output from the heuristic step. Actual distance calculations are performed only on these candidates.

AESA was an iterative approach, intended to improve as the number of iterations increased. The algorithm takes the lower bound of the following function:

$$G(q, t) = \max p \text{ Î } \left| d(q,p) - d(p,t) \right|$$

Equation 2: Approximating and Eliminating Search Algorithm

where *P* is the set of the NN candidates obtained in previous iterations, and *q* (query point), *t* (candidates) ∈ T (database). While this implementation provided auspicious performance over a substantial number of iterations, the spatial indexing scheme struggled during early iterations as there were few members within the set of P [4]. A slight performance in complexity during build time was introduced in 1992 by Mico et. all; AESA with linear preprocessing time and memory requirements (LAESA) [5]. A more significant contribution came from K Figueroa et al. in 2009 in the form of iAESA [6]. In this approach, each candidate maintains a list of previously used pivots sorted by closeness. Subsequently, closer pivots are given preferential treatment. While this offered non-trivial gains with respect to search performance, the obvious tradeoff came in the form of a greater storage complexity in building the index. 2011 marked a seminal contribution the spatial indexing field in the form of PiAESA, a widely-cited implementation that is still demonstrated in modern pivoting research [7]. Socorro et. all improved upon various specifications of AESA in several significant ways. The first was to introduce an alternative dissimilarity function in early iterations to exclude candidates. The AESA algorithm is parameterized to include an additional input value R; until reaching iteration R, the algorithm does not focus on finding best candidate to the nearest neighbor problem. It instead treats objects that most increase the value of the lower bound (e,.g.: improve the heuristic during the approximation step) preferentially. If during R iterations the distance to current NN remains unchanged, the algorithm switches back to AESA.

Several other technical contributions to the spatial indexing field are worth noting, particularly those that relate to pivot selection. Foremost, the efficacy of Random Pivot Selection (RPS) has long-been and continues to be surprisingly high [8]. Thus, RPS serves as an appropriate selection technique in certain use cases and can be used as a null hypothesis is testing the performance of other techniques (e.g.: does this proposed pivot selection technique outperform the implementation when the pivots are placed randomly?).

Outside of RPS, the task of selecting pivots from within the database is traditionally defined as Incremental Outlier Selection Techniques. Selection techniques Maximum of Minimum Distances and Maximum of Sum of Distances; the latter is more widely utilized. Maximum of Sum of Distances selects pivots that are as far away from one another as possible [9]. A more involved approach arrived with the Sparse Spatial Selection technique (SSS) from Pedreira et. all in 2007 [10].

SSS begins with one, arbitrarily selected pivot. Each remaining point is then evaluated for pivot candidacy and selected if distance to any existing pivot is >= a fraction (α) of the maximum distance between objects in the database (M), where α is in |0.35, 0.4|. This is an attempt to ensure that pivots are both located far away from one another (no closer than Mα) and well-distributed in the metric space. Dynamic Pivot Processing is an “on-demand” adaption of the SSS, where the set of pivots can handle growth or contraction of the dataset at an ad-hoc basis [11]. Theoretically, the index will then adapt to insertions or deletions within the database.

Regarding the number of pivots, Brisaboa et. Al. proposed a methodology that has since been widely embraced by the indexing community in [10]. The general concept is that while metric spaces may not necessarily have an explicit dimensionality, they do have an intrinsic dimensionality. This followed the work of Chavez et. all [12] in deriving the intrinsic dimensionality of a data set from the histogram of distances between its data members. Brisaboa asserted that the dimensionality of the data set- and not its size- should dictate the number of pivots. This aphorism has yet to be refuted in relevant literature.

Another variation of the index was the Hybrid Dynamic Spatial Approximation Tree, combining spatial approximation with pivot indexing [13]. This pipeline first detects clusters, within the dataset, assigns pivots based on the existing distribution, then sorts the pivots by SSS.

A notable research shift in 2011, in which Barrientos et. Al. proposed leveraging GPU computing to aid in nearest neighbor queries [14]. This novel approach made use of a GPU card in a single-computer environment to accelerate brute-force search. For a kNN search, each block executes an individual query and threads communicate results via shared memory. This strategy incurs a small penalty, especially when compared to costs associated with storage of query results in global memory. This foray into distributed computing caught my interest, and here begins a key distinction from existing trends in spatial pivot indexing.

This is a notable departure from previous work in the sense that it adapts the software design to the eccentricities of the hardware. While previous spatial pivot indices were cross-platform, theoretical, and structured for general-purpose, this research is an attempt to leverage the design of a specific hardware environment (i.e.: an effort to use the mechanism parallelism provided by a graphics card).

The following year the same research team published research extending pivot indexing to a multi-GPU server [15]. This initiative focused on one of the previous exhaustive-search algorithms and extends it to a 4-GPU platform. The paper explores two basic approaches; a one and two-stage strategy. In the two-stage strategy, a portion of the search was conducted in the traditional CPU. The authors concluded that the performance gains from searching in the graphics card were substantially amortized as a result of the CPU-GPU communication. The results of the one-stage strategy (in which search was conducted entirely within the GPU) were more promising: a superlinear speedup of query performance. However, the amount of shared memory required is proportional to index size, and thus memory costs must be taken into consideration.

CHAPTER 3: SYSTEM DESIGN

Problem Statement

My principle goal is to define a distributed spatial indexing scheme that offers adequate range querying capabilities and improved performance of kNN search. Through pivot indexing, query performance can be accelerated via a heuristic that reduces the number of distance computations necessary to satisfy search parameters. Abstract data types include *points*, a two-dimensional object in the coordinate space, and *pivots*, a subset of the points in a database for which distances to every point in the database are pre-computed during index creation.

The principle data structure used in traditional spatial data indexing is tree-based. In pivot indexing, however, the key construct is a table that stores the distances between pivots and points. In the single node implementation, these values are stored in a Map within each pivot object, and easily retrieved in constant time during query operations.

However, ensuring lookup performance of O(1) becomes more complex in distributed systems, and additional care must be taken to ensure that the retrieval of pre-computed records is rapid. While use of the in-memory Map of the single-node implementation is trivial, queries of Accumulo must be constructed such that table scanning is efficient. The configuration of the cluster, design of the table, and syntax of the query all contribute to performance of retrieval- this is discussed in detail in Chapter 4.

Computer System Architecture

Single-Node Hardware Environment

The hardware used for single-node development and testing consisted of a Dell Precision M4800. This environment included an Intel i7 processor, 32 GB RAM, 1 TB HDD, and 256 GB SSD. This environment was used to verify functionality of the custom indexing scheme prior to introducing the associated overhead of a distributed computing platform.

Distributed Computing Hardware Environment

The cluster consisted of twelve dedicated machines, none of which met current standards for high-end production servers in modern computing environments. The most rigorous hardware consisted of 24 GB of RAM- approximately half of the machines had 4GB, and varied from 500GB to 2TB of secondary memory storage. Processor requirements ranged from dual core i4 to quad core i7, although most consisted of i4. Approximately half of the machines on the cluster consisted of laptops that been reformatted to function as dedicated servers. All nodes ran the Ubuntu Server 14.04 operating system. This setup was located at the USACE ERDC Geospatial Research Lab in Alexandria, VA.

