Scaling k-Nearest Neighbours Queries (The right way)

Atoshum Cahsai, Nikos Ntarmos, Christos Anagnostopoulos, Peter Triantafillou

School of Computing Science
University of Glasgow, G12 8QQ, UK
Email: a.cahsai.1@research.gla.ac.uk, {nikos.ntarmos, christos.anagnostopoulos, peter.triantafillou}@glasgow.ac.uk

***Abstract*—Recently parallel / distributed processing approaches have been proposed for processing k-Nearest Neighbours (kNN) queries over very large (multi-dimensional) datasets aiming to ensure scalability. However, this is typically achieved at the expense of efficiency. With this paper we offer a novel approach that alleviates the performance problems associated with state of the art methods. The essence of our approach, which differentiates it from related research, rests on (i) adopting a coordinator-based distributed processing algorithm, instead of those employed over data-parallel execution engines (such as Hadoop/MapReduce or Spark), and (ii) on a way to organize data, to structure computation, and to index the stored datasets that ensures that only a very small number of data items are retrieved from the underlying data store, communicated over the network, and processed by the coordinator for every kNN query. Our approach also pays special attention to ensuring scalability in addition to low query processing times. Overall, kNN queries can be processed in just tens of milliseconds (as opposed to the (tens of) seconds required by state of the art. We have implemented our approach, using a NoSQL DB (HBase) as the data store, and we compare it against the state-of-the-art: the Hadoop-based Spatial Hadoop (SHadoop) and the Spark-based Simba methods. We employ different datasets of various sizes, showcasing the contributed performance advantages. Our approach outperforms the state of the art, by 2-3 orders of magnitude, and consistently for dataset sizes ranging from hundreds of millions to hundreds of billions of data points. We also show that the key constituent performance overheads incurred during query processing (such as the number of data items retrieved from the data store, the required network bandwidth, and the processing time at the coordinator) scale very well, ensuring the overall scalability of the approach.**

***Keywords*-Big Data, kNN Query, NoSQL, Hadoop, Spark**

I. Introduction

We live in the era of big data, where devices are continuously generating large amounts of data, particularly multi-dimensional (m-d) data such as spatial data, geo-tagged data, etc. However, traditional methods that have been used for querying such (typically much smaller) datasets in centralized systems, have proved inadequate to handle ever growing data. The urgent need for *big data* analytics has led to the development of several distributed/parallel data-processing frameworks. Arguably, the most popular of such frameworks are Hadoop-MapReduce (MR) [7] and Apache Spark [27]. MR and Spark are designed for batch processing in parallel over a large cluster of commodity hardware and have been used to solve many large-scale data analytics problems scalably. In spite of their popularity, such frameworks cannot always provide an ideal solution for ad hoc big data querying. However, many a researcher continue to propose solutions for big data query processing based on these frameworks, even when this is flawed. As an example, let us consider using MR or Spark SQL [3] to compute k Nearest Neighbour (kNN) queries. Both approaches access the whole dataset regardless of the size of the dataset or the value of k. Briefly, executing kNN queries in this way is very costly in terms of query response times, memory usage, cpu usage, and network and disk bandwidth.

In order to process m-dimensional (m-d) kNN queries efficiently, several authors propose accessing only relevant subsets of the dataset at query time. The latest such works are the MR-based SHadoop [8] and the Spark-based Simba[23] systems. From a design philosophy point of view, both approaches: (i) divide a dataset into several partitions (subsets), each of which contains data elements that are located relatively close to each other in the Euclidean space; (ii) build a local m-d index over each data partition in order to avoid linear scanning of the partition; (iii) build a m-d global index over the entire dataset in order to prune out irrelevant partitions during query execution. This design philosophy improves the efficiency of query precessing.

However, the size of a partition is determined by the settings in which a particular method operates. For example [8] operates within the Hadoop ecosystem. As such, it defines the minimum size of a data partition to be at least as large as the block size of the Distributed File System (e.g., HDFS) where data reside; for HDFS, this translates to partitions being at least 128MB each (default HDFS block size). Similarly, [23] operates within the Spark ecosystem and thus calculates the size β of a partition as: $\beta = \lambda((1-\alpha)M/c)$, where λ is a system parameter (usually 0.8) capturing run-time memory overheads, c is the number of cores, M is the total memory reserved for Spark on each worker node, and α is the fraction of M reserved for RDD caching; thus, β is usually in the hundreds of MBs if not GBs. Regrettably, though, setting the minimum partition size to such large values has a negative impact on overall query processing time as explained below.

