Pivot-based Data Partitioning for Distributed k Nearest Neighbor Mining

by

Caitlin Anne Kuhlman

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

January 2017

APPROVED:

_____

Professor Elke A. Rundensteiner, Thesis Advisor

_____

Professor George Heineman, Thesis Reader

_____

Professor Craig Wills, Department Head

Abstract

This thesis addresses the need for a scalable distributed solution for k-nearest-neighbor (kNN) search, a fundamental data mining task. This unsupervised method poses particular challenges on shared-nothing distributed architectures, where global information about the dataset is not available to individual machines. The distance to search for neighbors is not known a priori, and therefore a dynamic data partitioning strategy is required to guarantee that exact kNN can be found autonomously on each machine. Pivot-based partitioning has been shown to facilitate bounding of partitions, however state-of-the-art methods suffer from prohibitive data duplication (upwards of 20x the size of the dataset). In this work an innovative method for solving exact distributed kNN search called PkNN is presented. The key idea is to perform computation over several rounds, leveraging pivot-based data partitioning at each stage. Aggressive data-driven bounds limit communication costs, and a number of optimizations are designed for efficient computation. Experimental study on large real-world data (over 1 billion points) compares PkNN to the state-of-the-art distributed solution, demonstrating that the benefits of additional stages of computation in the PkNN method heavily outweigh the added I/O overhead. PkNN achieves a data duplication rate close to 1, significant speedup over previous solutions, and scales effectively in data cardinality and dimension. PkNN can facilitate distributed solutions to other unsupervised learning methods which rely on kNN search as a critical building block. As one example, a distributed framework for the Local Outlier Factor (LOF) algorithm is given. Testing on large real-world and synthetic data with varying characteristics measures the scalability of PkNN and the distributed LOF framework in data size and dimensionality.

Acknowledgements

I would first like to express my gratitude to my Thesis Advisor Professor Elke Rundensteiner for her guidance and encouragement. She stuck with me through the process of tackling this problem - and it was a long process. Despite setbacks, we persevered to complete a work I feel very proud of. As a result of her guidance I have developed my skills as a researcher and learned a lot about collaboration, dedication and grit. She sets an inspirational example to aspire to.

I would also like to sincerely thank my collaborators Lei Cao and Yizhou Yang. Lei's experience and guidance were invaluable in the process of developing these techniques, and his previous research laid the foundation of this line of inquiry. Without Yizhou's tireless work no progress would have been made, and I am so appreciative of her help and collaboration. The outlined approach for scalable outlier detection was developed with them, and has led to exciting other research on this topic.

I'm also grateful to my Thesis Reader Professor George Heineman for his advice and patient support.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Detecting the nearest-neighbors of all points in a dataset is a ubiquitous task across numerous data mining endeavors. Many techniques, including classification and regression [1], clustering [2, 3], and outlier detection [4, 5, 6] require a k-nearest-neighbor (kNN) search to be performed for every point in the dataset. The usefulness of this task (also known as the all kNN query [7], similarity search [8], or kNN join [9]) has prompted much research. Furthermore, real-world applications increasingly rely on kNN search over very large datasets, requiring resources beyond what is feasible on a single machine. In such cases, distributed compute clusters enable analysis of huge datasets through parallel processing across many machines. Therefore, the development of highly distributed solutions for kNN search is no longer an option, but a necessity.

A distributed solution for kNN search can facilitate high value applications. This thesis demonstrates this for one method, the Local Outlier Factor algorithm (LOF) [5]. This unsupervised, density-based outlier detection method detects anomalies in both sparse and dense areas of an input dataset, by examining a ”neighborhood” for each point de-

fined by its kNN. This data-driven outlier detection strategy, which can identify unusual occurrences solely by evaluating other data observations around a target point, is a powerful tool for detecting outliers in a rich variety of data distributions. LOF has been shown to better identify attacks in intrusion detection systems than other approaches [10], and achieve better accuracy in correctly identifying anomalous points for datasets with varying characteristics [11], outperforming other popular outlier detection methods.

1.2 Limitations of State-of-the-Art

Given the usefulness of kNN search and the LOF algorithm and the enormous amount of data available, it is clear that efficient distributed solutions can provide great value by leveraging modern distributed infrastructures.

Distributed kNN Search

However, in the literature most distributed kNN search approaches from Locality-Sensitive Hashing [12] to Z-Curves [9] only produce approximate results. Although a small amount of error may be acceptable in some applications, in many cases the exact results are required. In the case of outlier detection, outliers by definition make up an extremely small portion of the dataset which may fall within the margin of error of an approximate solution. It is apparent that an effective method for exact kNN distributed search is required to facilitate such methods on large data.

Although a distributed kNN join approach is proposed in [9] that can be adapted to support exact kNN search, it is not scalable to datasets containing above 10 million points, as confirmed in the authors' own experiments. In order to ensure kNN search can be conducted independently on each machine of a distributed system, the method duplicates a huge amount of data across the cluster. This leads to prohibitively high

communication and computation costs. Worst yet, for skewed data, more points may be assigned to a single machine than what it can physically accommodate, resulting in job failures. Overall, to date no distributed approach has been proposed that scales exact kNN search to large datasets.

Distributed Outlier Detection

The need for distributed solutions for outlier detection has been widely recognized and addressed by researchers. Recent efforts have begun to explore distributed solutions for outlier detection using global distance measures [4, 13, 14, 9]. However these techniques are not designed to handle datasets with varying density. As best can be determined, no distributed work has been proposed to date to perform local outlier detection on large datasets.

1.3 Challenges

Designing an efficient distributed solution to exact kNN search, and by extension LOF, is challenging. In a distributed system with a shared-nothing architecture, an input dataset has to be partitioned and sent to different machines. A distributed solution depends on two key elements: (1) the design of a data partitioning plan which groups nearby points together, and (2) a strategy to guarantee that the neighbors of each point can be found locally within a single partition. The latter challenge is typically addressed by augmenting each partition with *supporting points* [3, 9, 15] which allow for local autonomy of data mining tasks. This however requires that points which are processed in one partition and also fall within supporting areas of other partitions be duplicated and sent to possibly many machines. High data duplication results in excessive communication costs. Thus, an effective strategy to limit prohibitive data duplication rates is crucial.

Predicting the support points for each partition is challenging, since the distance from a point to its kth-nearest-neighbor may vary considerably based on the data distribution. The distance from a point to its neighbors will be small if it falls in a dense area, while it could be very large if it instead falls in a sparse area. In parameterized methods, such as distance-based outlier detection methods [16] and density-based clustering [3], the distance to search for neighbors is fixed. For exact kNN search however there is no a priori known parameter which determines a fixed bound on the size of supporting areas.

Although the method proposed in [9] could in principle be adapted to bound the support points of all points in a given data partition, this bound corresponds to a *safe* but *worst case* estimation. This very conservative bound could lead to a large number of replicas. Hence an effective strategy is required to bound the support points of each data partition with a minimized duplication rate while still ensuring the correctness of the kNN search.

In addition to high communication costs, distributed kNN search also carries a high computational complexity. An efficient centralized kNN search algorithm must be applied locally on each machine. End-to-end execution time is determined by the longest running individual process, and therefore load balancing is essential. This proves to be yet another challenging task since real-world data varies in density and distribution, which may impact the number of points assigned to each supporting area, causing load variation from machine to machine.

Finally, real-world data often includes many features for each data point. The utility of distances between neighboring points has been debated as it applies to high-dimensional data [17]. The inherent limitations imposed by the curse of dimensionality on the effectiveness of nearest neighbor search are beyond the scope of this work. It has been noted however that the performance of nearest neighbor algorithms typically grows exponentially in the number of dimensions [18]. For very large data, this renders a high dimensional solution intractable.

However, a robust implementation of kNN search that scales to handle a reasonable number of features, such as up to 10, is significantly more versatile that one that can handle only 2 or 3 dimensions. The added complexity of evaluating more features per data point impacts the amount of data that can be processed by each machine, and thus the running time of the computation. A suitable partitioning method should account for this increased complexity.

1.4 Methodology

The challenges above require the design and implementation of an algorithm for distributed kNN, optimized for efficiency and scalability, and feasible for use on real-world data. In this work we propose an efficient distributed approach called *PkNN* (Pivot-Based kNN Search). Using the popular MapReduce paradigm [19], PkNN successfully scales kNN search to large datasets.

Similar to [9] PkNN first splits the data points based on their distances to a set of selected pivot points. However, unlike [9] PkNN no longer utilizes theses distances to directly estimate the worst case distance to neighbors of the points in each data partition. Instead PkNN proposes a counter-intuitive approach. It overturns the common understanding that in shared-nothing distributed infrastructures an efficient approach should complete the analytics task in as few as possible rounds. This observation is shown not to hold in the scenario of kNN search.

Instead, PkNN decomposes the kNN search into multiple phases. The insights learned in each step are fully utilized to dynamically guide the search in a data-driven manner. PkNN uses a *multi-pass adaptive support strategy* (MPASS) to define supporting areas. First, a preliminary kNN search learns the characteristics of the data assigned to each partition. Based on these insights a two-tier *support determination* procedure is performed

at first the partition level, and then the individual point level by introducing the concept of a *boundary hyperplane*.

Additional optimizations further reduce the computation and communication costs of PkNN. In particular, an *early termination strategy* effectively identifies those data points for which the kNN have been found in the preliminary kNN search phase. These points can be immediately removed from the next phase, avoiding unnecessary communication costs and redundant computation. The pivot-based methodology is harnessed on individual machines as well, where a *pivot-based index* is used within individual partitions to speed computation.

These strategies effectively reduce the data duplication rate and scale kNN search. Experimental evaluation demonstrates that PkNN outperforms the existing state-of-the-art in runtime, data duplication, and scalability. The impact of parameter selection and data dimensionality is thoroughly investigated. Using real data, PkNN is confirmed to scale to handle upwards of 1 billion data points. PkNN is further shown to support distributed LOF computation by providing the key piece of a distributed LOF framework, called *PkNN-LOF*.

For the proposed distributed algorithms in this thesis, the Hadoop distributed computing framework [20] using MapReduce [19] has been chosen. This popular technology is desirable for its scalability in number of nodes, flexibility in the data model, fault tolerant execution, and cost effectiveness as an open-source technology which can run on commodity hardware. As a shared nothing-architecture, it allows for the design of a distributed algorithm general enough to be adapted to alternate platforms. Therefore, the algorithms proposed can also be utilized on other popular shared-nothing architectures.

Contributions:

1. A distributed approach in MapReduce called *PKNN* is proposed that for the first

time scales exact kNN search to billion point level datasets.

2. The MPASS performs computation over several rounds and fully explores the insights collected in each phase for multiple optimization objectives such as limiting unnecessary search, pruning completed points early, and speeding up the kNN search process.

3. PkNN is shown to facilitate distributed computation of a high-value unsupervised learning task, the Local Outlier Factor algorithm.

4. Evaluation on the large real world and synthetic datasets shows that *PKNN* significantly outperforms the state-of-the-art approaches in various scenarios.

Chapter 2

Preliminaries

Here we formalize the distributed kNN problem. Frequently used symbols are listed in Table 2.1. In typical applications, the data instances, which we refer to as points, consist of *feature vectors* which describe entities in the real world [8]. The task is to find for each point a set of k other points which are similar. We assume this similarity measure to be a well defined distance metric $|\cdot|$ such as any $L^p\ norm$.

2.1 Distributed kNN Search

Definition 1. ***kNN Search.*** *$\forall p \in D$, find the set of points $kNN(p)$ where $kNN(p) = \{q_1, q_2, ..., q_k \in D \mid \forall r \in D - kNN(p), |p, q_i| \leq |p, r|,\ r \neq p\}$.*

The naïve solution for centralized kNN search is to simply compare each point with every other point in the dataset and choose the k closest. This requires access to all points in the dataset, i.e. it is quadratic in time complexity. For today's huge datasets, the time to complete this exhaustive search is prohibitive. Furthermore, the data may be too large to fit into memory on a single machine. Therefore the kNN search task must be completed without local access to the entire dataset.

Symbol	Definition
p, q, r	data points
kNN(p)	the k nearest-neighbors of p
V_i	Voronoi Cell
v_i	pivot corresponding to V_i
$V_i.core$	$\{p \in D \mid \lvert p, v_i\rvert \leq \lvert p, v_j\rvert \; i \neq j\}$
$V_i.support$	all points in the *supporting area* of V_i
core-dist(V_i)	max $\lvert p, q\rvert \; \forall p, q \in V_i.core \; q \in kNN(p) \in V_i.core$
H_{ij}	hyperplane boundary between cells V_i and V_j
hp-dist(q, V_i)	distance from $q \in V_j$ to H_{ij}
support-dist(V_i)	max $\lvert p, v_i\rvert + \lvert p, q\rvert \; \forall p, q \in V_i.core \; q \in kNN(p) \in V_i.core$
ib(V_i)	$\lvert v_i, v_j\rvert/2$ where v_j is the closest pivot to $v_i \in V_i$

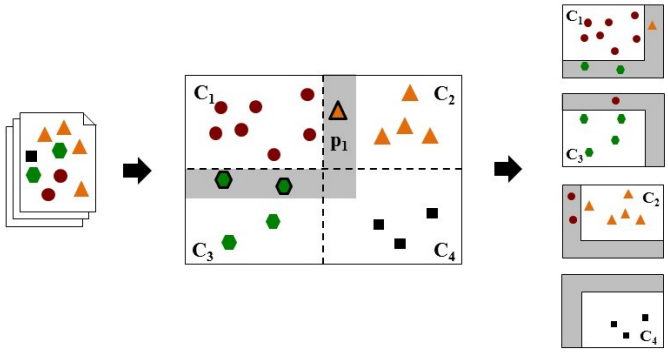
Table 2.1: Table of Symbols.