While the hardware offerings of the cluster were not up to standards found in modern server farms, this was an opportunity to put one of the key tenants of the Hadoop methodology to the test- theoretically, significant parallelization via HDFS should be achievable without the use of premium computing resources. While cutting edge results were not expected, the same cluster was able to ingest messages from the Twitter API and store into Accumulo with throughput of approximately 5,500 messages/second.

While these figures are by no means groundbreaking, I took them to suggest that the cluster would serve as an adequate sandbox for testing the custom pivot indexing developed for this project.

Data Introduction

Testing was performed on two types of datasets; synthetic and real world. The synthetic datasets ranged from 5,000 to 750,000 points, randomly generated within the minimum-bounding box of the continental United States. To ensure the performance of the indexing schema was evaluated on a real-life system, two datasets corresponding to real world phenomena were also tested. These collections were used as an attempt to evaluate performance on datasets that included local neighborhoods of both dispersion and clustering; this is an important metric, as real-life data is usually not uniformly distributed. Each dataset can be observed in Figure 2 (the California Roads network dataset) and Figure 3 (the Twitter dataset) below.

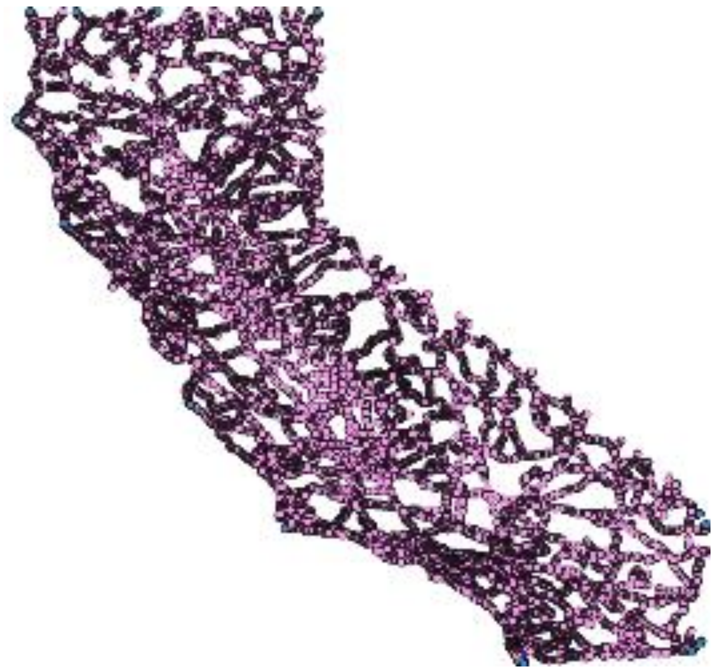


Figure 2: Test dataset consisting of ~21k points over California

The first dataset consisted of 21,048 points, downloaded from the SNAP online portal from Stanford University. This layer represents a road network; in the larger database available from this resource, a graph structure can be obtained which uses points to identify each intersection and undirected edges to depict the roads connecting them. This dataset uses only the intersections.

This layer serves an example of benefit that can be obtained from testing on real-world data. Clustering is observable along the western coast and the area immediately inland of the west coast, while dispersion is present in some of the eastern portions of the state along the Nevadan border.

The size of this layer proved ideal for initial testing of the both the single-node and distributed implementations. Striking a balance between a dataset that does not require significant time to test but also places some degree of strain on a computing platform is a tricky but essential part of the development process.

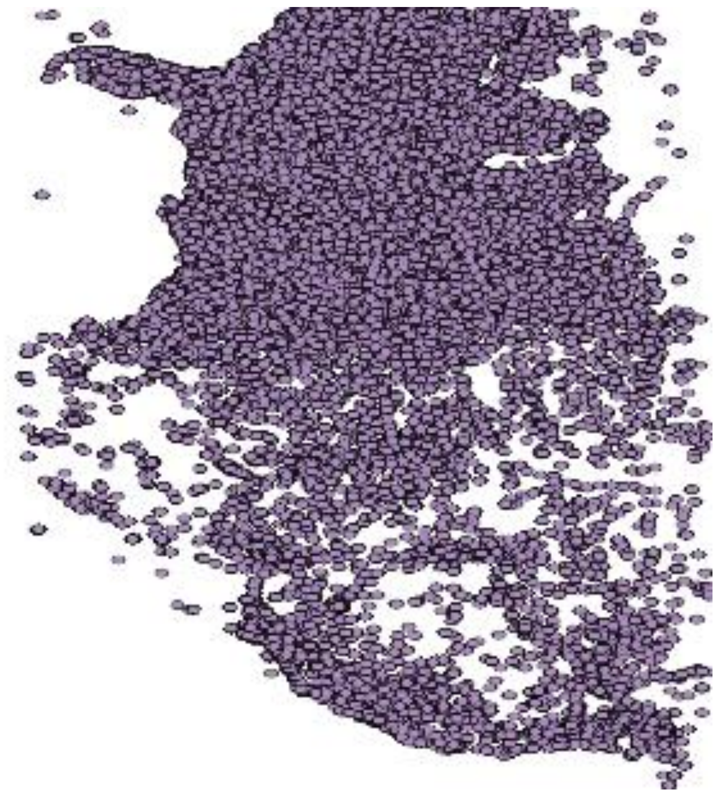


Figure 3: Twitter dataset over Mid-Western American, ~1.1 million points

I selected a second dataset that would serve as the “upper bound” for the single node and distributed implementations. Since the objective of the study was to observe how query performance would scale as the volume of input data increased, this collection proved ideal in analyzing how a particular function within the indexing scheme behaved with a larger dataset.

The Twitter data was obtained from Dr. Dieter Pfoser’s GGS 692 Web GIS Course; it consists of social media messages discussing a particular entertainment topic, and includes 1,132,878 distinct observations. As with the smaller dataset, this collection offers examples of both clustering and dispersion- however, here these phenomena occur at a much larger scale.

External Software Initiatives Leveraged

This study used several existing software libraries. Most prominently, the Apache Hadoop and Apache Accumulo libraries were chosen for their extensive functionality that stores data in a distributed manner. The Apache Storm and Apache Kafka libraries were also used for real-time processing. The Java Topology Suite was used for convex hull computation during index creation. The Google GSON library was also included for serializing data for storage in Accumulo. Apache Maven was used for dependency management.

CHAPTER 4: METHODOLOGY

Overview

The methodology section is intended to reflect the research process as it occurred. Each step in the planning and development process was broken down into stages, characterized by an overarching goal- for example, the objective of Method 2.0 was to move from randomly-generated data to persistent data and begin collection of performance metrics. This is an effort to characterize the perennial nature of the methodology cycle.

$$(q,r)_d = \{u \in U / d(q,u) \leq r\})$$

Equation 3: Range Query: retrieve all the objects u ∈ U within a radius r of the query q

The methodology had five broad goals: implement and verify functionality of the range (1) and kNN (2) queries at the single-node level, implement and test functionality of the range (3) and kNN (4) queries on a distributed cluster of servers, and implement and test range and kNN functionality of Geomesa with the same hardware on the same data. For completeness, formal definitions of range and kNN queries are provided above and below this passage respectively. The following section details the challenges encountered in debugging and optimizing the pivot-indexing scheme for this study, and the methods used to remedy them.