Motivating Example: Consider a kNN query over a check-in dataset of spatial points stored in HDFS. Assume each point is represented by X and Y coordinates in a 2-d space. Here, each coordinate is represented by a double precision floating point number, thus a point needs $2 \times 8 = 16$ bytes in total. During kNN query execution, when $k = 10$ (resp. $k = 100$ or $k = 1000$), the optimal would be that only 160 (resp. 1600 or 16000) bytes of data are to be retrieved as the final answer to the query. Unfortunately, the-state-of-the-art methods [8] [23] would process at least one partition (128 MB), containing $\approx 8.4 \times 10^6$ points to compute the final answer. This clearly shows that data organization, storing partitions into HDFS blocks (or similar), is highly inefficient due to the facts that: (i) even though most of the time the value of k is small, high volumes of data must be scanned and loaded into memory, and (ii) many points that are located relatively *far* from each other (in Euclidean distance) would be packed and stored together in one partition, solely in order to meet the lower size requirement of a partition. Consequently, as the DFS block size increases, the chance of compacting *non-neighbours* into one partition increases, and so does the chance of accessing data items that do not contribute to the final kNN answer, thus wasting time and resources.

But why the above approaches do not employ smaller partition sizes and/or smaller block sizes? When these methods run over HDFS, their rationale for enforcing such a constraint is guided by [1], [14], [30], [29]: the meta-data for all partitions in HDFS are managed by a centralized node called the NameNode; too many small files (one per partition) can easily overload the NameNode, thus potentially compromising the overall health of the cluster.

Departing from the existing systems that link the size of a partition to the DFS block size, we propose a *coordinator-based approach*, which partitions and indexes large-scale data into several small data partitions (hereinafter referred to as *cells*) whose size is determined only with respect to the desired performance of kNN query processing. Consequently, at query time, our approach is capable of surgically accessing significantly smaller subsets of the dataset.

Specifically, our major technical contributions are:

- Revisit the design philosophy that underpins kNN query processing over very large datasets, going against the grain and state-of-the-art.
- Offer a different way to index and organize the dataset that enables surgical accesses to only very small subsets of the dataset.
- Offer coordinator-based query processing algorithms that exploit the above and which overall ensure:
  - High performance: up to 3 orders of magnitude lower query processing times than the state of the art.
  - High scalability: ensuring that compute-storage-network resources are utilized efficiently, for datasets of various sizes, ensuring also high scalability.
- Offer an implementation and extensive performance evaluation of our approach versus the state of the art (Spatial Hadoop, Simba), which substantiates and quantifies our performance and scalability claims.

The paper is structured as follows: Section II reviews background and related work. Section III presents the problem analysis and fundamentals. Section IV explains the rationale of our approach while Sections V and VI elaborate implementation details. Section VII reports on experimental results. Finally, Section VIII concludes the paper.

II. Background and related work

A dataset is a collection of d-dimensional vectors, hereinafter referred to as *points*. We overview the background upon which all query processing methods rely. We then review the state of the art in detail.

A. Background

1) Multidimensional Indices: are crucial for locating data relevant to a given query without accessing the whole dataset. Traditional data indexing methods are not suitable for indexing m-d data, giving rise to several m-d indexing solutions [4], [5], [9], [11], [20], [16]. For concreteness, we adopt the Quad Tree (QT) [9] indexing method. QT can efficiently index uniformly and non-uniformly distributed points. When skewed data are provided, QT divides the most populated region recursively, so that all leaf nodes of the tree contain approximately an equal number of points.