Definition 2. ***Distributed*** k***NN Search (Problem Statement).*** *Given dataset* D *distributed across* m *machines in a distributed compute infrastructure, perform* k*NN Search by processing* m *disjoint subsets (called cells)* $\{ C_i \mid C_1 \cup C_2 \cup ... \cup C_m = D, \; C_i \cap C_j = \emptyset \}$ *on separate machines independently and in parallel while minimizing end-to-end execution time.*

We target MapReduce [19] style distributed computing platforms where information about data partitions is not shared between machines. Our task is to ensure that kNN search can be completed autonomously for the portion of the dataset processed on each machine. If data is distributed among machines randomly, then it is likely that the neighbors of a given point will be sent to a different machine. Therefore a data partitioning strategy which preserves data proximity is desired. However, for points that lie along the boundaries of the cell, some neighbors may still be sent to a different machine.

A *supporting area partitioning strategy* ensures that the neighbors of all points can be found by augmenting each cell with extra points which may be neighbors of those points near the boundary [3, 9, 15]. First, *supporting areas* are determined for each cell. The points inside a cell C_i are denoted as $C_i.core = \{p \mid p \in C_i\}$. The data points within the

supporting area of the cell, denoted $C_i.support$, may affect the kNN decision of the core points of C_i.



Data stored on HDFS blocks is mapped to a *supporting area* partitioning plan. Point p_1 will be mapped to both cell C_1 as a supporting point, and cell C_2 as a core point.

Figure 2.1: Supporting area of partition C_1.

Definition 3. ***Supporting Area.*** *The* ***supporting area*** *of a cell C_i contains at least all data points that satisfy the following two conditions: (1) $\forall q \in C_i.support$, $q \notin C_i.core$, and (2) there exists at least one point $p \in C_i.core$ such that $q \in kNN(p)$.*

This strategy categorizes data points into two classes, namely *core points* and *support points*. Supporting areas must be sufficient to guarantee that the kNN of all core points in a cell C_i can be found among $C_i.core$ and $C_i.support$.

Figure 2.2 illustrates a simple *supporting area partitioning plan* being used in the execution of a MapReduce job. The dataset is divided into 4 cells, with the supporting areas of each cell highlighted in gray. First, points are mapped to the core and supporting areas of cells. Then, data assigned as core and support points of each cell C_i are mapped to the same data partition, processed by a single reduce task. In this way, some points are included in multiple partitions. For example, Reducer 1 receives a data partition containing the circle-shaped points in $C_1.core$, as well as support points in the gray area. Some of the points in $C_1.core$ are duplicated and sent to the supporting area of C_2 and processed independently by a different reducer.

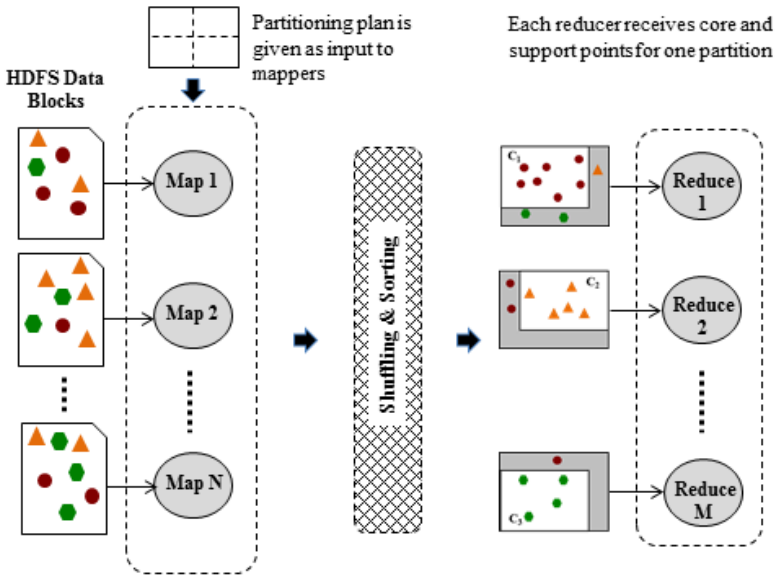


Figure 2.2: *Supporting area* partitioning plan used in the execution of a MapReduce job.

This strategy for distributed kNN search correctly discovers neighbors. However it hinges on the fact that supporting areas must meet the criteria of Definition 3. To accomplish this, PkNN takes advantage of *pivot-based* data partitioning to define data cells. This in turn facilitates a data-driven process to define effective supporting areas. It also provides a number of unique opportunities for optimization, as is described below.

2.2 Pivot-based Data Partitioning

The pivot-based data partitioning strategy is a general method to divide a dataset by first choosing a small set of n initial points, or $pivots$, in a pre-processing step. Grouping data points according to their closest pivot results in a partitioning known as a Voronoi Diagram. It determines a unique division of a metric space applicable to higher dimensions and various distance measures. Figure 2.3 shows two Voronoi diagrams for $n = 9$ pivots. A formal definition of a Voronoi cell is given in Definition 4.

Definition 4. ***Voronoi Cell.*** *Given a dataset D and a set of n pivots $P = \{v_1, v_2, \ldots, v_n\}$*

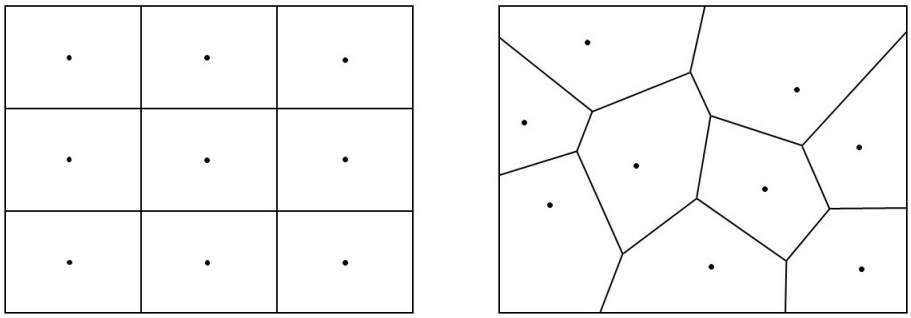


Figure 2.3: Voronoi cell partitioning using a lattice of points to form a grid (left) and irregularly shaped cells (right).

we have n *corresponding Voronoi cells* $V_1 \ldots V_n$ *where* $V_1 \cup V_2 \cup ... \cup V_n = D$, *and* $V_i \cap V_j = \emptyset$. *Each* v_i *serves as the pivot for cell* $V_i = \{p \mid |p, v_i| \leq |p, v_j|\}\, \forall v_j \in P, \forall p \in D, i \neq j$.

Indexing data by proximity to pivots preserves data location, and provides flexibility in that different pivot selection strategies result in cells of different shapes and cardinality. As Figure 2.3 illustrates, choosing pivots from the domain space in a regular lattice produces a uniform tessellation of cells - in this example a grid. Choosing pivots from the dataset itself, for instance using random sampling, produces cells which reflect the underlying geometry and distribution of the data.

As we will show, in the process of partitioning we learn valuable information, namely, the distance from each data point to the pivots. We can then exploit this key characteristic of *Voronoi cells* to bound their *supporting areas*.

2.3 MapReduce

The MapReduce computing model has emerged as a wide-spread solution for distributed data processing [19]. This work targets the popular Hadoop distributed computing platform [20], which implements MapReduce and offers an open-source framework for managing data on distributed computing clusters using commodity hardware.

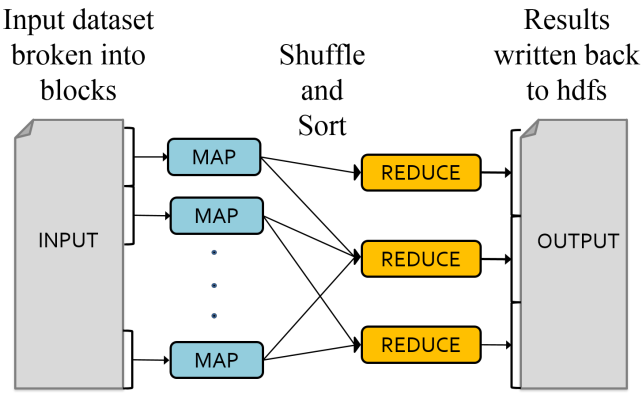


Figure 2.4: Hadoop MapReduce Framework.

A typical MapReduce job is depicted in Figure 2.4. The simple yet powerful programming model consists of 3 parts. First, the map phase: data is read in from the filesystem and a map function is applied to each data point. The input and output of the map function are key-value pairs. Results are stored temporarily, while mapping completes.

The second step is the shuffle and sort phase, where the output from the mappers is grouped according to some partition function on the key values. Each resulting data partition is sent to a node in the cluster for the reduce phase. A reducer function, which can only access one partition of data stored on its local machine, performs computation on the set of values associated with each key, and the results are written back to the filesystem.

Hadoop stores data in a distributed filesystem (HDFS), organized in small blocks which may be randomly distributed throughout the cluster. During the shuffle and sort phase, after each block has been processed by a mapper, large amounts of data are transferred between machines, sending data output from mappers to reducers. These communication costs greatly impact performance. The other main consideration when developing a MapReduce algorithm are the I/O costs of reading and writing to HDFS . These factors tend to dominate computational costs incurred while processing data in the map and reduce functions [21].

Chapter 3

Pivot-based k-Nearest-Neighbor Search

The key to fully exploit pivot-based data partitioning and effectively scale kNN search in a distributed algorithm is to break the process into several steps. PkNN uses a multi-pass adaptive support strategy (MPASS) to distribute data among partitions and determine sufficient supporting points for each based on the characteristics of the underlying data. PkNN utilizes insights gained at each phase to minimize data duplication, all the while avoiding redundant computation.

3.1 Multi-pass Adaptive Support Strategy

An initial job selects pivots which evenly partition the data by ensuring that no single partition is so large as to overwhelm the resources of a single reducer. Then, an initial kNN search is conducted *only over the core points in each cell* in another job. Information collected during this first pass kNN search is used to determine a tight upper bound for each cell on the distance from any core point to its kth-nearest-neighbor outside of the cell. A final job performs support determination at both the cell level and point level, followed by a supplemental kNN search *over support points* that identifies neighbors which were

not available to local partitions initially.

Surprisingly, we find that the costs incurred from executing multiple jobs are heavily outweighed by achieving a much tighter estimation of the size of supporting areas. In this way the number of actual points that are replicated due to the supporting area requirement are reduced. It is important to observe that no repeated computation ensues, as the sets of points considered in the kNN search phases are naturally disjoint - core points in the first step and support points in the second. Next each phase is described in detail.

3.1.1 Pivot Selection

A first pass over the data performs uniform random sampling via a distributed reservoir sampling technique [22]. This ensures that each point has an equal likelihood of being chosen, and the resulting subsample is a good estimation of the data. That is, points in very dense areas of the dataset are more likely to be chosen than points in sparse areas. The subsample will reflect the distribution of the data for this reason. For an input dataset D with size d, and a limit on the number of points m which can fit into memory on a single machine, a percentage $\alpha \leq m/d$ points are selected from those processed by each mapper. Then this representative subsample is s-ent to a single reducer.

At this stage, to avoid choosing too few pivots, the number of core points which will be assigned to each pivot can be estimated. Reservoir sampling is applied a second time to choose the desired number of pivots from the subsample. For each cell V_i, we count the number of points c_i from the subsample which are closer to it than any other pivot. An estimate of the number of core points from the entire data set which will be assigned to V_i will be $\gamma_i = c_i/\alpha$.

If γ_i for any partition exceeds m, then we anticipate that a single reducer will not have enough resources to process the data assigned to that partition. In this case, the sampling method can be repeated iteratively choosing a greater number of pivots each time until

enough pivots are chosen, this strategy ensures reasonably sized partitions.

Given a satisfactory set of pivots, an initial kNN search is next conducted over the core points in each partition. Information collected during this step is used to determine data-driven bounds on the size of the support area for each partition. The bounds given below are then applied in a subsequent job to complete the kNN search.

3.1.2 Bounding Support Areas

For a point $q \in V_j$ to be a nearest-neighbor of $p \in V_i.core$, the distance from p to q must be less than the distance from p to its kth nearest neighbor in $V_i.core$. By first evaluating the core points of each cell, the maximum distance of any core point p to its kth nearest core neighbor r can be determined. This value gives a tight upper bound on the distance from core points to possible neighbors outside the cell. We define this as the *core-distance* of the cell.