The first step pertains to the initial implementation of the indexing scheme on the single node; the second describes the adjustments made to the scheme for it to use non-random data. The third step details the optimizations of the single-node index creation process, including the use of a convex hull to accelerate maximum distance search. The fourth step adds nearest neighbor search. The fifth step extends the schema to a HDFS/Accumulo platform, and the sixth adds in Apache Storm and Apache Kafka. The seventh and final step describes the final architecture of the distributed pivot index.

$$\{kNN(q) \subseteq U : |kNN(q)| = k, u \in kNN(q), v \in U - kNN(q), d(q,u) \leq d(q,v)$$

Equation 4: kNN Query: Retrieve the k nearest neighbors of point q

Step 1.0: Initial Implementation of Single Node Range Query

As previously stated, the initial development was designed to run on a single-node implementation. In addition to providing a simple platform to verify the underlying logic of the code, this environment was devoid of the many complications associated with distributed computing (e.g.: network latency, system administration, hardware failure, hot-spotting and unequal distribution of data within nodes). At this primitive stage of the iterative process, randomized values were relied on to simplify developmental tasks- to this end, the datasets, query points, and pivots were populated randomly at runtime.

The objective of the first step was to create a Java program that performed preprocessing, index creation, and issued a number of range queries upon each execution. As the points were randomly generated (and the time taken to issue a range query) the performance was not measurable, but this still proved effective in stages of early debugging. The program created the data set, arbitrarily elected pivots, selected a query point from the non-pivot points, issued range query, ran the triangle inequality heuristic, and then finally outputted the resulting points. No information was persisted at any stage of the execution cycle.

A design decision was made to define two parameters to a range query request- an input point, and a radius. This produced a circular object within the metric space, and is semantically similar to the common form of most range queries (in which the question is asked, “what objects fall within the extent of this geometric shape?). The program does not include support for range queries where the input is a polygon or predefined shape- however, this type of input could be converted trivially by calculating the centroid of a given input polygon.

At this stage, I did not collect metrics regarding performance complexity. All operations occurred in memory, and at this step my objective was to ensure functionality of the indexing scheme. To verify the veracity of the output, the dataset and results of the range query were written to Well Known Text files then imported into QGIS for visual inspection. After approximately ten runs, I determined that the program was producing correct output to range query results.

Step 2.0: Implementation of Persistent Data and Profiling

Having verified the functionality of the program at a high level, I now sought to evaluate the efficiency of the system. Because a significant metric of an index’s performance is the speed with which it performs basic computations, I elected to begin profiling the program at various stages. This consisted of timing the execution of various functions within each step of the index creation and querying.

In beginning to profile the performance of the index, I quickly noted the inconsistency of the tests. Since the input was randomized and uniquely generated each runtime, it was impossible to collect authentic statistics on the system. While the performance of tasks varies a trivial number of milliseconds from run to run (particularly depending on the number of applications running on the computer at a given time), the inconsistency of the range query results was made clear- the query point and pivots were consistently different. To remedy this, I began testing on the two datasets detailed in Chapter 3.

Having obtained consisted input, I then began recording performance information on the indexing scheme. The California roads dataset completed index build in average 2215 milliseconds, with 2130 milliseconds spent on pivot selection. The index of the Twitter dataset was completed in an average 7,026,866 milliseconds to complete (nearly two hours), with 6,975,206 milliseconds spent on pivot selection. Range querying was completed in an average 403 milliseconds for the California roads dataset, and 697 milliseconds for the Twitter dataset.

With the results of this preliminary profiling it became immediately clear that pivot selection was consuming an enormous portion of the index construction time- clearly, a scheme that takes nearly two hours to construct an index of one million points will not be widely adopted by the geospatial computing community. To remedy this, I began to revisit the principles of pivot index construction. I also attached additional metrics to isolate the step (or steps) in which the lag was occurring.

Step 3.0: Optimizations of Single-node Index Creation

Recall that the principle of Sparse Spatial Selection dictates that a given point is added to the collection of pivots if it is greater than a fraction of the maximum distance between points away from all existing pivots. This technique is useful in ensuring that good coverage within the database is achieved, and also effective in ensuring that the number of pivots corresponds roughly the intrinsic dimensionality of the dataset. Clearly, the maximum distance between any two points in the datasets is crucial information- but how is this information derived programmatically?

To pinpoint the inefficiency within the index construction, it was essential to review the stages that occur within the build process. The algorithm first obtains the maximum distance between pairs of points. Next, a point is selected at random to be the first pivot. The algorithm then proceeds to iterate over every point in the database, an *O(n)* operation. Points that satisfy SSI are added to the pivot list, then another *O(n)* operation of recording the distance from each pivot to every point in the database occurs.

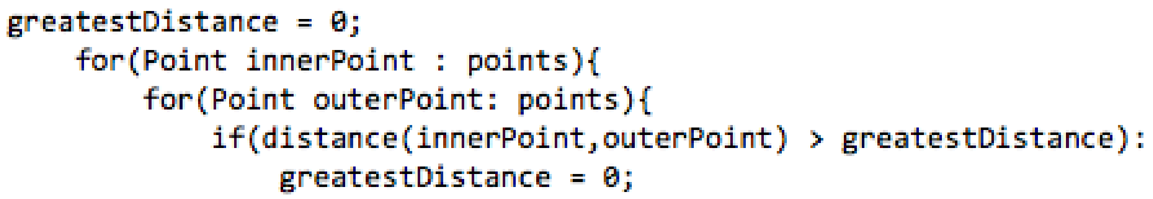
After further profiling it became apparent that the stage responsible for the overwhelming portion of the time spent selecting pivots was the search from maximum distance. After quick analysis of the program, this was unsurprising. The original design went about search for maximum distance in a $O(n^2)$ manner, and while this technique (commonly known as "brute force search") provided a quick and dirty solution it also scales notoriously poorly. Thus, the root cause of the slow construction of the index was that the search for maximum distance was occurring in quadratic time.

At this point, I began to consider additional approaches. Could this maximum distance be derived without examining every point in the database? If so, would the results from this heuristic search be reliable?

Convex Hull

Upon multiple iterations of the test cycle, it became immediately clear that the largest bottleneck within building of the index occurred during search for the greatest distance between two points (this will be explained in-detail in the Methodology section).

To remedy this, I set out to identify heuristics to accelerate the discovery process. The original search for greatest distance was conducted in O(n^2) time; that is, a search was conducted in a nested loop, detailed in pseudo code in Figure 4.



```
greatestDistance = 0;
    for(Point innerPoint : points){
        for(Point outerPoint: points){
            if(distance(innerPoint,outerPoint) > greatestDistance):
                greatestDistance = 0;
```

Figure 4: Logic of original maximum distance search

As anticipated, this calculus scaled poorly. Over 10 attempts on to determine the greatest distance between any two points on a dataset containing ~1.13 million points, the calculation to determine the greatest distance took an average 2.05 hours to complete. Indisputably, this is inacceptable for creation of an index.

Subsequently, I set out to identify heuristic strategies to optimize this aspect of index creation. After review of several methodologies, the inner-workings of the convex hull proved substantive. The convex hull algorithm attempts to find the most minimal set that contains every point within a given collection [10]; in layman's terms, this is an algorithmic solution to determine the boundary points of a given dataset.

Having derived the boundary (i.e.: the most extreme) points within a collection, it follows that the greatest distance within a collection occurs within two boundary points. Thus, the algorithm charged with determining the greatest distance between two points will benefit from excluding points that lie within the center of the collection.