2) kNN Query Processing: Most tree-based indexing methods have two steps in order to compute the kNN answer: (Step 1) Computing an initial solution, and (Step 2) Verifying correctness of said solution. In Step 1, the closest cell (leaf node) to the query point is identified by traversing the index tree. Subsequently, the k-nearest neighbours that reside in that cell are determined. In Step 2, a circle with a specific radius centred at the query point is considered. Any cell that overlaps with the circle is checked whether it contains some points, whose distance to the query point is less than any distance of a point that belongs to the initial kNN answer set from Step 1. If such a point exists in any of the candidate cells, then it is inserted into the kNN set and the furthest point is removed.

A body of research has focused on computing the optimal radius size. For example [19], also adopted by [2], estimates a radius size through the distance between the query point and the furthest *corner of the cell* which encompass the query point. This is achieved without accessing the dataset but only utilizing the information stored in the index. However, this estimated radius size might be quite larger than it should be, and thus a high number of candidate cells may be accessed at query time. A variant to this approach is discussed in [8], where the radius size is estimated by the distance between the query point and the k-th nearest point that lies within the cell that contains the query point. In this case, the dataset must be accessed to compute the radius size. The radius size could be smaller compared to [19] but several data accesses would be required to compute the kNN.

3) HBase: An Overview: Large-scale data can hardly be stored in a centralized server. Even large-scale distributed relational databases are viewed as non-scalable. Thus, typically modern distributed database systems, such as NoSQL databases or DFSs (such as HDFS) are being used. Arguably, HBase [10] is one of the most popular NoSQL databases, offering an implementation of the BigTable[6] model. HBase is highly available and scalable, provides a simple key-value API and is designed to store large datasets. We have chosen HBase in our implementation as the basic data store because it does not need to retrieve the entire DFS block into memory during query execution [21]. A table in HBase is divided horizontally (i.e., at rowkey boundaries) into regions, each of which, in turn, has several HFiles. An HFile stores a sorted list of key-value pairs on disk. Additionally, each HFile contains a simple index of the rowkeys it contains, and HBase keeps track of which storage nodes and regionservers are responsible for every region. These features enable HBase to support efficient random data accesses.

B. Related Work

1) Hadoop/MR-based Approaches: Hadoop GIS [1] is a scalable and high performance spatial data warehousing system for running large scale spatial queries in Hadoop. However, Hadoop GIS only supports 2-dimensional data. The state-of-the-art SHadoop [8] divides a dataset into a number of partitions, each of which has equal size to a HDFS block. SHadoop employs two indices: a global index and a local index, which are used to prune out irrelevant data elements. In order to answer kNN queries, SHadoop might require two MR jobs to ensure the correctness of the final kNN answer. Interestingly, although not discussed in the academic papers describing SHadoop, the source code the authors have (thankfully) made available also includes a non-MR approach – a sign that they also realize the tension between distributed operation and high performance. Nonetheless, to be fair, we will compare the performance of our approach against both variants of SHadoop (MR and non-MR). However, we should clarify that SHadoop as a whole is a good step forward for scalable spatial queries, offering an overall system for many types of spatial queries and not just kNN queries. The point we make in this paper is that the SHadoop approach is lacking in terms of performance for kNN query processing and that our approach reconciles the performance-scalability tension better – we do not discuss other types of spatial queries, such as kNN joins, spatial joins, etc. AQWA[2] is another recent method for KNN query processing. Like SHadoop, AQWA splits the dataset into many cells each of which has the same size as the block size of the underlying HDFS. Unlike SHadoop, AQWA only has a global index and does not employ local indexes within cells/partitions; all points that reside within a selected cell are loaded into memory and scanned one by one in order to deliver the kNN list. Therefore, it has significant extra CPU time overhead compared to SHadoop.

2) Approaches on HBase: Several methods have been proposed to expedite spatial query processing using HBase. The MD-HBase system [17] builds a m-d index over a dataset stored in HBase. MD-HBase uses k-d trees and QTs to quantize the space, and Z-ordering to convert m-d points into 1-d rowkeys. The system in [13] proposes a novel key-formulation schema, based on R+ trees over a dataset stored in HBase. These studies investigate how to effectively access a dataset by employing m-d indices. The main focus of both of the above works is different from ours: they stress on design of HBase row key. HGrid [12] builds a m-d hybrid index structure over HBase, using QT and a regular grid. HGrid adapts QT to partition the space into a number of sub-spaces each of which is further divided into several cells using regular grid. The leaf nodes of the QT correspond to rows in HBase, while each cell of the regular grid corresponds to a column in a row. In addition, HGrid stores a small number of points per cell to improve query response time. Our approach differs from HGrid in several ways: (1) HGrid does not include a systematic way of ensuring that a very small number of data points is accessed, and (2) to save memory space of the QT, HGrid opts to add many columns in a row; however, as the number of columns increases (e.g., above several hundreds), query performance deteriorates significantly [12].