Definition 5. The core-distance of a Voronoi cell V_i.
core-distance$(V_i) = max(|p, q|) \; \forall \; p, q \in V_i.core$ *where* $q \in kNN(p) \in V_i.core$.

The *core-distance* of each Voronoi cell gives a natural bound on the size of its supporting area. Leveraging this notion, we are now ready to design a customized evaluation rule to determine whether a point $q \in V_j$ belongs to $V_i.support$. The rule depends on the ability to determine the distance from a point q to a partition V_i. When the commonly employed Euclidean or Mahalanobis distances are used, the boundaries of Voronoi cells are comprised of piecewise linear hyperplanes described by Theorem 3.1.1. In this case, the exact distance from any point to the hyperplane boundary of two cells can be computed [23].

Theorem 3.1.1. *The boundary between two adjacent Voronoi cells V_i and V_j is a hyperplane H_{ij} containing their midpoint. In $\mathbf{R}^d$ the hyperplane is given as $H_{ij} = y : y^T n + p = 0$*

where n is the normal vector orthogonal to the plane. $\forall y \in H_{ij}, |y, v_i| = |y, v_j|$. *The distance of any point* $p \in \mathbf{R}^d$ *to the plane is* $|p, H_{ij}| = \frac{|s^T n+p|}{\|n\|_2}$ *[23]*

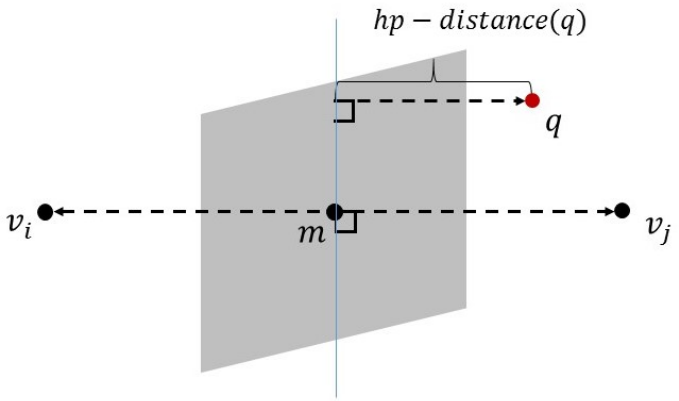


Figure 3.1: The *hp-distance*(q) to the hyperplane boundary between two cells V_i, V_j.

However, in arbitrary metric spaces, computing the distance from a point to the exact boundary between cells may not be possible. Fortunately, a lower bound on this distance given by Theorem 3.1.1 is shown via triangle inequality in [8] to hold, and can be used in place of an exact calculation.

Theorem 3.1.2. *Given* $q \in V_j, p \in V_i$ $i \neq j$, $|q, p| \geq \frac{|q,v_i|-|q,v_j|}{2}$.

Using either the exact calculated distance in Theorem 3.1.1 or the lower bound distance in Theorem 3.1.2, we can compute a distance measure from a point to a cell, hereafter referred to as the *hp-distance*. With this, Lemma 3.1.3 provides the means to evaluate whether a point should be mapped to the supporting area of a cell.

Definition 6. *: The hp-distance from a point to a Voronoi cell* V_i**.**

hp-distance $(q, V_i) = |q, H_{ij}|$ $\forall q \in V_j$ *where* H_{ij} *is the boundary of Voronoi cells* V_i, V_j *and* $i \neq j$.

Lemma 3.1.3. *If* $q \in kNN(p)$ *for some* $p \in V_i,\ q \in V_j,\ i \neq j$, *then hp-distance*$(q, V_i) <$ *core-distance*(V_i).

Proof. Let $r \in V_i$ be the kth-nearest-neighbor of $p \in V_i$ among $V_i.core$. By Definition 5, $|p, r| \leq$ *core-distance*(V_i). If $q \in V_j$ is in the true kNN(p), then $|q, p| \leq |r, p|$. By Theorem 3.1.2 *hp-distance*$(q, V_i) \leq |q, p|$. □

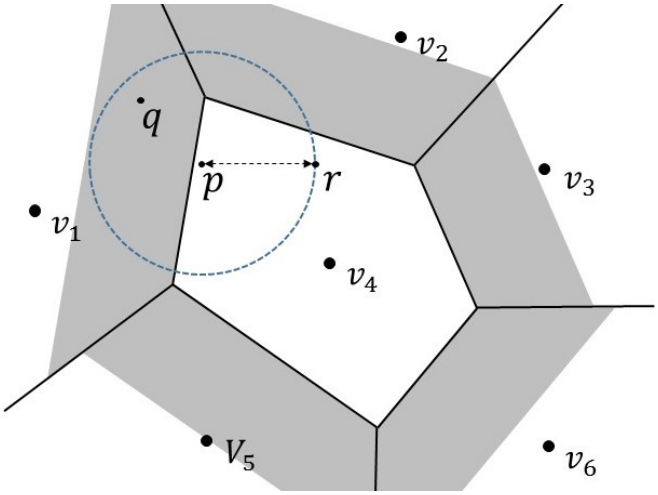


Figure 3.2: *Core-distance*(V_4) used to define *supporting areas*.

Figure 3.2 illustrates the intuition behind the *core-distance* bound in a 2D space. The shaded areas represent an extension of the linear boundaries of cell V_4 by its *core-distance*, which in this case has been determined by the distance from p to its kth-nearest-neighbor r. In order for a point q to be a closer neighbor to p than r, it must fall within this *supporting area*.

3.1.3 Support Cell Determination

While Lemma 3.1.3 is sufficient to determine all supporting points for each partition and facilitate distributed kNN search, it may still lead to unnecessary data duplication. When pivots v_i and v_j are far from each other, even if points in $V_i.core$ lie close to the hyperplane boundary with V_j, they may not necessarily be support points of V_j, since V_j may actually be very far V_i, in particular if the Voronoi cells are not adjacent.

Therefore, we now design a lightweight optimization to further reduce the number of points assigned to support areas. This two-tier strategy leverages pivot-based partitioning to perform support determination at the both the cell and individual point level granularity. We provide a bound on the distance from each cell to *candidate support cells*. That is, for each cell V_i we determine which other cells V_j the points in $v_i.core$ may support.

Definition 7. ***Support-distance of a Voronoi cell.*** *support-distance*$(V_i) = max(|v_i, p| + |p, q|)$ *where* q *is the* k*th nearest-neighbor of* p $\forall p, q \in V_i.core$.

The *support-distance* captures the maximum distance from a pivot to a possible support point of its cell. This distance can be used to prune other cells which could not possibly contain support points, as stated in Lemma 3.1.4. Performing this pruning at the cell level, we not only reduce unnecessary data duplication, but also reduce the number of cells a point must be checked against when mapping points to supporting areas. This in turn reduces the amount of computation necessary to evaluate Lemma 3.1.3.

Lemma 3.1.4. *Given Voronoi cells* V_i, V_j *and their corresponding pivots* v_i, v_j $i \neq j$*, if the support-distance*$(V_i) \leq |v_i, v_j|/2$*, then* V_j *does not contain any support points of* V_i.

Proof. Recall that by Definition 4, since q is in V_j, it is closer to the pivot v_j than to any other pivot. Therefore, $|q, v_i| \geq |v_i, v_j|/2$. We give a proof by contradiction: Let some point $q \in V_j$ be a supporting point of cell V_i. Then by Definition 3, $\exists p \in V_i.core$ such that $|p, q| < |p, r|$ where $r \in V_i.core$ is the kth nearest neighbor of p out of all the core points in V_i. Assume that Theorem 3.1.4 is not true. Then we have:

$$
\begin{aligned}
|r, p| + |p, v_i| &< |v_i, v_j|/2 && \text{by assumption} \\
|q, p| + |p, v_i| &< |v_i, v_j|/2 && \text{def 3} \\
|q, v_i| \leq &|q, p| + |p, v_i| && \text{triangle inequality} \\
|q, v_i| &< |v_i, v_j|/2 && \text{transitivity}
\end{aligned}
$$

This results in a contradiction. If $|q, v_i| < |v_i, v_j|/2$, then $q \in V_i$, which violates the original assumption. □

3.1.4 Comparison with State-of-the-Art Support Area Bound

In [9], pivot-based partitioning is used by Lu et al. in the context of distributed *k*NN join processing. This join task is to find the kNN of all points in one set R among points in a different set S. The kNN search problem addressed in this thesis is an example of a self-join in this case where $S = R$. The authors derive a worst-case bound θ used to determine the size of supporting areas containing points in S for each cell $V_i \in R$. The kNN search for the join is conducted in a single MapReduce job using this worst-case estimate to map points to supporting areas.

This bound θ in [9] is based on the relationship of pivots to a small number of points in each cell collected during data partitioning. The distance from any point $p \in V_i$ to its kth-nearest-neighbor q is bounded by $\theta_i = |v_i, v_j| + max(|p, v_i|) + |v_j, r| \; \forall p \in V_i.core, \; i \neq j$. This is determined by first finding for each Voronoi cell V_j the $kNN(v_j)$ from $V_j.core$. Let U be this set of the kNN of all pivots. Then θ_i for each V_i is computed using the distance from v_i to the kth closest point $r \in U$ and corresponding pivot v_j such that $r \in V_j.core, \; i \neq j$. The intuition for this bound is that we can find at least k points closer to any point $p \in V_i$ than θ_i. A formal proof can be found in [9].

While an elegant solution, this approach leads to very high data duplication rates, with the entire dataset being replicated a large number of times over in practice. By comparison, in the PkNN method the data driven *core-distance* bound is the actual maximum distance of any point $p \in V_i$ to its kth-nearest-neighbor in $V_i.core$. Clearly this is a much better gauge than θ_i.

Furthermore, to check if a point $q \in V_l$ is a supporting point of a cell V_i, Lu et al. [9] use a rule comparing $|q, v_l|$ against a bound $LB(q, V_i) = max(0, |v_i, v_l| - U(V_i) - \theta_i)$

where $U(V_i)$ is the $max(|p, v_i|)$ $\forall p \in V_i$. We can see that any time v_l is closer to v_i than θ_i, $LB(q, V_i)$ will be exactly zero, and all points $q \in V_j$ will be mapped to the supporting area of V_i. This clearly would cause unnecessary data duplication. In addition, any point q that is far from its pivot v_l relative to $|v_i, v_l|$ will also be unnecessarily duplicated.

In contrast, PkNN uses the two-tier assignment strategy described in Sections 3.1.2 and 3.1.3, avoiding excess data duplication. The core points of each Voronoi cell V_i are evaluated in an initial pass, allowing the use of a natural bounds on the distance to both candidate support cells and points in $V_i.support$. Cells that are far from V_i are safely ignored using the pruning strategy of Lemma 3.1.4. No entire cell is mapped to $V_i.support$, instead support determination is performed at the level of individual points using Lemma 3.1.3.

Experimental evaluation in Section 5.2 compares PkNN to the PBJ method in [9]. A 20 fold improvement in data duplication for PkNN over the state-of-the-art is observed.

3.2 Early Output Optimization

In addition to limiting the size of supporting areas, and thereby the duplication rate, our proposed two-pass PkNN approach provides another important opportunity for optimizing kNN search. We observe that while a distributed kNN search requires a supporting area partitioning strategy to ensure that the kNN of all points can be found locally, in practice only those core points that lie close to the boundaries of each cell will find neighbors in the supporting areas. If the kNN of a point can be guaranteed to be found among the core points of its cell, then there is no need to search in a second pass among the support points. These points can be written to disk early, saving on further I/O as well as search time within partitions, instead of being evaluated again in a second round. To determine which core points can be pruned early, we introduce the notion of an *interior bound*.

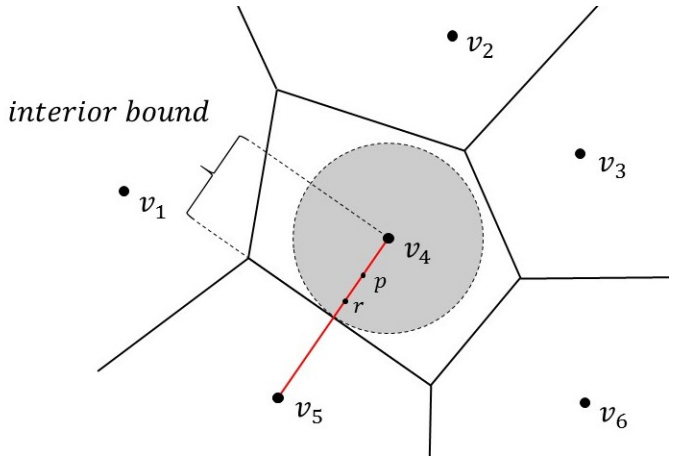


Figure 3.3: *Interior bound*(V_4) used to determine early output.