This optimization enables the search algorithm to approach greatest distance computations via a divide-and-conquer strategy. Traditional implementations of convex hull search begin by partitioning the data points into "upper" and "lower" segments (in two-dimensional data, this occurs by examining the midpoint of the 'y' values). After determining two distinct segments, a thread is charged with identifying every possible combination of triangles. Any point that is not contained within a triangle is subsequently labeled as a point lying within the convex hull- that is, the algorithm assumes that any point that cannot be triangulated must be on the outermost layer of the data.

Of course, additional synchronicity of the search may be achieved trivially; the metric space may be subdivided recursively, with each partition receiving a designated thread charged with performing distance computations for each pair.

As previously stated, the solution to this problem was found in the use of convex hull- that is, a polyline that encloses every point in a dataset. The greatest distance in any dataset should occur between two boundary points, and thus, this approach creates reliable output without having to compute every point in the database. Knowing that any existing solution for a convex hull would likely require adjustment to comply with my custom data types (e.g.: points, pivots), I sought to find an open source implementation that could be easily adapted. After considering several candidates, I settled on an implementation from the website accompanying the text of an O'Reilly text on Java algorithms [43]. This approach attempted to form a triangle around every point in a dataset, and any point that could not be enclosed within a triangle is treated as a boundary point.

This approach was also highly parallelizable. The algorithm begins by partitioning the data plane, and from this point it is trivial to assign a thread to each partition and perform a local search in synchronization. The search process concludes with a joining of each search thread. I expected this approach to both reduce the time spent on maximum distance search and to also return a high recall (i.e.: not report any false negatives).

The algorithm met both of my criteria, handily. The search improved from 2,130 milliseconds to 805 milliseconds on the California Roads datasets and from 6,975,206 milliseconds to 1,238,918 milliseconds in the Twitter dataset- improvements of 62% and 82% respectively. By all numeric accounts, this was an unqualified success. However, loading the newly created convex hull datasets into a GIS editor provided a story that only visual confirmation could tell.

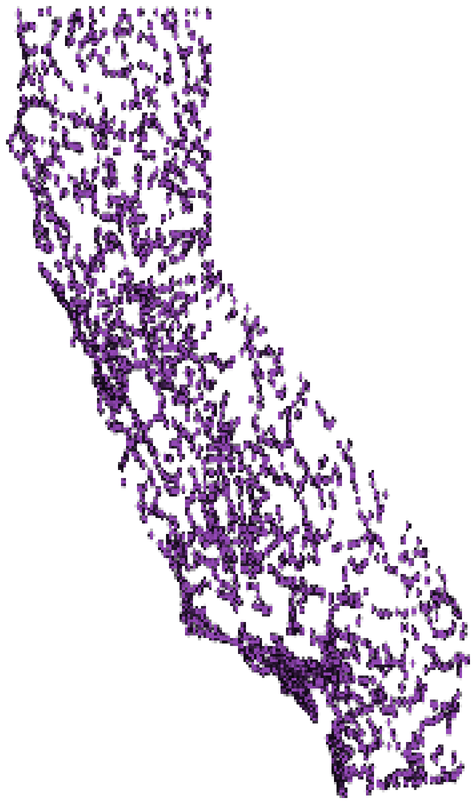


Figure 5: Output of Erroneous Convex Hull Output

There is no need to scroll to compare this output (Figure 5, output of the erroneous convex hull operation) to the original California Roads dataset; the algorithm reduced the number of points from 21,048 to approximately 5,000, but there are still a significant number of interior points that have been incorrectly left within the bounds of the boundary. This incorrect identification of non-boundary points effectively just performs a kind of "thinning" on the datasets- which may have utility in different spatial applications, but is clearly not the intended purpose in this scenario. My expectations had been incomplete- the results satisfied my wish of high recall, but also offered incredibly poor precision.

I then moved on to another implementation- this one in an existing open-source package, the Java Topology Suite. The JTS convex hull functionality required that input data be in the format of abstract data types within the JTS library- however, the “Point” abstraction from JTS is logically comparable to the one from my project, and this conversion was performed trivially. I began testing with the expectations of speed, recall, and now also precision.

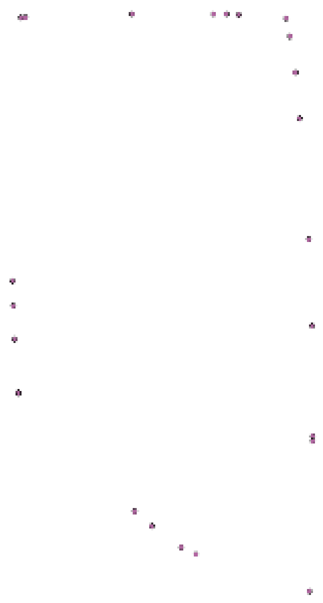


Figure 6: Correctly generated convex hull

The performance improvements were indisputable. Search within the California Roads dataset was now completed in 73 milliseconds, and 245 milliseconds for the Twitter dataset.

Further, upon visual inspection, the convex hulls appeared to be highly precise-now, points were included in the output if and only if they resided in the outermost boundary. Figure 6 visualizes correct output of the convex hull operation performed on the Twitter dataset- only the border points remain. Having made substantial improvements the iteration was concluded. On an average of five runs the indices for the California roads and Twitter datasets concluded in 96 milliseconds and 991 milliseconds respectively.

Step 4.0: Implementation of Single-node kNN Search

In the final stage of development for the single-node purposes, I sought to examine the flow of the existing range query and to implement kNN workflow. To understand this process, I began by revisiting the underpinnings of the range query in a pivot-indexing schema.

Once a query point is submitted range query algorithm, the first step is to identify the pivot to which the point is closet. This is done with the assumption that pivots are subject to Tobler's First Law of Geography- theoretically, a point is more closely related to a pivot that is close to than pivots that are farther away. This calculation inserts pivots into a priority queue, where priority is distance to the query point.

After pruning points by triangle inequality, actual distance computations are performed in the refine step. A tree set (a sorted collection that does not allow duplicates) is used to store candidate notes that satisfy the range criteria (made possible by the comparator defined in the Software Engineering Definitions section). The tree set can then be iterated by the calling application, which will be in ascending order of points closet to the query point.

In selection of a kNN point from within the pivot indexing literature, I was drawn to the simplicity of an approach that relied on issuing successive range queries (i.e.: range queries are performed with increasingly large radii until the requisite number of neighbors are discovered) [14]. This was perhaps the most common kNN strategy that appeared in the literature, and seemed accommodating to the scope of this study. However, the crucial aspect of the index construction lies in the question- how does one determine the size of the radius for the initial range query? Ideally, this range should be one has an acceptably highly probability of returning all or the majority of the requisite number of neighbors- however, the range should also not be so large that it performs an oversized query that indiscriminately tests points in the database unnecessarily.

In effect, the answer is to pursue localized information of density. Said in a different way, the question being asked here is “given the average density of the dataset, how large of a circle would I need to draw in order to discover *k* neighbors?” To answer this question, we can begin by identifying a subset of the dataset that the query point belongs to. Provided we have some baseline density information for the local area of which the query point is a resident, it can be said that the value of the initial range query can be articulated as $k = D x A$, where D is local density and A is the area of the local polygon. Since the area is circular, the formula can be expressed in the following terms:

$$k = density \times \pi \times r^2$$

Equation 5: Density formula for original radius value of kNN search

Obtaining density information can be possible at two disparate times- during index build, or during query time. Obviously, since this is an operation that may require traversal through a large number of points, the design was made to alter the index creation process to include support for ascertaining local information.