3) Spark Based Approaches: Spark [27] is another cluster-based batch-oriented big data-parallel processing platform. The main advantage of Spark is the ability to run computations in memory. Spark defines resilient distributed datasets (RDD). RDDs represent a collection of items distributed across many nodes that can be manipulated in parallel. Spark has several extensions that provide different features: Spark SQL [3] for working with structured data, Spark Streaming [28] for processing of live streams of data, MLlib [15] with machine learning algorithms, and GraphX [24] for manipulating graphs.

In order to improve m-d kNN query processing in Spark, several works have been proposed: GeoSpark [26], SpatialSpark [25] (kNN joins and spatial joins over geometric objects), and Simba [23]. Simba [23] extends Spark SQL by adding specialised spatial indices and by using them during query planning and execution. More specifically, Simba employs global and local indices in order to access relatively small amount of data. According to its authors, Simba has the best performance compared against all above Spark-based competitors. Due to this fact, we decided to compare our solutions against Simba.

Compared to our solutions, Simba has two drawbacks. First, considering the value of k is small, Simba needs to load relatively large RDDs in memory; the main focus of indexing in Simba is to reduce only the CPU cost, failing to reduce disk IO and networking costs. Second, during query execution, the global index of Simba might select unnecessary RDDs; this is due to the fact that the circle centred at the query (q), which is used to identify relevant RDDs, has a much larger radius than needed, because it is computed as the distance between q and the furthest corner of R_i, where R_i is the closest RDD (partition) to q.

III. PROBLEM FUNDAMENTALS

Definition 1: A cell C in a d-dimensional space $\mathbb{R}^d$ is defined by the triplet:

$$C := \langle \mathbf{w}, r, |C| \rangle,$$

where $\mathbf{w} = [w_1, \ldots, w_d] \in \mathbb{R}^d$ is the lower boundary point, $r > 0$ is a fixed width in each dimension, and $|C|$ refers to the number of d-dimensional points in the cell.

Definition 2: A grid G in a d-dimensional space $\mathbb{R}^d$ is a set of m non-overlapping cells $G = \bigcup_{i=1}^{m} \mathcal{C}_i$.

Definition 3: A query point $\mathbf{q} \in \mathbb{R}^d$ is a d-dimensional vector: $\mathbf{q} = [q_1, \ldots, q_d]$; a point $\mathbf{p}$ in grid G is a d-dimensional row: $\mathbf{p} = [p_1, \ldots, p_d]$. The Euclidean distance between query $\mathbf{q}$ and point $\mathbf{p}$ is:

$$\|\mathbf{q} - \mathbf{p}\| = \left(\sum_{i=1}^{d} (q_i - p_i)^2 \right)^{\frac{1}{2}}.$$

Definition 4 ([19]): The minimum distance of a query point $\mathbf{q}$ from a given cell $C \in G$ with lower boundary point $\mathbf{w}$ and width r, denoted by $f(\mathbf{q}, C)$, is:

$$f(\mathbf{q}, C) = \|\mathbf{q} - \mathbf{s}\|,$$

where $\mathbf{s} = [s_1, \ldots, s_d]$ and

$$s_i = \begin{cases} w_i, & \text{if } q_i < w_i; \\ w_i + r, & \text{if } q_i \geq w_i + r; \\ q_i, & \text{otherwise.} \end{cases}$$

Definition 5: Given $m > 0$, a dataset $\mathcal{D} = \{\mathbf{p}_1, \ldots, \mathbf{p}_{|\mathcal{D}|}\}$ of d-dimensional points is divided into m partitions $\mathcal{D}_i, i = 1, \ldots, m$, such that it holds: $(\mathcal{D} = \bigcup_{i=1}^{m} \mathcal{D}_i) \wedge (\mathcal{D}_i \neq \emptyset) \wedge (i \neq j \Rightarrow \mathcal{D}_i \cap \mathcal{D}_j = \emptyset)$.