Definition 8. ***Interior bound of a Voronoi cell.*** $ib(V_i) = |v_i, v_j|/2$, *where* v_j *is the closest pivot to* v_i.

During pre-processing, with one pairwise comparison of the pivots, noting that the number of pivots is very small compared to the dataset size, the interior bound for each cell can easily be determined. Lemma 3.2.1 provides a means to determine whether or not the true kNN are guaranteed to have been found in the first kNN search phase for core points that lie within the area defined by the interior bound. This evaluation assures that points which meet this condition do not require kNN search over support points. Since the search for these points is complete, their results can be written to disk early, and they do not need to be evaluated as core points in the second kNN search phase, avoiding unnecessary communication costs and redundant computation.

Lemma 3.2.1. *For* $p, q \in V_i.core$ *where* q *is the* k*th nearest neighbor of* p *in* $V_i.core$*, if* $|p, v_i| + |p, q| < ib(V_i)$*, then* $kNN(p)$ *in* $V_i.core$ *are guaranteed to be the true* $kNN(p)$.

Proof. If q is not the true kth nearest neighbor of p, then $\exists\ r \in V_j, i \neq j$ s.t $|r, p| < |q, p|$. We know that since $r \notin V_i, |r, v_i| > |r, v_j|$, and that $|v_i, v_j|/2 \geq ib(V_i)$. If we assume

Lemma 3.2.1 is not true, then we have:

$$
\begin{aligned}
|p,q| + |p,v_i| &< ib(V_i) && \text{by assumption} \\
|p,r| + |p,v_i| &< ib(V_i) && \text{def 1} \\
|r,v_i| \leq &|p,r| + |p,v_i| && \text{triangle inequality} \\
|q,v_i| &< ib(V_i) && \text{transitivity}
\end{aligned}
$$

If $|r, v_i| < ib(V_i)$, then $r \in V_i$, which violates our original assumption. □

Figure 3.3 depicts the geometric intuition behind the internal bound. For the point p, its kth-nearest-neighbor q lies exactly in the direction of the hyperplane bound between V_4 and the closest neighboring cell V_5. We can see that it is not possible for p to have a closer neighbor in V_5, nor in any other adjacent cell. Therefore, p has found its true kNN in the first pass, and does not require further processing.

3.3 Pivot-based Indexing for kNN Search

To mitigate the high computational cost of kNN search, PkNN requires an efficient local kNN search technique be used within data partitions. Given the quadratic complexity of an exhaustive kNN search, indexing schemes such as variations of the R-tree [24] are often used to speed up computation. However, these techniques carry overhead due to the construction of necessary data structures. Furthermore, they do not scale well in the dimensionality of the data [6]. Fortunately, PkNN obtains information during data partitioning that can facilitate an efficient local kNN search through the use of a simple indexing technique. In [13] indexing by distance to some reference point is shown to speed up a search for neighboring points within a certain threshold for the purpose of outlier detection. In our context, we have the ideal reference point for such an index in

the pivot of each cell, and can adapt this approach to find the kNN for all core points.

The key insight to motivate such an index is that if a point under consideration is very close to the pivot of its own Voronoi cell, then it is likely that its neighbors are close to the pivot as well. This implies that points far from the pivot may not need to be considered as potential neighbors. Similarly, for points far from the pivot, their neighbors are likely to be far as well. Therefore, by checking points with a similar distance to the pivot first, the Voronoi cell structure can be exploited to find neighbors quickly.

To construct this *pivot index* for a cell V_i all points in $V_i.core$ and any $V_i.support$ are sorted by their distance to the pivot. We note here that since this distance is computed during data partitioning, the distance value can be embedded with each point for all further stages of computation. Using this pre-computed distance for local kNN search, sorting requires only relatively cheap comparisons of floating point values as opposed to distance computations. Figure 3.4 depicts the sorted list of points used as an index for a Voronoi cell V_i.

For each point $p \in V_i.core$, a search for $k\text{NN}(p)$ is conducted along this index order, alternating traversal of the list in ascending and descending order. Intuitively, by searching along the index order in this way, points with a similar distance to the pivot as p will be checked first. The $k\text{NN}(p)$ can be found without traversing through the whole data set by utilizing the following termination criterion.

Lemma 3.3.1. *Let M be a data partition consisting of the core points and any support points for Voronoi cell V_i sorted by their distance to the pivot v_i. Let $p \in V_i.core$ be the test point under consideration. The distance from p to each point in M is evaluated alternating traversal of the list in ascending and descending order from p. Let $kdist$ be the distance to the kth closest point in M found so far. When considering the distance from p to a point $q \in M$, the search can terminate immediately if: $\Big||p, v_i| - |q, v_i|\Big| > kdist$.*

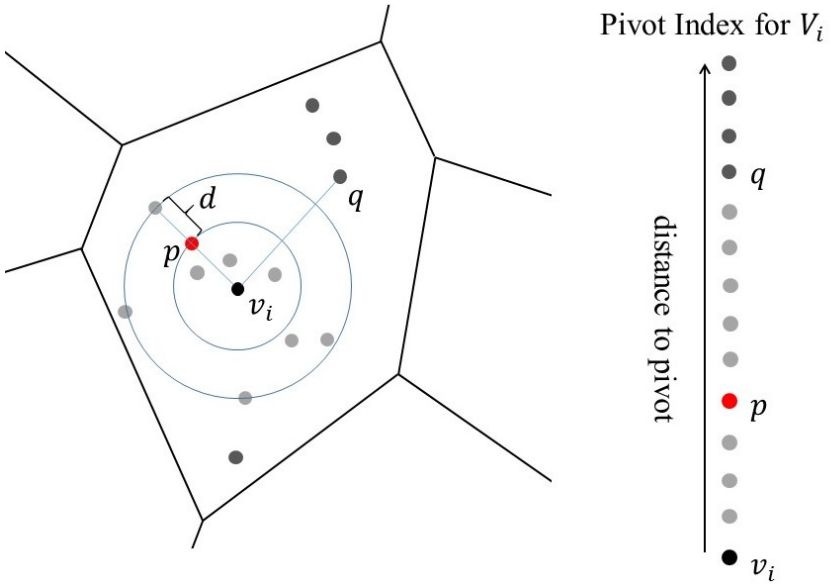


Figure 3.4: On the left is a Voronoi cell V_i. The pivot index is shown on the right, with points sorted by their distance to the pivot v_i. The search for the kNN(p) can stop at point q, since $|q, v_i| - |p, v_i| > d$, saving checking 4 additional points far from v_i.

Proof. First we show that if the above condition is true, then q cannot be closer to p than distance $kdist$.

$$
\begin{aligned}
|p,q| &> \Big||p,v_i| - |q,v_i|\Big| && \text{triangle inequality} \\
\Big||p,v_i| - |q,v_i|\Big| &> kdist && \text{by assumption} \\
|p,q| &> kdist && \text{transitivity}
\end{aligned}
$$

Next we show that for any point r evaluated after q, the above condition holds true as well. Since the points are evaluated in alternating order along the pivot index, each point evaluated will either be (1) closer to the pivot than p and q or (2) farther from the pivot

than p and q.

$$|r, v_i| > |p, v_i|, |r, v_i| > |q, v_i| \implies |p, r| > \Big||p, v_i| - |q, v_i|\Big| \quad (1)$$

$$|r, v_i| < |p, v_i|, |r, v_i| < |q, v_i| \implies |p, r| > \Big||p, v_i| - |q, v_i|\Big| \quad (2)$$

□

Lemma 3.3.1 can provide significant savings on the number of comparisons needed for kNN search and speed execution time. In the worst case, a neighbor will have to check all others in order to satisfy the stopping criteria. Therefore complexity in the worst case is $O(n^2)$. However, in the best case, the k-nearest-neighbors are sequential along the index order and the minimal number of comparisons are performed. Figure 3.4 illustrates a typical case.

Furthermore, this stopping criteria allows us to re-use previous work. Simple floating point comparisons of pre-computed distances from each point to the pivot are a cheaper way to evaluate neighbors than computing the actual distances between all points, especially as the dimensionality of the data increases. In the PkNN method, this *pivot index* provides an even more powerful speedup in the second round of kNN search. At this stage of computation, each point already has the distance to its kth-nearest-neighbor among core point associated with it. This means the stopping criteria avoids comparison with unnecessary support points.

Experimental evaluation in Section 5.1.1 confirms these observations. A speed up in the overall kNN search time for PkNN using the pivot index improves over an implementation of PkNN using the naive nested-loop algorithm by a factor of 100.

3.4 Overall PkNN Framework

Here we now show how the MPASS strategy is realized in one integrated optimized PkNN framework using the MapReduce paradigm. Figure 4.3 illustrates the overall framework.

Pivot Selection: (MapReduce job 1). Mappers sample the dataset, and a set of pivots is chosen by a single reducer. The pivot list is sent to all machines via the distributed cache for subsequent jobs.

Data Partitioning: (MapReduce job 2). A single job consisting of only a map phase assigns each point to a Voronoi cell based on its closest pivot.

After this job the pivot list is evaluated locally on the NameNode and refined. A constraint that cells contain at least $k+1$ core points is enforced by simply dropping pivots for cells with less points. This ensures that every core point can find k core neighbors in its cell. A pairwise comparison of the remaining pivots determines the *interior bound* for each cell. The pivot list is updated with this information, and put back into the cache.

Initial kNN Search: (MapReduce job 3). Mappers use the cell assignments from the previous job. Any point which was mapped to a cell with less than $k + 1$ points in data partitioning is reassigned to the next closest pivot. Then, each reducer receives all the core points of one Voronoi cell, along with each point's distance to the pivot. These distances are re-used for pivot-based indexing. The output of the initial kNN search is a key-value pair. The pivot id is the key, and the value is a tuple containing all information needed in subsequent steps.

As the initial kNN search is conducted, the *core-distance* and *support-distance* for each cell are computed and written out to a separate file on HDFS. This information is sent along with the pivot list to all machines for the next MapReduce job.

Support Determination: (MapReduce job 4 - setup and map). In a one-time setup phase on each machine, a hash table is constructed using the *support-distance* to map each cell to a list of cells it may support. Mappers then read in the key-value pairs output from the previous job. To map points to supporting areas, Lemma 3.1.3 is applied only for the cells in the support list of each point's core cell. Each point is also checked against the *interior bound* of the cell. If the kNN search for a point is given by Lemma 3.2.1 to be complete, it is written out to HDFS early. The rest of the data are again mapped to their Voronoi cells as core points.

Supplemental kNN Search: (MapReduce job 4 - reduce). Finally, reducers complete the kNN search for the remaining core points in each cell. Pivot-based indexing is again used, however in this phase only comparisons to supporting points are necessary, as all neighbors among core points have already been found in the first step.

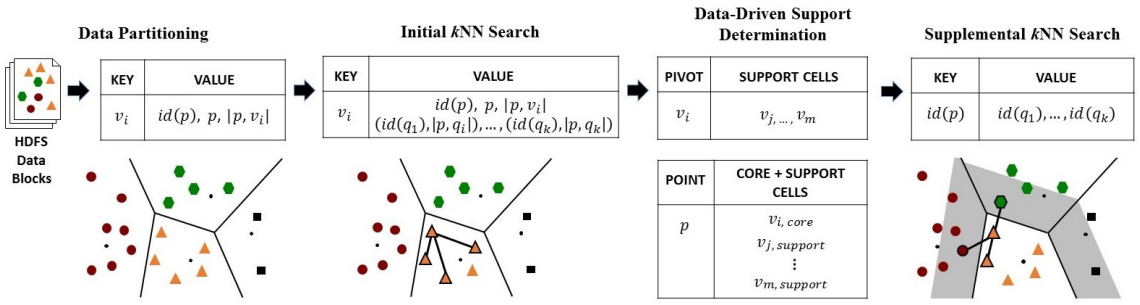


Figure 3.5: PkNN Framework.

3.5 Adaptive Support Determination

PkNN provides effective bounds on the size of supporting areas derived from the data within each individual partition. This greatly reduces the data dupication rate of previous

worst-case bounds given in [9]. However, the strategies described in the sections above fundamentally depend on some distance between points in the dataset. Both the theoretical worst-case estimation given in [9], and the data-driven MPASS approach are sensitive to the distribution of points in the dataset. When applied to real data with various characteristics, a robust distributed kNN search method must account for the possibility of extreme data skew, and the presence of outliers in the data. To truly scale kNN search to modern data quantities, PkNN requires an *adaptive* partitioning solution.

The specification of a given compute cluster determines a limit m on the number of points which can be processed on a single machine. The data assigned to a single partition must not overwhelm the resources of any reducer. Therefore our partitioning problem is subject to the constraint $V_i.core + V_i.support <= m, \forall\, V_i \in D$. As shown in Figure 2.3, the cardinality of Voronoi cells is determined directly by the set of pivots used, and the distribution of the underlying data. Since the actual data space is unknown, the pivot selection step only approximates the number of core points assigned to partitions. The number of pivots chosen can be increased to avoid partitions which are assigned too many core points. However, this will not address the number of support points.