The process begins as an extension of convex hull generation. Since the program now uses Java Topology Suite, it is possible to determine the minimum bounding box (and centroid of the minimum bounding box) with two additional function calls from the client application. From this information, the minimum-bounding box of the dataset can be easily calculated, and the minimum bounding box can be segmented into four quadrants (the area of each quadrant is also stored). Then, during the loop for pivot candidacy, a point in polygon query is performed to determine which quadrant the point falls in. The totals for each quadrant are calculated at the end of the loop, and with that localized density information is available for handling of incoming nearest neighbor queries.

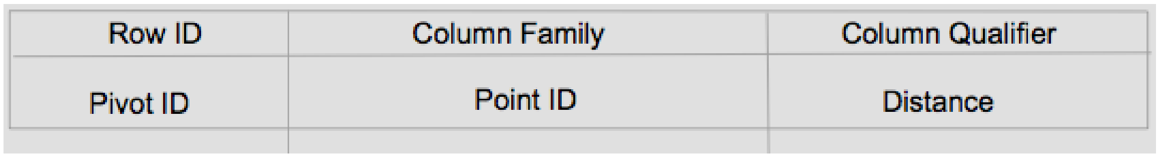
During the query, a simple point in polygon query is performed to determine the quadrant to which the point belongs. Then, since values of k and A are already known, density information can be quickly derived in a dynamic way.

In an average of five runs, kNN queries (where k = 5% of the entries in the database) on the California roads dataset and Twitter dataset ran in 124 milliseconds and 10,860 milliseconds respectively.

Step 5.0: Implementation of Distributed Range Query

Having achieved functionality of range and kNN querying at the single node level, the next objective was to implement a functional range query on a distributed platform. While the core logic of the pivot processing was expected to remain constant, a number of modifications would need to be made to account for the additional complexity of introducing additional software to the index structure.

At this stage, the two major software components added to the pivot indexing code base were Apache Hadoop and Apache Accumulo. Since Accumulo handles much of the distribution of records amongst the Hadoop File System "under the hood," my code interacted exclusively with Accumulo. Figure 7 illustrates the Accumulo table structure used for the pivot table in this implementation.



Row ID	Column Family	Column Qualifier
Pivot ID	Point ID	Distance

Figure 7: Table design in Accumulo

The interface presented to the developer by Accumulo is tabular- while much of the traditional functionality of a relational database is absent, some similarities remain. A key distinction is that records within Accumulo tables can have varying numbers of attributes. That is, while a SQL table ensures that all records within a table conform to a fixed dimensionality (and have NULL values for any missing attributes), Accumulo does not enforce this. This behavior is expressed as the “wide-column” characteristic of such data stores.

Accumulo also does not require that row IDs be unique- instead, a row must be unique by some combination its row ID, column family, column qualifier, time stamp (useful in versioning), or visibility (a label denoting the requisite authorizations for viewing a record). Since privacy and versioning were not germane to this study, only row ID, column family, and column qualifier were used for establishing unique identifiers.

The row ID design (above) was used to emulate the pivot table used in prior studies. The value (the actual distance between a given point and a pivot) was stored in the column qualifier field. This decision was made because Accumulo offers a number of server-side iterators that do not consider the value field.

The basic flow of the system is as follows. Once the values are read from a file into Accumulo storage, pivot selection and pivot mapping occur. Since a large number of pivot map entries would be created (*n* x *r*, where *n* is the number of pivots and *r* is the number of records), a priority item was to ensure that hotspotting did not occur within the cluster. That is, I expected values not to be stored in a distributed fashion as they all began with the same prefix (instead of being uniformly distributed across each node, records would be concentrated on one or two servers). This is because Accumulo builds a binary search tree index used for lookups based off the row ID- thus, when records all begin with the same values and are not split uniformly across the cluster, lookup performance can be significantly degraded.

To address this, I implemented a hashing algorithm when storing the row ID. Before writing the record to Accumulo the values were run through a one-way hashing process- lookup could be achieved by running query input through the same procedure. Due to this hashing, records were now being stored evenly across the cluster and “split” in a uniform manner. However, this alteration also slowed the speed of index creation considerably- build time for the index now took an average of ~80,000 milliseconds longer to complete for the 21,000 point dataset. The hashing implementation I originally used was intended to create encrypted passwords for database security, and in an effort to avoid collisions ran each hashed value through 100 iterations- hence, it was incredibly slow. As this delay was obviously unacceptable, I removed the hashing. However, this prompted search of additional optimizations.

The largest bottleneck I discovered was in index creation process when writing to the table. At first, writes were converted to an Accumulo mutation object (an action that manipulates a row in a table- either creating a new one or overwriting an existing one), stored in an Array List, and then written once. However, this approach did not scale well- the underlying assumption was that the mutation object could support the entire dataset, and as input dataset grew above ~60,000 the Java Virtual Machine began to throw out of memory exceptions.

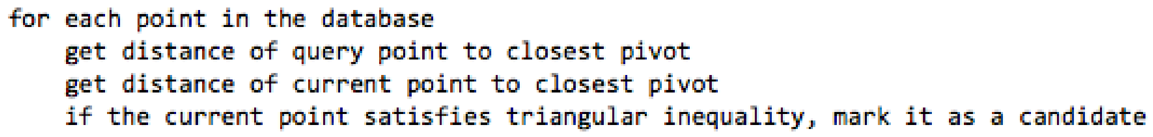
To remedy this, I first began writing once per record. However, this introduced significant overhead and drove the I/O costs of index creation up undesirably. Creation of the California Roads dataset increased from 1645 milliseconds (from writing once) to 9251 milliseconds (writing once per record- 21,048 writes).

After some experimentation, I ultimately adopted an approach to flush the writer every 50,000 entries. This presented itself as a compromise between not writing too frequently and managing concerns of primary memory. The mutation list appears to always support ~50,000 items without exception, and the cost of connecting to the database and network transmission were minimized.

During this process I also noted improved performance from opening and closing a new Accumulo writer object (as opposed to creating a single writer object at the top of the loop, flushing periodically, then closing the writer once). The writing process occurred in approximately 3,000-3,200 milliseconds when opening and closing the writer object repeatedly- when using a single writer object and closing once, writing took 3,500 – 3,900 milliseconds. From the official Accumulo documentation, my best guess was that closing the writer releases resources that enable the JVM to function more efficiently.

Having improved upon initial writing time, I proceeded with testing and evaluating performance of the range query functionality. The first attempt at a range query of the California Roads completed in 123,628 milliseconds (2.06 minutes)- of that interval, 122896 milliseconds (2.04 minutes) of were spent on the heuristic step. Running on the Twitter dataset, the heuristic step completed in 2,947,904 milliseconds (49.13 minutes).

At this point it became necessary to evaluate the performance of the heuristic step, as this stage has accounted for 99.03% of the range query processing time. At this iteration, the heuristic step performed three operations during its loop over every point in the database. This process is represented by the pseudo-code shown in Figure 8.



```
for each point in the database
    get distance of query point to closest pivot
    get distance of current point to closest pivot
    if the current point satisfies triangular inequality, mark it as a candidate
```

Figure 8: Pseudo-code demonstrating first iteration of distributed heuristic step

The process of obtaining the distances in the first two lines is a simple lookup, as both distances are pre-computed during index build. However, since the number of pivots roughly corresponds to the dimensionality of the input dataset, and since I used two-dimensional data for testing, it can be assumed that the distance from the query point to each of the pivots can be done before entering the loop and held in memory. Thus, this reduced the number of lookups roughly in half, and produced the expected results. Range queries for the California Roads dataset improved from 123,68 milliseconds to 62,615 milliseconds- an improvement of approximately 50%.