$|\mathcal{D}|$ is the cardinality of the set $\mathcal{D}$.

Definition 6: Given $\alpha > 0$, the upper-bound of points stored in a partition $\mathcal{D}_i$ is $\alpha \geq |\mathcal{D}|/m$ where m is the number of partitions.

Definition 7: Given a partition $\mathcal{D}_i$, cell C_i is the smallest (sub)space within which all points of $\mathcal{D}_i$ lie.

Definition 8: Given a balanced tree data structure, let x denote the maximum number of children per node. In a tree of height h, the total number of nodes z and the number of leaf nodes l are, respectively:

$$z = \sum_{i=0}^{h} x^i \ , \ l = x^h.$$

Definition 9: Given a query point $\mathbf{q}$ and a dataset $\mathcal{D}$, the k Nearest Neighbours (kNN) of $\mathbf{q}$ is the set $\mathcal{A}$:

$$(\mathcal{A} \subseteq \mathcal{D}) \wedge (|\mathcal{A}| = k) \wedge (\forall \mathbf{p} \in \mathcal{A}, \forall \mathbf{p}' \in \mathcal{D} \setminus \mathcal{A}, \|\mathbf{p} - \mathbf{q}\| \leq \|\mathbf{p}' - \mathbf{q}\|).$$

Definition 10: Let a vertical or horizontal closed interval in a number line in 1-dimensional space starts at 0 and be divided into *n* finite consecutive half-open smaller intervals, each of which has equal length *r*. For a given random number q that lays on the i^{th} small interval, the starting number of the i^{th} interval is defined by $\lfloor \frac{q}{r} \rfloor \cdot r$.

IV. RATIONALE

In this section we focus on designing cells that are as small as possible and discuss the implications on scalability.

A. Cell Size Determination

Quad tree (QT) divides a dataset $\mathcal{D}$ into m cells, each of which contains at most α points. Every cell, C_i corresponds to a partition $\mathcal{D}_i \subset \mathcal{D}$, thus $|\mathcal{D}_i| \leq \alpha$. Every cell (tree leaf node) of the QT is associated with a unique corresponding row in the key-value data store (HBase table). Equation (1) shows an upper bound on the cell size α, given $|\mathcal{D}|$ points and the total number of cells (leaf nodes) is x^h:

$$\alpha \geq |\mathcal{D}|/x^h, \tag{1}$$

where x is the maximum number of children per node (4 for QTs). The maximum number of points α stored in a cell has significant impact on query response time. The higher the value of α, the higher the query response time as there are more points to consider. Hence, the value of α is desired to be small in order to improve query response time. But smaller values for α have negative implications for scalability: the coordinator has a finite memory available to store the index and smaller α values increase the size of the index; for very large dataset sizes this will pose scalability problems at the coordinator.

Question 1: How small can the value of α be? The value of α is dependent on the amount of available memory, β, at the coordinator. In order to determine α using (1), before constructing the index tree, the height of the tree h should be defined in terms of β. Hence, α is defined as a function of β, i.e., $\alpha = \alpha(\beta)$, as shown in (5). In Lemma 1 we provide a determination of α in terms of β.

Lemma 1: Let β, b, x and z be the total available memory, the size of a node of the tree in bytes, the maximum number of children per node, and the total number of nodes in a tree, respectively. The upper bound on the number of points α that can be stored in a cell (leaf node) is:

$$\alpha(\beta) \geq |\mathcal{D}| \times x^{1 - log_x((\beta/b)(x-1)+1)}$$

Proof: The total number of nodes in a given tree is:

$$z = \sum_{i=0}^{h} x^i = 1 + x + x^2 + ... + x^h \tag{2}$$

Also z can be determined based on a maximum available free memory, β and the size of a node of a tree in bytes, b,

$$z \leq \frac{\beta}{b} \quad (3)$$

From (2) we obtain that:

$$\begin{aligned} z &= \frac{(x^{(h+1)}-1)}{x-1} \iff \\ (h+1)log_x x &= log_x(z(x-1)+1) \iff \\ h &= log_x(z(x-1)+1)-1 \quad (4) \end{aligned}$$