Consider the case of an outlier in the dataset far from all other points. No matter the pivot set chosen, this point must be mapped to some cell. Due to the extreme distance of this point to its kth-nearest-neighbor, a skewed support area will result. To address this problem directly, a fixed constraint must be set on the maximum size of the core-distance of partitions. Luckily, the MPASS approach provides an opportunity to adapt data partitions to accommodate both the resources of the compute cluster as well as the distribution of the data.

To mitigate the impact of outliers in a dataset, a new threshold parameter $maxkdist$ is introduced which imposes a constraint on the possible size of individual supporting areas. Without this treatment, using pivot-based partitioning and derived bounds, the

PkNN method is still at the mercy of the distribution of points within each partition. This threshold enforces that $\forall\ V_i \in D$, *core-distance*$(V_i) \leq maxkdist$. If a partition contains core points too far from their kth-nearest-neighbor they are removed from the cell. Special handling for these points is now introduced.

Global Outlier Handling

As described in Section 3.1, the core-distance of each cell is determined during the initial kNN search phase of PkNN. To handle outlying points, the threshold parameter $maxkdist$ is specified in the configuration file for the initial kNN search job. As the distance from each point p to its kth-nearest-neighbor is found, points for which this value exceeds $maxkdist$ are flagged as *global outliers*.

Then, in the support determination phase these points are treated as additional "special" pivots. They differ from regular pivots in that each global outlier point is the only core point mapped to the partition defined. In other words, no additional points are assigned to this partition as the data partitioning stage has already completed. Since the global outlier points would have already found their kth-nearest-neighbor from among the core points in their original partition, this distance determines the core-distance of their new special partition. A list of the special pivots and corresponding core-distances are supplied to the support determination map phase via the distributed cache in addition to the regular pivot list. Support points are mapped accordingly.

kNN search is conducted at the reducer phase only for the single core global outlier point of each of these additional partitions. In this way, global outliers do not impact the support determination of their original partitions, nor do they skew the data duplication rate in the supplemental kNN search phase.

Chapter 4

Case Study on kNN Search for Distributed Outlier Detection

In this chapter we demonstrate one example of a case where PkNN can facilitate a distributed solution for an important data mining task. The Local Outlier Factor algorithm (LOF) [5] is a high value detection method based on the relationship between each point in a dataset and its kNN. Lacking an exact distributed solution for kNN search to handle modern data quantities, this algorithm is not feasible at scale.

4.1 Local Outlier Factor Algorithm

The LOF algorithm compares the relative density of each data point to the average density of nearby points. This technique succeeds in identifying outliers in cases where pure distance [16] and neighbor-based [4, 23] outlier detection methods fail. To illustrate this, Figure 4.1 illustrates a case where LOF is preferred over other methods. In the dataset depicted, points are densely clustered in one area and more sparsely distributed in another. Intuitively, points P and Q appear to be outliers which do not to fit the patterns of the data

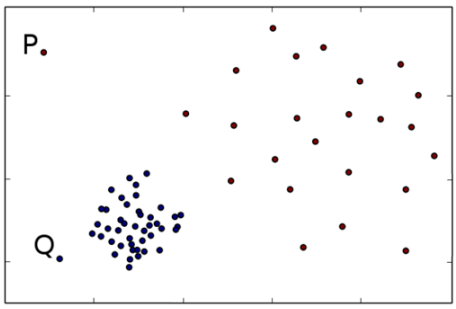


Figure 4.1: Outliers in dataset with varying densities.

in their respective vicinities. Unfortunately, simple outlier detection strategies categorize points as outliers based on some global distance measure. With this approach, any distance threshold which would identify Q as an outlier would also categorize many points in the upper right sparse area as outliers as well. By the same token, the use of a large enough threshold to categorize point P as an outlier would not identify Q as an outlier.

In comparison, the LOF algorithm [5] is robust to varying densities in a dataset. Instead of a fixed, pre-determined distance measure, the distance from each point to its kth-nearest-neighbor defines a neighborhood. Then, the density of each point is compared to the average density of its neighbors. In this way, LOF introduces a concept of "outlierness", where each point in the dataset is assigned an outlier score. The set of points with the highest scores are considered to be outliers. In the example above, the LOF algorithm will correctly identify both P and Q as more anomalous than the rest of the points in the dataset.

The computation of LOF requires a kNN search be performed for every point in the dataset. Then intermediate values must be computed for p and $k\text{NN}(p)$. The definitions below follow from the original paper.

4.1.1 LOF Concepts

For each point p in a dataset D, a single user-specified parameter k is used to determine a *k-neighborhood* made up of the kNN(p). The *k-distance(p)* is the distance from p to its kth-nearest-neighbor, and is used in lieu of a fixed distance parameter to determine the outlier status of each point.

Definition 9. ***k-distance of a point*** p**.**

The distance $|p, q|$ *between points* $p, q \in D$ *such that for at least* k *points* $q' \in D - p,\ |p, q'| \leq |p, q|$ *and for at most* $k - 1$ *points* $q' \in D,\ |p, q'| < |p, q|$.

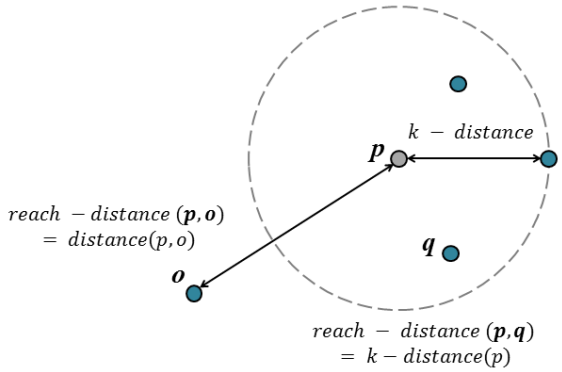


Figure 4.2: Neighborhood of p, $k = 3$.

Figure 4.2 shows the k-neighborhood of a point p. It is important to note that the neighbor relationship is not symmetric. If q is one of the kNN(p) it does not necessarily imply that p is one of the kNN of q. The k-distance value for each of the neighbors of p is used to determine another intermediate value for p, the *reachability-distance*.

Definition 10. ***Reachability Distance of*** p ***w.r.t.*** n**.**

reachability-distance$(p, n) = max($*k-distance*$(n),\ |p, n|)$.

For a point q far from p, the *reachability-distance* is simply the distance $|p, q|$, while for points close to p, such that the distance from p lies within the k-neighborhood of q, the

reachability-distance is the k-distance of q. This definition was introduced by Breuning et al. [5] as a smoothing factor for LOF. The *reachability-distance* is then used to compute the *local reachability distance* of p.

Definition 11. ***Local Reachability Density of a point*** p**.**

$$LRD(p) = 1/\left(\frac{\sum_{n \in kNN(p)} reach - dist(p,n)}{\|k - neighborhood\|}\right)$$

$LRD(p)$ is the inverse average reachability distance of p to kNN(p). The LRD value of every point in the dataset must be computed in order to compute the final LOF scores.

Definition 12. ***Local Outlier Factor of a point*** p**.**

$$LOF(p) = \left(\frac{\sum_{n \in kNN(p)} \frac{LRD(n)}{LRD(p)}}{\|k - neighborhood\|}\right)$$

Informally, $LOF(p)$ corresponds to the ratio of the average density of each neighbor of a point p, to the average density of p. LOF scores close to 1 indicate in-lying points, and the higher the score the more out-lying the point. The points with the highest LOF values in the dataset are considered to be outliers.

4.1.2 Centralized LOF Algorithm

The centralized procedure for computing LOF is a two-step algorithm. In the first step, the kNN of each point p are found and materialized in a database along with the distances from each neighbor to p. For dataset D with cardinality n the database table has size on order $n * k$. In the second step, the LOF scores are computed. Two passes are made over the database to compute the LRD and LOF values. In each of these passes the intermediate

values for the neighbors of each point are updated, maintained in the global data table, and then utilized in the next step of the computation.

A number of centralized algorithms are available for the kNN search in the first step. In the original LOF paper, Breuning et al. propose to use an $O(n)$ grid-based search for low-dimensional data, an index-based $O(\log n)$ method for medium dimension data, or an $O(n^2)$ method for high dimensional data. For the sake of simplicity here, let us assume an index-based method with $O(\log n)$ complexity (perhaps similar to our own pivot-based index described in Section 3.3). The total time complexity of the centralized algorithm is then $O(n \log n)$ for the first step and $O(n)$ for the second. We note here that the bottleneck for computation is the kNN search.

Applying the centralized approach on a shared nothing distributed architecture is not practical. First, when computing the kNN of each point the centralized approach assumes access to all other data points. If the data size exceeds the resources of a single node in the compute cluster, than a distributed kNN search is required.

Second, even if the kNN of each point can be computed, it is not feasible to store the resulting kNN of all points along with their distances on one single compute node as a database table, considering the size of the large dataset. Therefore such intermediate values have to be stored across different compute nodes. This inevitably complicates the next two steps of the LOF algorithm, namely the computation of LRD and LOF values. For example, to compute the LRD value for a point p we have to locate the compute nodes that store kNN(p), and then retrieve this information from corresponding remote tables. However, since the shared nothing architecture does not allow data exchange at will, a distributed mechanism has to be designed for the efficient maintenance and retrieval of intermediate results.

PkNN can address the first challenge above by facilitating kNN search in parallel across compute nodes. Once the kNN of each point p are found in the last stage of

the PkNN algorithm, the k-*distance*(p) is known. The second challenge is more difficult. Although the supporting area strategy of PkNN allows us to identify the neighbors of each core point within a given partition, some of these neighbors may be support points. In this case, the k-distance values of these support point neighbors are computed in different partitions. Therefore, the information necessary to compute the LRD of the core points within a single partition is not available on the local machine. To solve this problem we introduce an extension to the MPASS approach using two additional MapReduce jobs.

4.2 Distributed PkNN-LOF Framework

To facilitate LOF computation, the PKNN framework described in Section 3.4 is augmented with 2 additional passes over the data. As the output of the final kNN search phase, each reducer writes core points along with the IDs of their kNN and their k-distance values back to HDFS. At the same time, partitioning related information is also maintained and re-used as in all previous stages of PkNN.

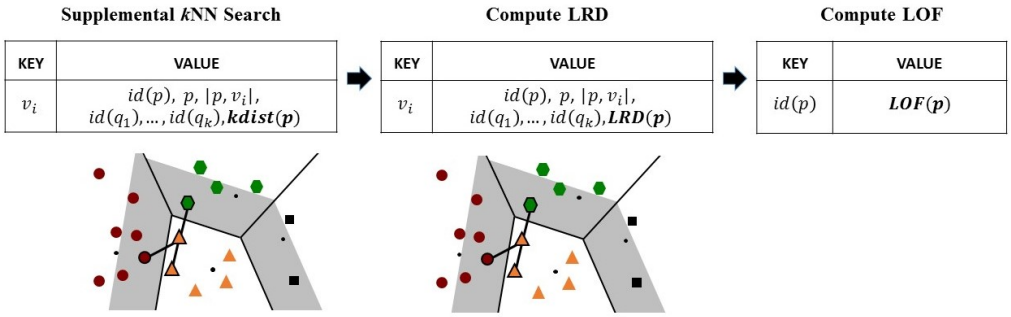


Figure 4.3: PkNN framework extended to include two additional rounds of computation for LRD and LOF values.

LRD Computation: (MapReduce job 5). Mappers read in the output from the previous job, and assign each point to the corresponding partition based on the partitioning information. Each reducer then inserts the received points (both core and support points) into

a hash table with the point *ID* as key and its respective k-distance as value.

Now for the core points in each partition, we have sufficient information to calculate the *LRD* scores. Even if the neighbors of a core point include any support points, each such support point now also has a k-distance value associated with it. Furthermore, the k-distances of each of p's neighbors can be located in constant time utilizing the hash table.

LOF Computation: (MapReduce job 6). This process of calculating values for each point, and maintenance of intermediate results is repeated for the final step of LOF computation. Mapping is the same as in the previous job, and then reducers compute a final LOF score using the LRD values.

This intuitive strategy successfully allows for the intermediate values computed in a single phase of computation to be shared among machines in the cluster, and avoids a single lookup table, as is used in the centralized algorithm. The necessary information is effectively passed to each MapReduce job for further computation by extending the MPASS strategy of PkNN.

Chapter 5

Experimental Evaluation

All experiments are conducted on a shared-nothing cluster with one master node and 28 slave nodes. Each node consists of 16 core AMD 3.0GHz processors, 32GB RAM, 250GB disk, and nodes are interconnected with 1Gbps Ethernet. Each server runs CentOS Linux (kernel version 2.6.32), Java 1.7, Hadoop 2.4.1. Each node is configured to run up to 4 map and 4 reduce tasks concurrently. Speculative execution is disabled to boost performance. The replication factor is set to 3.