This improvement was significant, however the runtime was still unacceptably high (1.04 minutes). Accumulo offers several pre-made mechanisms that run on the server and can be used to accelerate search- *iterators* and *filters*, the latter of which serves as a kind of iterator with limited functionality. Iterators are equipped with the ability to "seek" through a set of records (instead of loop through a database one point at a time, an iterator can "peek" into future records in hopes of intelligently guessing on which records to avoid checking unnecessarily). The base classes of the Filter and Iterator are both abstract, but several concrete examples are offered within the default Accumulo package.

I began by writing my own filter. This filter extended one of the existing examples, and included a "matches" function designed to exclude records that did not contain matching column family and column qualifier information. The process of attaching a filter or iterator to a scanner object is trivial, as the scanner class includes a function to add either type of object by defining the type of iterator (e.g.: MyFilter.class) and at integer representing priority. Accumulo iterators are designed to be “chained together” to accelerate search (i.e.: one iterator may search through a column family, while a second iterator searches through column qualifiers), while the priority parameters denote the sequence in which iterators should be applied.

Unfortunately, my attempt at using a custom filter to improve lookup performance from the query table proved detrimental. The runtime of the heuristic step increased from 62,615 milliseconds to 65,000 milliseconds.

Next, I leveraged functionality included in the administrative shell for Accumulo- bloom filters. The bloom filters included in the default package correspond to a Boolean flag that can be activated with a single command from the administrative interface. As data is written to the data store, Accumulo maintains metadata describing the location of the data with respect to the memory address. When this flag is set to true, the database loads the metadata into memory- a step that can theoretically boost the speed of lookups.

Activating the Bloom Filter on the pivot table did improve performance slightly- the heuristic step runtime decreased from 62,615 milliseconds to 59,420 milliseconds- but still did not produce the desired results. I next added manual splits in the pivot table- that is, explicitly defined the range at which the pivot table should begin to send records to other nodes in the cluster (e.g.: rows beginning with the prefix “pivot_0” through “pivot_3” should be stored on server 1, rows beginning with the “pivot_4” through “pivot_6” should be sent to server 2). The hope was that a better distribution might improve runtime performance. This yielded a further decrease of runtime from 59,420 milliseconds to 52,713 milliseconds.

After researching good practice for writing Accumulo queries, I discovered that the approach I was taking to obtain the lookup of the pre-computed distances was incorrect. As the official Accumulo documentation states, "*Lookups can then be done by scanning the Index Table first for occurrences of the desired values in the columns specified, which returns a list of row ID from the main table. These can then be used to retrieve each matching record, in their entirety, or a subset of their columns, from the Main Table.*" In plain English: for a given query, it is necessary to specify a beginning and end range of row IDs for the scanner to examine. In the case of an exact lookup, the row ID should be provided as both the start and the end value. In failing to provide this, the scanner object was continuing to process additional records after discovering the row ID in question.

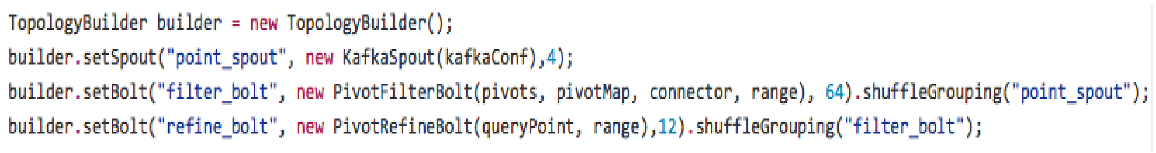
With the knowledge of how to query the database in a way that would more likely produce results in constant time, I modified the scanner input accordingly. The heuristic runtime improved from 52,713 milliseconds to 22,577 milliseconds (22.57 seconds).

Method 6.0: Implementation of Range Query as Storm Topology

The inefficiencies of the initial range query had been improved from 2.02 minutes to 22.57 seconds. While these gains were significant, an indexing scheme that responded to range queries in more than 20 seconds would be far from pragmatic. Thus, having made improvements to the server side of the query performance, I set out to implement parallelization of the client.

Apache Storm is a distributed computation system that enables high throughout for real-time applications. The core abstraction of the Storm framework is the "*tuple*"-

that is, a record or message that is passed around a network “*topology”* comprised of “*bolts*.” A bolt is a user-defined class that enables the user to dictate what actions should be taken with a given tuple (e.g.: determine if the tuple should be emitted to another bolt, modified, written to a database etc.). “*Spouts”* are responsible for introducing new tuples into the topology. Thus, a spout is always the entry point for a collection of tuples, and a spout is connected to one or more bolts. Apache Storm was chosen as it offers the opportunity for substantial parallelization in a highly abstracted way.



```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("point_spout", new KafkaSpout(kafkaConf),4);
builder.setBolt("filter_bolt", new PivotFilterBolt(pivots, pivotMap, connector, range), 64).shuffleGrouping("point_spout");
builder.setBolt("refine_bolt", new PivotRefineBolt(queryPoint, range),12).shuffleGrouping("filter_bolt");
```

Figure 9: Code example illustrating parallelization management in Apache Storm

Figure 9 demonstrates the brevity of configuration process for parallelization in Storm- in four lines, the topology used for range querying in this application was constructed and the “parallelism hint” (integer value in each of the last three lines) refers to the number of tasks that should be run by a given thread on an application. Storm is a distributed framework, and a natural fit for the central themes of this study.

Apache Kafka is a distributed queuing system that is frequently paired with Storm. Kafka queues (referred to as “topics”) enable high throughput, and are effective at offering new data into a network for consumption- needless to say, they are ideal for use as “spouts” in Storm topologies.

The key objective of this method was to create a storm Topology that allowed for significant parallelism during the range query process. This action was distinct from the index creation process. The workflow begins with the receipt of a range query request; every point in the database is held in a Kafka queue, then released into the storm topology at the beginning of the query. From the Kafka spout, every tuple (point) arrives at a Filter Bolt, where points are evaluated for triangle inequality. Tuples that fail the check are dropped from the network; tuples that pass are passed onto a Refine Bolt, where the actual distance computation is performed. Figure 10 illustrates the final architecture of the implementation.

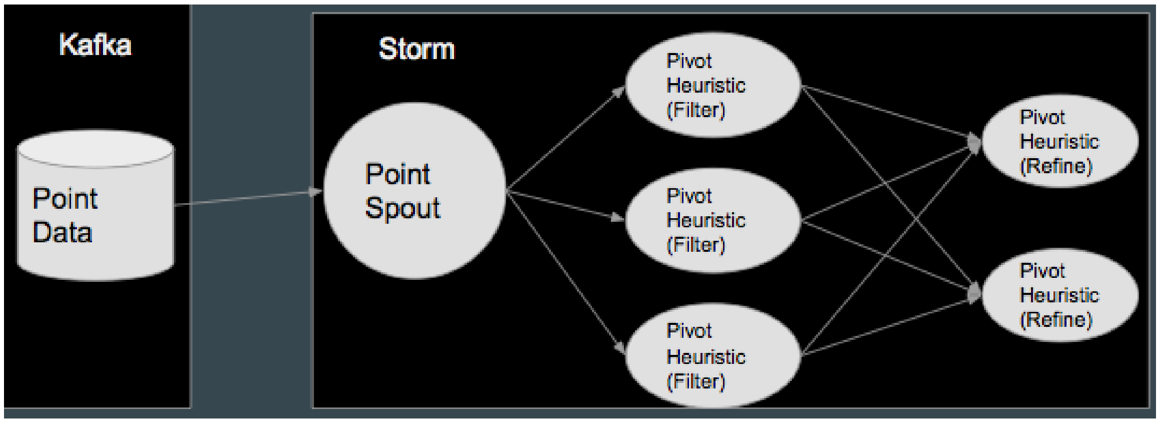


Figure 10: Final Architecture of Range Query

Effectively, this process converted the pivot range query workflow into a Storm topology. The results were favorable; the range query of the California Roads dataset now ran in 954 milliseconds, and the Twitter dataset in 5994 milliseconds (5.99 seconds). This was the final iteration devoted to range querying in this study.

Method 7.0: Implementation of Distributed kNN Search

The final step was to implement distributed kNN querying. The indexing results were already in place to support density information, and the only alteration was to supply the topology with information regarding the requisite number of neighbors.

The biggest modification occurred within the topology. In the range query model, the workflow ended upon reaching the refine bolt. However, as the kNN queries consist of successive range queries of increasing radius values, I connected the refine bolt back to the filter bolt. In this way, tuples remain in the topology until *k* nearest neighbors are discovered. When *k* is realized, a shutdown of the topology occurs programmatically.

The additional data required to run nearest neighbor queries was entered into the network primarily with static variables. A counter was introduced to increment the range query with respect to the number of times a tuple had been processed. The kNN query executed over the California Roads and Twitter datasets in 1,903 milliseconds and 20,574 milliseconds respectively.

CHAPTER 5: RESULTS AND ANALYSIS

As expected, the distributed range query performance did not exceed that of Geomesa. The execution time of the range query in the pivot index ranged from 738 milliseconds (for the smallest dataset, 5,000 randomly generated points) to 5,994 milliseconds (largest dataset, 1.1 million points). Ultimately, the range query completed is roughly six seconds. While this is a vast improvement from the original 49 minutes, the Geomesa range query fared better as the dataset scaled. The Geomesa range query was completed in 381 milliseconds for the 5,000-point dataset and 1,081 milliseconds for the Twitter dataset. Ultimately, Geomesa outperformed the pivoting index by approximately 5 seconds at the largest scale.

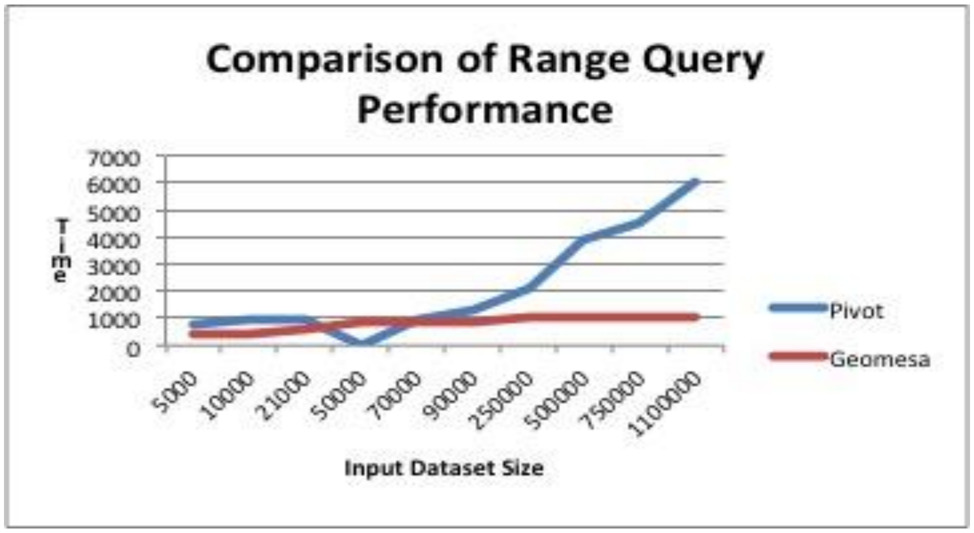


Figure 11: Range Query Performance

However, also as expected, the pivoting approach outperformed Geomesa on kNN search. The pivoting execution time ranged from 1,457 milliseconds to 20,574 milliseconds; the Geomesa kNN queries ranged from 4,113 milliseconds to 227,279 milliseconds. At the largest scale, the gap between nearest neighbor search was approximately 3.4 minutes. Figure 11 illustrates a graphical comparison of the performance of this implementation against Geomesa.

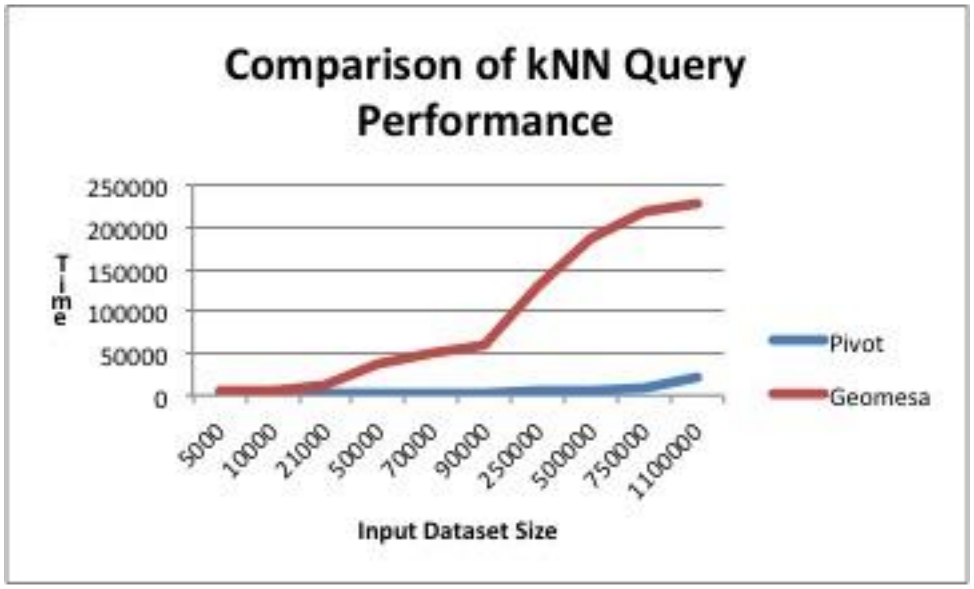


Figure 12: kNN Query Performance Chart

While Geomesa outperformed the pivoting with respect to range querying capabilities, I would argue that the results of the two indicate a comparable methodology. Five seconds is fairly trivial at datasets of approximately one million points. However, the gap between the two approaches in nearest neighbor querying is substantial. Ultimately, the results that I expected to see from the outset of the study were confirmed by these figures. Figure 12 illustrates a graphical comparison of the performance of this implementation alongside Geomesa for nearest neighbor search.

CHAPTER 6: CONCLUSION AND FUTURE WORK

While the performance of the nearest neighbor search for the pivot-indexing scheme was an improvement, it should be noted that Geomesa is a robust software package that offers a wide breadth of functionality. The indexing scheme developed for this study dealt exclusively with two-dimensional coordinate information. Geomesa includes support for textual search, temporal querying, and a number of other features. Thus, while the kNN performance is an improvement, the pivot indexing presented here does not represent an improvement in terms of offering additional services through the Apache Accumulo framework.