Datasets. We utilize the *OpenStreetMap* [25] dataset to evaluate the performance of our strategy on real world data. *OpenStreetMap* is one of the largest real datasets publicly available and has been used in other similar research work [15]. It contains the geolocation information for physical landscape features such as a buildings and roads all over the world. As the default for our experiments two attributes are utilized, namely *longitude* and *latitude*. In addition, hierarchical datasets have been built to evaluate the scalability of PkNN with regard to the data size. A subset of Massachusetts is the smallest, then all Massachusetts, then New England, up a dataset containing all data from the western hemisphere. The number of data points gradually grows from 10 million to over 1 billion.

Synthetic data. Synthetic datasets are used evaluate PkNN on data with a varying

number of dimensions. To produce realistic synthetic data sets, data clusters with varying sizes and random position are constructed following Zipf distribution as described in [26]. Then a small amount of noise ($< 0.01\%$ of the dataset size) is added via uniform random sampling of the domain space, introducing outliers which do not conform to the distribution of the rest of the data. Datasets with increasing numbers of attributes, from 3 to 10, were generated.

Methods. We compare our PkNN approach against the state-of-the-art distributed solution for kNN search in MapReduce [9] called PBJ. The authors of the [9] generously shared their code, which we adapted to run on a single dataset. We also adapted it to use the pivot index introduced in Section 3.3 to perform kNN search within local partitions. As shown by our evaluation in Section 5.1.1, this greatly increases the speed of the search and allows for a fair comparison of the PBJ and PkNN methods. All methods are implemented in Java.

Metrics. End-to-end execution time is measured, which is common for the evaluation of distributed algorithms. Furthermore, the **execution time for the key stages** of the MapReduce workflow is broken down to evaluate the performance of different stages of computation. This includes the pre-processing time for pivot selection, data partitioning, and k-NN search in one or two steps. The other key metric we employ is the **duplication rate** across the whole dataset. This measure captures how effectively supporting areas are bounded. This value is calculated as the total number of core and support points processed divided by the total number of points in the input dataset.

5.1 Impact of Optimizations

We first evaluate the key optimizations of PkNN individually: the pivot index introduced in Section 3.3, the early output optimization given in Section 3.2, and the use of support cells given in Section 3.1.3. These experiments measure the impact of these techniques on the performance of PkNN. All subsequent experiments employ these strategies to speed execution time.

A subset of the OpenStreetMap Massachusetts dataset containing 10 million points is used for the experiments in this section. The number of pivots is set to 100, and k is set to 5. To perform a fair comparison of PkNN and PBJ, they are required to use the same pivot set for each experiment, as the choice of pivots and number of partitions impact performance significantly. Therefore we omit the grouping step introduced in [9], which starts with a very large number of pivots and then groups them into a small number of final partitions. As their own evaluation shows, this grouping step only results in a modest decrease in execution time [9].

5.1.1 Evaluation of Pivot Index for kNN Search

First we evaluate the performance of the pivot index introduced in Section 3.3 to facilitate kNN search within local data partitions. PkNN and PBJ, both implemented using the pivot index, are compared to versions called PkNN-NL and PBJ-NL which use a nested-loop exhaustive kNN search within partitions. On the left, Figure 5.1 shows the total execution time in log scale for the kNN search step of each method (whether performed in one MapReduce job in PBJ or two jobs in PkNN).

This experiment reveals a number of insights. The pivot index provides a significant improvement in runtime for both distributed search methods. A 100x speedup over the naïve method for kNN search is observed when the pivot index is used for PkNN. We

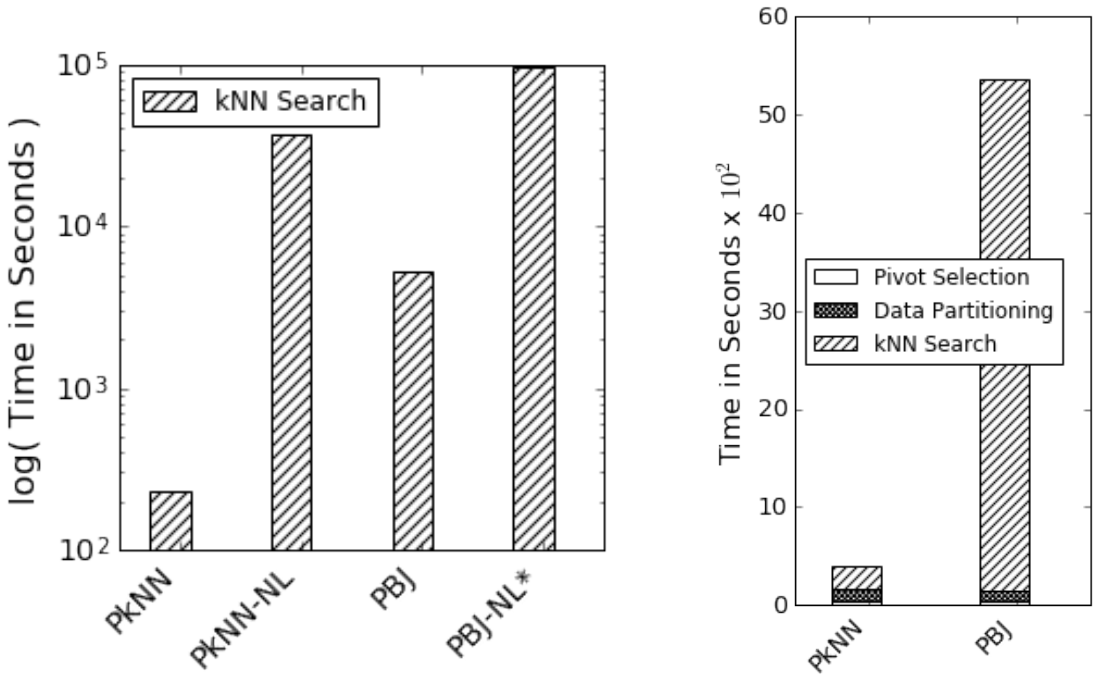


Figure 5.1: On the left the pivot index is compared to a nested loop implementation for kNN search. On the right, all stages of computation for PkNN and PBJ using the pivot index are compared.

note that the execution time of PBJ-NL is an underestimate since execution was stopped after 27 hours, however even using this conservative estimate, the PBJ kNN search phase using the pivot index was over 18x faster than the nested loop version.

On the right of Figure 5.1 all stages of computation for PkNN and PBJ are compared. PkNN achieves a speedup of over 20x the runtime of PBJ, even when both methods use pivot indexing. As subsequent experiments show, this decrease in time of execution is directly correlated with the decreased data duplication rate made possible by the MPASS solution.

5.1.2 Evaluation of Early Output and Support Cells.

Next the impact of two key optimizations of the PkNN method are evaluated. Figure 5.2 compares three versions of PkNN. The first called PkNN-NoOpts does not use the early

output evaluation given in Section 3.2, and does not use support cell determination, as described in Section 3.1.3. That is, Lemma 3.1.3 is evaluated for every point outside each cell, instead of using Lemma 3.1.4 to first perform support determination at the cell level.

The next method called PkNN-Early introduces early output to the method. We can see we get a modest improvement in execution time using this technique. The computational cost to evaluate Lemma 3.2.1 appears to mitigate some of the savings in communication costs that early output achieves. Finally PkNN-Support introduces the use of support cells to the PkNN-Early method. Here a performance boost of a factor of 5 is observed. By performing a 2-tier support determination evaluating first cells and then individual points there is a savings in both computational and communication cost. Both early output and support cells are used in all subsequent experiments.

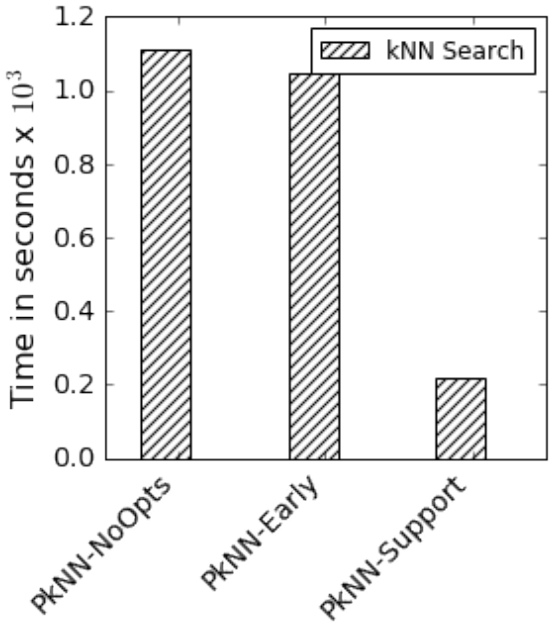


Figure 5.2: Impact of early output and support cell optimizations for the kNN Search phase of PkNN.

5.2 Impact of Parameters.

The parameters used in the PkNN and PBJ methods are next evaluated. Experiments are conducted on the small Massachusetts dataset containing 10 million points. This data set has areas of varying density throughout the domain space. First, k is fixed at 5 and the number of pivots is varied from 50 to 900.

5.2.1 Impact of Pivot Number.

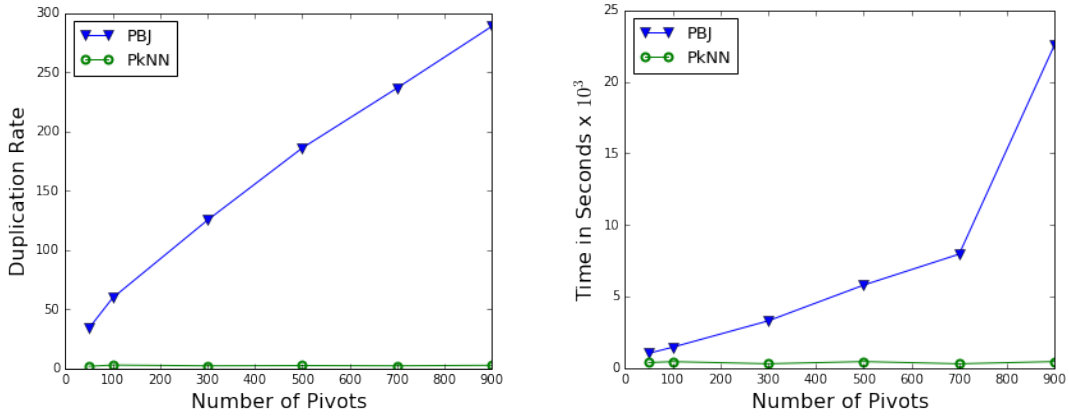


Figure 5.3: Impact of number of pivots on PBJ and PkNN methods.

To evaluate the impact of the number of pivots selected, both the end-to-end runtime as well as the data duplication rate are measured for PkNN and PBJ. Figure 5.3 shows the MPASS method employed by PkNN clearly results in much lower data duplication (shown on the right), and consequently execution time (shown on the left) compared to the state-of-the-art. Even using a small number of pivots, the minimum duplication rate for PBJ is around 30. The number of pivots is increased to evaluate the rate at which data duplication increases with the number of partitions. For a truly scalable solution, the benefit of adding partitions (necessary as the data size grows) should not be outweighed by the increased communication costs. The PBJ method shows a quadratic increase in

running time. PkNN on the other hand increases linearly, and at a slow pace. Figure 5.3 shows that the duplication rate closely corresponds to the running time of the algorithm.

5.2.2 Impact of k Number of Nearest Neighbors.

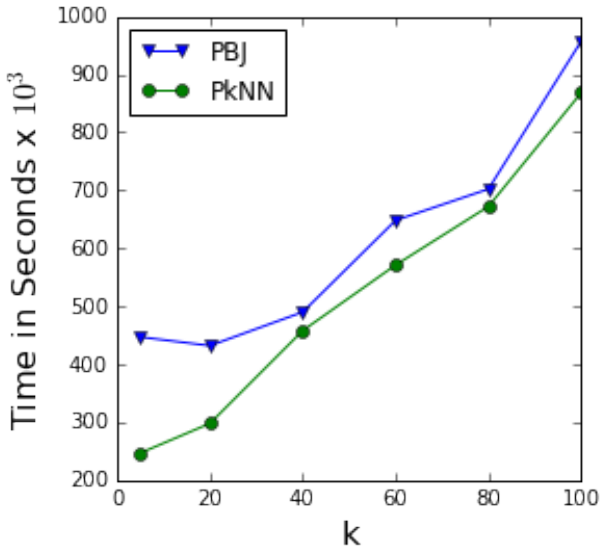


Figure 5.4: Impact of parameter k (number of neighbors) on end-to-end execution time for PBJ and PkNN methods.

Next the impact of the parameter k is evaluated. For this experiment a subset of the small Massachusetts dataset containing 3 million points is used. The number of pivots is fixed at 50, and the value of k increased from 3 up to 100. The running time of PkNN and PJB is shown in Figure 5.4. The duplication rate is not impacted by the parameter k. Here, an advantage of the single job solution in PBJ is revealed. Although PkNN has a large benefit over PBJ at low values of k due to the lower rate of duplication, this is mitigated at higher values due to the communication costs between the two kNN search phases of the MPASS solution. Since neighbor information is embedded with each data point after the first search step, much more intermediate information is communicated from mappers and reducers in the second kNN search step. For values of k close to the

number of pivots used, the running time for the two methods appears to increase at about the same rate. This implies that for very large data, since the number of pivots will be high compared to even large values of k, the benefit of PkNN will be more pronounced.