However, the objective of this study was to demonstrate that pivot indexing could function at a high level in distributed computing environments. Further refinement of pivot indexing could make it increasingly competitive with Geomesa and other spatial indexing platforms that create spatial data structures within the Accumulo platform (such as Geowave).

As documented in Chapter 5, I learned valuable lessons regarding the sensitivity of record lookups in Accumulo. An exact row ID must be provided as both the start and end of the query input; otherwise the lookup will suffer extreme inefficiency. Additional measures such as Bloom Filters and manually splitting tables on row ID may yield significant performance gains. The greatest potential for query acceleration lies in development of custom Accumulo iterators; these custom classes are designed to be "chained together," and this could be of great use for both nearest neighbor and range search for spatial data.

Relevant future work may include exploration of GPU processing to accelerate indexing; also, an approach oriented around custom MapReduce queries at the Hadoop File System level could qualify as a novel solution. The implementation referenced in this study relies on Apache Accumulo, a software suite that sits atop of HDFS. An implementation that functioned purely in Hadoop (such as the Spatial Hadoop framework put forth by the Computer Science Department at the University of Minnesota) could produce favorable results, as this would forgo the associated overhead of Accumulo.

REFERENCES

[1]Volker Gaede , Oliver Günther, Multidimensional access methods, ACM Computing Surveys (CSUR), v.30 n.2, p.170-231, June 1998 [doi>10.1145/280277.280279][2]Park, H. H., Lee, C. G., Lee, Y. J., & Chung, C. W. (1999). Early separation of filter and refinement steps in spatial query optimization. In Database Systems for Advanced Applications, 1999. Proceedings., 6th International Conference on (pp. 161-168). IEEE.

[3]Enrique Vidal Ruiz, An algorithm for finding nearest neighbours in (approximately) constant average time, Pattern Recognition Letters, Volume 4, Issue 3, 1986, Pages 145-157, ISSN 0167-8655, http://dx.doi.org/10.1016/0167-8655(86)90013-9.
(http://www.sciencedirect.com/science/article/pii/0167865586900139)

[4]Raisa Socorro, Luisa Micó, Jose Oncina, A fast pivot-based indexing algorithm for metric spaces, Pattern Recognition Letters, Volume 32, Issue 11, 1 August 2011, Pages 1511–1516

[5]María Luisa Micó, José Oncina, and Enrique Vidal. 1994. A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. Pattern Recogn. Lett. 15, 1 (January 1994), 9-17. DOI=http://dx.doi.org/10.1016/0167-8655(94)90095-7

[6]Karina Figueroa, Edgar Chavez, Gonzalo Navarro, and Rodrigo Paredes. 2010. Speeding up spatial approximation search in metric spaces. J. Exp. Algorithmics 14, Article 6 (January 2010), .61 pages. DOI=http://dx.doi.org/10.1145/1498698.1564506

[7]Raisa Socorro, Luisa Micó, Jose Oncina, A fast pivot-based indexing algorithm for metric spaces, Pattern Recognition Letters, Volume 32, Issue 11, 1 August 2011, Pages 1511–1516

[8]László Kovács. 2012. Reduction of distance computations in selection of pivot elements for balanced GHT structure. In Proceedings of the 8th international conference on Machine Learning and Data Mining in Pattern Recognition (MLDM'12), Petra Perner (Ed.). Springer-Verlag, Berlin, Heidelberg, 50-62. DOI=http://dx.doi.org/10.1007/978-3-642-31537-4_5

[9]Socorro R. ,Mico L. , Oncina J. : Efficient search supporting several similarity queries by reordering pivots, Proc. of the IAESTED Signal Processing, Pattern Recognition and Applications, pp. 114-141, 2011

[10]O. Pedreira and N. R. Brisaboa. Spatial selection of sparse pivots for similarity search in metric spaces. In SOFSEM 2007: 33rd Conference on Current Trends in Theory and Practice of Computer Science, LNCS (4362), pages 434-445, 2007.

[11]Salvetti, M., Deco, C., Reyes, N., & Bender, C. (2011). Adaptive and dynamic pivot selection for similarity search. Journal of Information and Data Management, 2(1), 27.

[12] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza- Yates, and José Luis Marroquín. Searching in metric spaces. ACM Computing Surveys , 33(3):273-321, September 2001.

[13] Diego Arroyuelo, Francisca Muñoz, Gonzalo Navarro, and Nora Reyes. Memory-adaptative dynamic spatial approximation trees. In Proceedings of the 10th International Symposium on String Processing and Information Retrieval, SPIRE, 2003.

[14] Ricardo J. Barrientos , José I. Gómez , Christian Tenllado , Manuel Prieto Matias , Mauricio Marin, kNN query processing in metric spaces using GPUs, Proceedings of the 17th international conference on Parallel processing, August 29-September 02, 2011, Bordeaux, France

[15] Barrientos, R.; Gomez, J.I.; Tenllado, C.; Matias, M.P.; Marin, M., "Range Query Processing in a Multi-GPU Environment," in Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on , vol., no., pp.419-426, 10-13 July 2012

[16] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop GIS: a high performance spatial data warehousing system over mapreduce. Proc. VLDB Endow. 6, 11 (August 2013), 1009-1020. DOI=http://dx.doi.org/10.14778/2536222.2536227

[17] Ablimit Aji, George Teodoro and Fusheng Wang: Haggis: Turbo Charge A MapReduce based Spatial Data Warehousing System with GPU Engine. To Appear in Proc. of the Third ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial-2014), Nov 4, 2014, Dallas, TX, USA.

[18] Yunqin Zhong, Xiaomin Zhu, and Jinyun Fang. 2012. Elastic and effective spatio-temporal query processing scheme on Hadoop. In Proceedings of the 1st ACM

SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial '12). ACM, New York, NY, USA, 33-42. DOI=http://dx.doi.org/10.1145/2447481.2447486

[19] Bingsheng Wang , Haili Dong , Arnold P. Boedihardjo , Chang-Tien Lu , Harland Yu , Ing-Ray Chen , Jing Dai, An integrated framework for spatio-temporal-textual search and mining, Proceedings of the 20th International Conference on Advances in Geographic Information Systems, November 06-09, 2012, Redondo Beach, California [doi>10.1145/2424321.2424418]

[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26, 2, Article 4 (June 2008), 26 pages. DOI=http://dx.doi.org/10.1145/1365815.1365816

[22] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107-113. DOI=http://dx.doi.org/10.1145/1327452.1327492

[23] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. 1996. The quickhull algorithm for convex hulls. ACM Trans. Math. Softw. 22, 4 (December 1996), 469-483. DOI=http://dx.doi.org/10.1145/235815.235821

[24] George Heineman, Gary Pollice, and Stanley Selkow. 2008. *Algorithms in a Nutshell*. O'Reilly Media, Inc..

BIOGRAPHY

Kevin Tyler graduated from Forest Park High School, Woodbridge, Virginia, in 2009, and received his Bachelor of Science from Mary Washington University in 2012. He was employed by the Society for Worldwide Interbank Financial Telecommunication from 2013-2014, Booz Allen Hamilton from 2014-2015, and has been with the US Army Corps of Engineers Geospatial Research Lab since 2015. He will begin pursuit of a PhD of Computer Science at George Washington University in the fall of 2016.