5.2.3 Impact of Number of Dimensions.

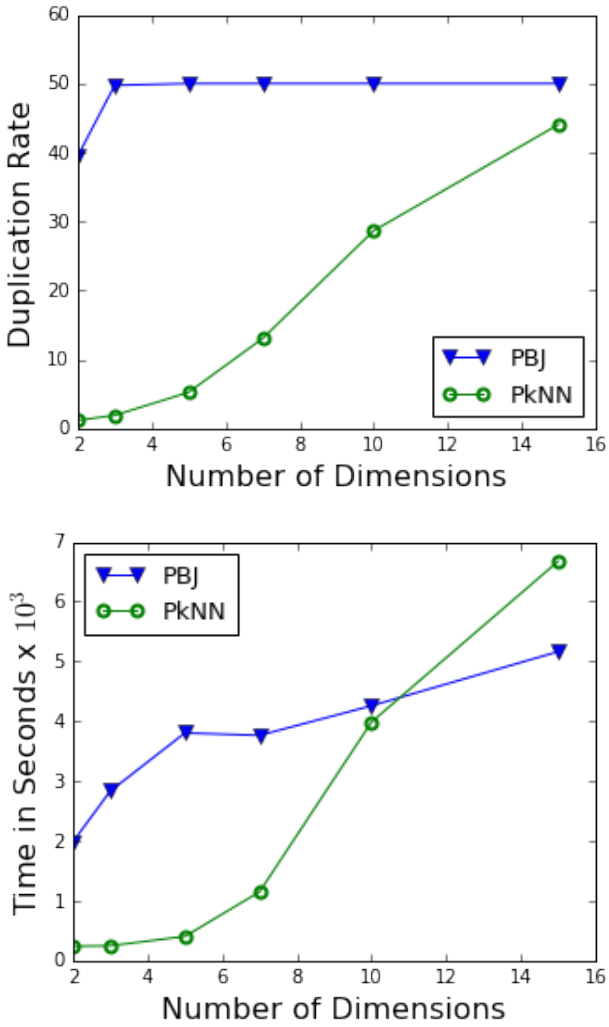


Figure 5.5: Impact of data dimensionality on PBJ and PkNN methods.

The performance of the two methods is compared on synthetic datasets containing 3 million points. The number of dimensions increases from 2 to 15. Figure 5.5 compares the

duplication rate and running time respectively of both methods on the synthetic generated data. 50 pivots are used and k is set to 3.

To interpret these results we note that the maximum possible duplication rate is exactly equal to the number of pivots used (when all data is sent to every partition). Figure 5.5 shows that the bounds used in the PBJ method are ineffective at limiting data duplication on dimensions greater than 2. On higher dimensional data the duplication rate is close to or equals to the number of pivots, meaning that the entire dataset is duplicated and sent to each reducer. The duplication rate for PkNN remains under 30x the size of the dataset. However, it can also be observed that the running time of PkNN degrades to the worst case at 10 dimensions. Poor performance of distance based methods above 10 dimensions has been has been shown to be a common phenomena [17], therefore this result is not surprising. At 15 dimensions, PBJ outperforms PkNN, indicating that the communication cost of the large amount of intermediate data in the PkNN method negatively impacts the execution time at high dimensions.

5.3 Scalability.

The experiments in Section 5.2 clearly show that PkNN method outperforms the state-of-the-art. Attempts to evaluate PBJ on data larger than 10 million points were not successful due to job failures caused by the high data duplication rate. In this section the scalability of PkNN on bigger data is evaluated. For our next set of experiments we use 2d data from the Open Street Map dataset and fix k at 3. To choose the number of pivots, it was observed that a ratio of 1:100,000 performed well on the small dataset in initial experiments. Therefore the number of pivots for each dataset was chosen accordingly. Table 5.1 shows the datasets used for evaluation.

Dataset	Number of Points	Number of Pivots	Mean Partition Size
Small Massachusetts	10,000,000	100	100,000
Massachusetts	31,136,409	300	103,788
Northeast	80,464,841	800	100,581
North America	812,233,510	8,000	101,529
Western Hemisphere	1,185,194,762	10,000	118,519

Table 5.1: Dataset Sizes.

Figure 5.6 shows the scalability of PkNN in the size of the input dataset up to 1 billion points. This demonstrates the MPASS solution used in the PkNN method facilitates processing of data orders of magnitude larger than the previous state-of-the-art, and truly scales to handle modern data quantities.

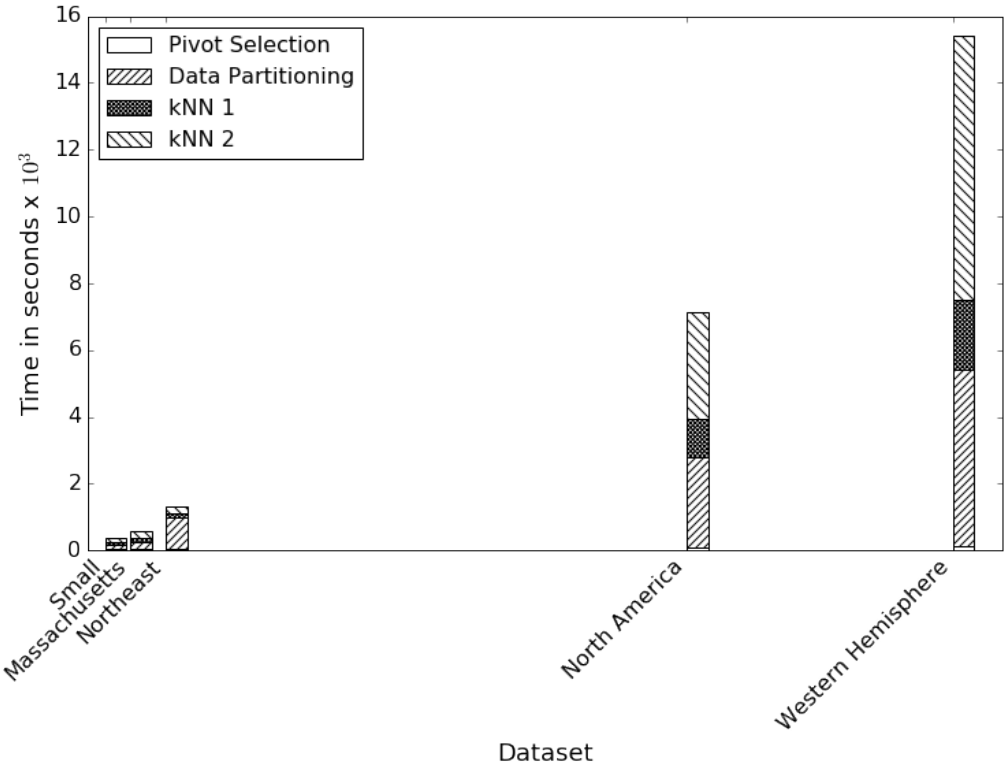


Figure 5.6: Impact of data size on overall runtime of PkNN method.

5.4 PkNN-LOF Evaluation

Section 4.2 describes a framework for the distributed computation of LOF scores using an extension of PkNN. Here we evaluate the performance of the method. The small Massachusetts dataset is used with 25 pivots and k is fixed at 3. Figure 5.7 shows the runtime for each stage of computation in PkNN: Pivot Selection, Data Partitioning, and two kNN Search Phases. The PkNN-LOF framework includes two additional phases of computation to compute LRD and LOF values for each point. It can be observed that the kNN search step of the LOF algorithm is indeed the bottleneck for computation, as the LRD and LOF phases outperform the runtime of both kNN search steps.

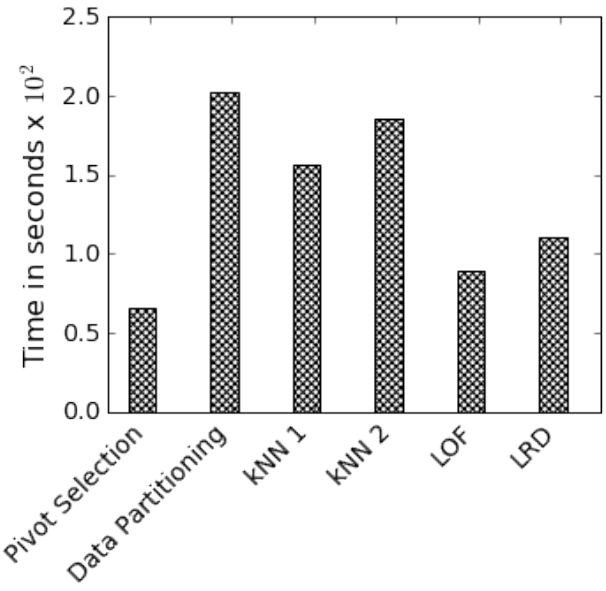


Figure 5.7: Breakdown of PkNN-LOF computation on Massachusetts dataset.

Chapter 6

Related Work

6.1 Distributed kNN Search

kNN search is a well-studied problem with many solutions to mitigate the high O(n^2) complexity of the problem. In centralized algorithms, spatial indexing structures are typically used, such as grid-based indices for low dimensional data, or hierarchical tree structures [24, 27, 28], as they can achieve an O($nlogn$) expected complexity. Pivot-based partitioning in centralized solutions has been shown to perform well for data with varying distributions [29] and higher dimensions in [23], where the authors focus on creating a spatial index for high dimensional data. Recursive data partitioning is used along with VA-File quantization to handle data with upwards of 50 dimensions. This centralized technique confirms the use of pivot-based partitioning as appropriate in higher dimensional spaces, as well as for use with other distance measures than Euclidean, such as the Mahalanobis distance which they evaluate.

In recent years, a number of distributed solutions for kNN search have been proposed for message-passing and peer-to-peer distributed systems [7, 30, 31]. However, shared-nothing distributed architectures such as Hadoop [20] and Spark [32] are preferred due

to their scalabilty, fault tolerance, and ability to run on commodity hardware. An exact distributed solution to the kNN join in MapReduce is presented in [15]. In this approach, the dataset is divided into blocks and all possible pairs of blocks are formed, each corresponding to one data partition. In this way, every data point is compared with all other points in the dataset. As the authors note in [15], this method is impractical on large data due to the quadratic number of partitions.

To address the shortcomings of this exact solution, the authors of [15] then propose an approximate solution using space filling curves to partition data while retaining approximate proximity. As in our PkNN method, points near the boundaries of partitions are duplicated and sent to multiple partitions. Supporting points are approximated using the z-curve index. Another approximate method uses Locality Sensitive Hashing [12] to hash nearby points to the same partitions with high probability. This solution is designed to process single queries at a time and only the point under consideration is duplicated and sent to multiple partitions. In this scenario, high data duplication is not a concern, as opposed to our problem where the kNN of all points in the dataset are found at the same time.

The current state of-the-art solution [33] for exact distributed kNN search is given in [9]. This approach also uses pivot-based partitioning to divide the dataset. Then worst-case estimation based on the distance from the pivots to a small number of points in each partition bounds the size of supporting areas. As shown in their experiments and confirmed by our results, this leads to extremely high data duplication, upwards of 20x the size of the original dataset. Our data-driven PkNN approach over multiple MapReduce jobs instead establishes a much tighter bound on supporting areas. Using this bound, PkNN achieves data duplication rates close to 1 on datasets orders of magnitude larger that those previously evaluated (See experimental evaluation in Section 5.3).

kNN search is also used as a core step in a number of different machine learning algo-

rithms. For instance, DBSCAN clustering searches for the nearest-neighbors of a point, and checks if more than *minPts* are within a certain distance *eps*. In this case, the area that needs to be checked for neighbors is *fixed across all partitions and pre-determined* by the user specified parameter *eps*. This simplifies the problem in a distributed setting, as this fixed value can be used to predetermine the supporting area size [34]. For other methods which depend on the distance to the kth nearest neighbor, such as distance-based [6] and density-based outliers [5] the size of supporting areas may vary across partitions. Our proposed PkNN method can be leveraged to facilitate distributed solutions for the-https://preview.overleaf.com/public/syhbdvhnbxwn/images/a4e27a99e1d0049f0913445a7419d6f4dcaf8a62.j algorithms as well, as is demonstrated by the PkNN-LOF framework presented in this work.

6.2 Distributed LOF

Breunig et al. developed the notion of local outliers in opposition to that of distance-based outliers proposed by Knorr and Ng in [16]. To overcome what they saw as shortcomings of a binary definition of outliers, they sought to define a degree of outlier-ness. One method [6] had been proposed to rank outliers using nearest-neighbor relationships, however it still hinged on a distance measure. Inspiration for LOF came from density based clustering algorithms including DBSCAN [3] and BIRCH [35]. These methods can identify outlying points, however they classify them as noise.

As best can be determined, no distributed LOF algorithm has been proposed to date. In [36], Lozano and Acunna proposed a multi-process LOF algorithm on one single machine. All processes share the disk and main memory and therefore can access any data in the dataset at any time. Clearly this approach cannot be adapted to popular shared-nothing distributed infrastructures targeted by our work.

Distributed solutions have been proposed for distance-based outliers. Hung and Cheung [37] give a parallel version of the basic *Nested-loop* algorithm [16]. This method requires strict synchronization between worker nodes, assuming all nodes can communicate with each other by message passing. Angiulli et al. [4] presented a distributed algorithm for another major variation of distance-based outliers [6]. It utilizes a solving set of points sampled from the original dataset to approximate whether a given point p is an outlier by comparing p to only the elements in this sample set. This *approximate* result requires the solving set to be broadcast to each node.

Another solution from Bhaduri et al. [13] utilizes a ring overlay network architecture. Their algorithm passes data blocks around the ring allowing the computation of neighbors to proceed in parallel. Along the way, each point's neighbor information is updated and distributed across all nodes. A central node maintains and updates the top-n points with the largest k nearest neighbor distances.

Unlike our solution, reliance on special architectures and message passing between nodes limits the scope of the applicability of the above mentioned methods. In general, these strategies are not practical for MapReduce-like shared nothing infrastructures which do not feature a central node, and where mappers and reducers work independently from each other.

Chapter 7

Conclusion

This thesis presents PkNN, a distributed solution for kNN search. It demonstrates that the common assumption in the design of MapReduce algorithms that using fewer jobs results in lower costs does not hold in the context of kNN search. The MPASS method used for support area partitioning over multiple rounds of computation effectively limits data duplication and scales PkNN to billion point datasets. Pivot-based data partitioning is shown to facilitate data-driven bounds on supporting areas, and provide a flexible means of distributing data in a compute cluster. Optimizations for early kNN output and pivot-based indexing for local search are also given. Together, these components provide a scalable approach to distributed kNN search, and a frame of reference for further study in this area.

Chapter 8

Future Work

Additional Data Mining Tasks

The PkNN method presented in this thesis provides a distributed solution to a high-value data mining task: kNN search. As the PkNN-LOF case study demonstrates, the PkNN method can be extended to facilitate other data mining tasks which depend on a kNN search. In addition to outlier detection [5], future extensions could include clustering [2, 3] and classification [1] methods. The strategies outlined in this work can also provide a basis of study for pivot-based partitioning as applied to data partitioning for other distributed algorithms.

Pivot selection

In PkNN an intuitive strategy to select pivots is presented. Uniform random sampling is used to select a subset of the data as pivots. With this strategy, the pivot set reflects the underlying data distribution, providing more examples chosen from dense areas as opposed to sparse. However, random sampling does not provide any guarantees that particularly dense clusters of data will be grouped into the same partition. In fact, since more pivots are likely to be chosen from dense areas, natural clusters of data are likely to

be broken up among many partitions. Pre-processing via clustering or vector quantization may help to identify pivots at the center of dense clusters, perhaps improving on random selection, albeit with an added overhead cost. While distributed clustering methods are available, the process is expensive.

Furthermore, it is not obvious whether it is preferred to have dense or sparse areas fall in the center of partitions, or along boundary divisions. On the one hand, having more data close to the partition edge means there will be many duplicated points in the support areas. On the other hand, the points in these areas will be close together, requiring only a small support area. In addition to density of the data, the volume of the cells also plays a role in the size of supporting areas. Therefore it may also be desirable to constrain the position of pivots relative to each other, for instance by choosing a regular lattice of points as described in Section 2.2. In this case the resulting Voronoi diagram yields a homogeneous partitioning, and the distance of data points to pivots becomes predictable.

Figure 8.1 shows some alternative strategies for pivot selection from the Massachusetts dataset. Future work could develop improvements to the pivot selection phase of PkNN, perhaps customizing the selection strategy to the input dataset.

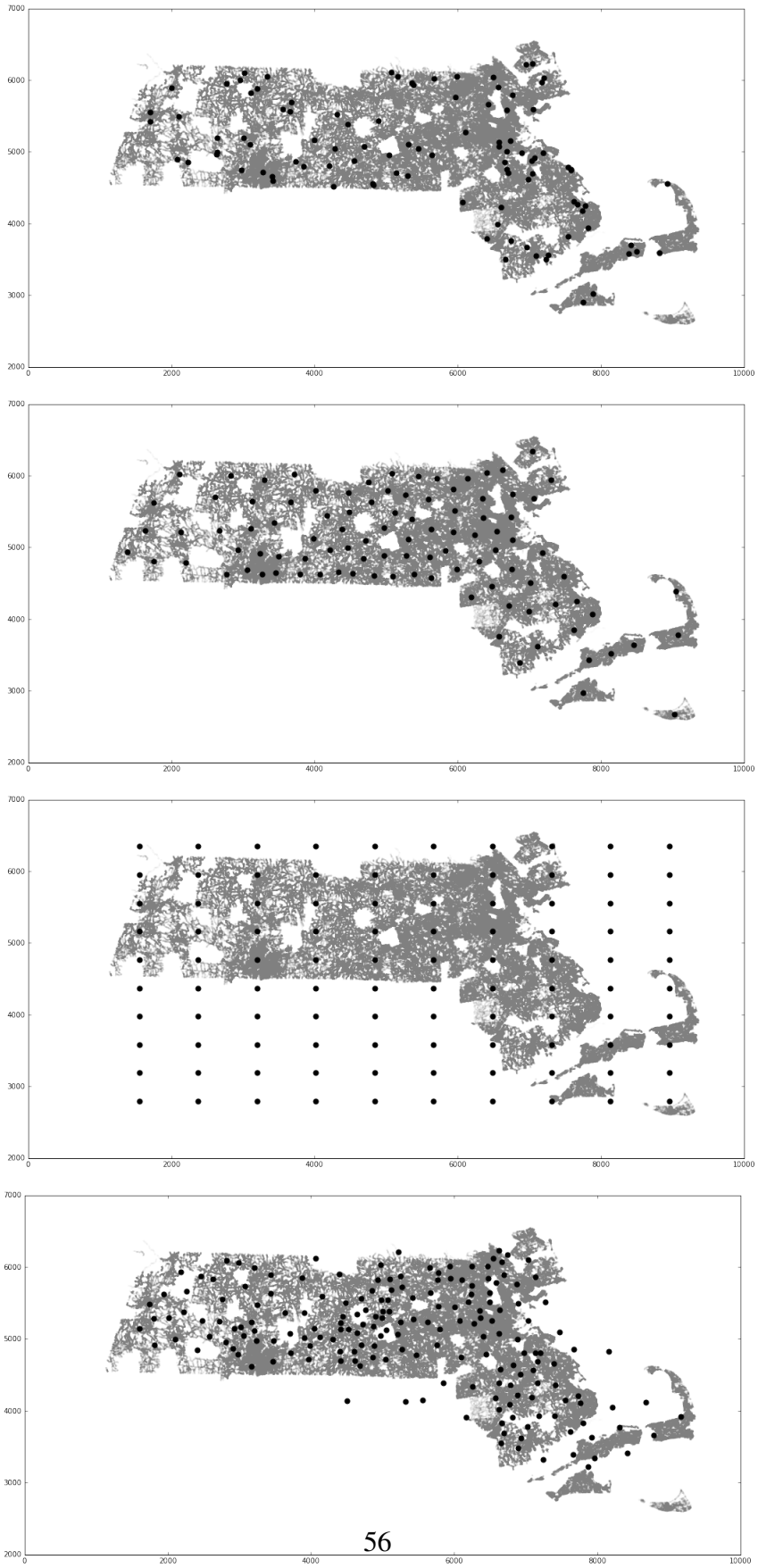


Figure 8.1: Various pivot selection strategies. From the top down: uniform random sampling, k-means clustering, lattice, and sparse sampling.

Bibliography

[1] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[2] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure," pp. 49–60, ACM Press, 1999.

[3] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise.," in *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining*, vol. 96, pp. 226–231, ACM, 1996.

[4] F. Angiulli, S. Basta, S. Lodi, and C. Sartori, "A distributed approach to detect outliers in very large data sets," in *Euro-Par 2010-Parallel Processing*, pp. 329–340, Springer, 2010.

[5] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: Identifying Density-based Local Outliers," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, (New York, NY, USA), pp. 93–104, ACM, 2000.

[6] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets," in *ACM SIGMOD Record*, vol. 29, pp. 427–438, ACM, 2000.

[7] G. Chatzimilioudis, C. Costa, D. Zeinalipour-Yazti, W.-C. Lee, and E. Pitoura, "Distributed in-memory processing of all k nearest neighbor queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 4, pp. 925–938, 2016.

[8] G. R. Hjaltason and H. Samet, "Index-driven similarity search in metric spaces (survey article)," *ACM Transactions on Database Systems*, vol. 28, no. 4, pp. 517–580, 2003.

[9] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1016–1027, 2012.

[10] A. Lazarevic, L. Ertöz, V. Kumar, A. Ozgur, and J. Srivastava, "A comparative study of anomaly detection schemes in network intrusion detection.," in *SIAM International Conference on Data Mining*, pp. 25–36, SIAM, 2003.

[11] A. F. Emmott, S. Das, T. Dietterich, A. Fern, and W.-K. Wong, "Systematic construction of anomaly detection benchmarks from real data," in *Proceedings of the ACM SIGKDD workshop on outlier detection and description*, pp. 16–21, ACM, 2013.

[12] A. Stupar, S. Michel, and R. Schenkel, "RankReduce – processing k-nearest neighbor queries on top of mapreduce," in *Proceedings of the 8th Workshop on Large-Scale Distributed Systems for Information Retrieval*, pp. 13–18, 2010.

[13] K. Bhaduri, B. L. Matthews, and C. R. Giannella, "Algorithms for Speeding Up Distance-based Outlier Detection," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 859–867, ACM, 2011.

[14] A. Koufakou, J. Secretan, J. Reeder, K. Cardona, and M. Georgiopoulos, "Fast parallel outlier detection for categorical datasets using mapreduce," in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pp. 3298–3304, IEEE, 2008.

[15] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," in *Proceedings of the 15th International Conference on Extending Database Technology*, pp. 38–49, ACM, 2012.

[16] E. M. Knorr and R. T. Ng, "Algorithms for mining distance-based outliers in large datasets," in *Proceedings of the 24th International Conference on Very Large Data Bases*, pp. 392–403, Morgan Kaufmann Publishers Inc., 1998.

[17] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is nearest neighbor meaningful?," in *International conference on database theory*, pp. 217–235, Springer, 1999.

[18] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys*, vol. 41, no. 3, p. 15, 2009.

[19] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[20] "Apache hadoop." `https://hadoop.apache.org/`. Accessed: April 23, 2016.

[21] A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman, "Upper and lower bounds on the cost of a map-reduce computation," in *Proceedings of the VLDB Endowment*, vol. 6, pp. 277–288, VLDB Endowment, 2013.

[22] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software*, vol. 11, no. 1, pp. 37–57, 1985.

[23] S. Ramaswamy and K. Rose, "Adaptive cluster distance bounding for high-dimensional indexing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 6, pp. 815–830, 2011.

[24] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, vol. 14, pp. 47–57, ACM, 1984.

[25] "Open street map." `http://www.openstreetmap.org/`. Accessed: April 23, 2016.

[26] C. R. Palmer and C. Faloutsos, *Density biased sampling: an improved method for data mining and clustering*, vol. 29. ACM, 2000.

[27] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pp. 426–435, 1997.

[28] K. I. Lin, H. V. Jagadish, and C. Faloutsos, "The tv-tree: An index structure for high-dimensional data," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 3, no. 4, pp. 517–542, 1994.

[29] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "idistance: An adaptive b+-tree based indexing method for nearest neighbor search," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 2, pp. 364–397, 2005.

[30] P. Haghani, S. Michel, P. Cudré-Mauroux, and K. Aberer, "LSH at large-distributed knn search in high dimensions," in *Proceedings of the 11th International Workshop on Web and Databases*, 2008.

[31] D. Novak and P. Zezula, "M-chord: a scalable distributed similarity search structure," in *Proceedings of the 1st International Conference on Scalable Information Systems*, p. 19, ACM, 2006.

[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets.," *HotCloud*, vol. 10, pp. 10–10, 2010.

[33] G. Song, J. Rochas, F. Huet, and F. Magoules, "Solutions for processing k nearest neighbor joins for massive data on mapreduce," in *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 279–287, IEEE, 2015.

[34] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan, "Mr-dbscan: an efficient parallel density-based clustering algorithm using mapreduce," in *IEEE 17th International Conference on Parallel and Distributed Systems*, pp. 473–480, IEEE, 2011.

[35] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," in *ACM SIGMOD Record*, vol. 25, pp. 103–114, ACM, 1996.

[36] E. Lozano and E. Acuña, "Parallel algorithms for distance-based and density-based outliers," in *Fifth IEEE International Conference on Data Mining*, pp. 729–732, 2005.

[37] E. Hung and D. W. Cheung, "Parallel mining of outliers in large database," *Distributed and Parallel Databases*, vol. 12, no. 1, pp. 5–26, 2002.