Given that $\alpha(\beta) \geq |\mathcal{D}|/x^h$ and substituting z from (3):

$$\alpha(\beta) \geq |\mathcal{D}| \times x^{1-log_x((\beta/b)(x-1)+1)} \quad (5)$$

■

In (5), when the value of β increases, the value of α decreases. Therefore, the value of α decays exponentially w.r.t β. Fig. 1 shows that $\alpha(\beta)$ is an exponential decay function ($|\mathcal{D}| = 90 \cdot 10^9$, $x = 2$, $b = 32$ bytes).

B. Candidate Cells Determination

In general, there will be cases where more than one cell must be accessed in order to answer a kNN query. Without loss of generality, assume a query point $\mathbf{q}$ lies within a cell C_c close to a boundary line. Possibly, some points that reside in adjacent cells to C_c might be part of the kNN list. Accordingly, adjacent cells to C_c should be checked in order to answer a kNN query correctly.

Question 2: Is it possible to identify the *relevant* cells that contain relevant points without accessing the whole dataset?

If so, w.r.t improving query response time:

Question 3: How can the smallest possible number of adjacent cells be identified?

Those are fundamental questions in order to compute the kNN list efficiently and are addressed in this section. As mentioned, the most popular technique to identify adjacent cells to the cell C_c that overlaps q is by creating a circle centred at $\mathbf{q}$ with a radius ρ. Then, all adjacent cells that overlap with the circle are selected as candidate cells. In the literature, methods for determining the value of the radius ρ include: (i) by the distance between $\mathbf{q}$ and the k-th nearest data element that lies within C_c [8], or (ii) by the distance between $\mathbf{q}$ and the furthest corner of C_c from $\mathbf{q}$ [2].

In our approach, the value of ρ is determined through the distance between $\mathbf{q}$ and the k-th nearest point $\mathbf{p} \in C_c$. We adopted this method due to two key facts. (F1) In spite of the fact that this approach requires to access the back-end data store twice, it has almost negligible overhead compared to the high initialization overhead cost of MR-based approaches [8]. (F2) Contrary to [2], which estimates the value of ρ generously, our approach calculates the tightest possible value of ρ, thus avoiding unnecessary data accesses as much as possible.

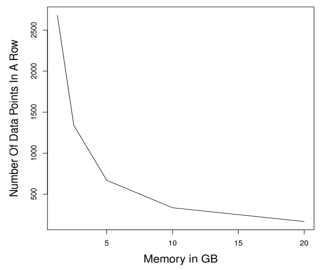
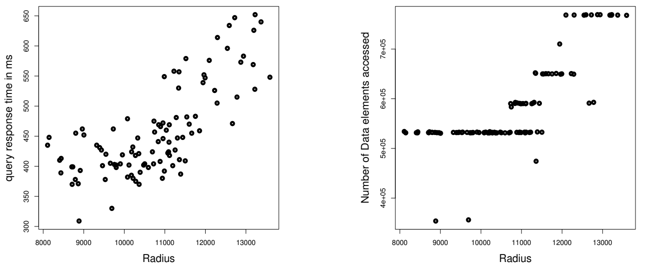


Figure 1: As β increases α decreases exponentially.



(a) Query response time (in ms.) vs. radius ρ.

(b) Number of accessed points vs. radius ρ.

Figure 2: Correlation radius (ρ) with (a) query response time and (b) number of accessed data points.

We study the impact of ρ on query response time as well as on the number of candidate cells. Using our approach and different values for ρ we generate the relevant plots in Fig. 2a and Fig. 2b. Fig. 2a shows that there is a strong positive correlation between ρ and query response time; i.e., as the value of ρ increases, query response time also increases. As illustrated in Fig. 2b, there is also a positive correlation between ρ and number of points that are accessed at query time. Hence, in order to reduce the value of ρ, unlike existing works [2], our approach calculates the exact value of ρ by accessing all points that reside in the closest cell.

V. Coordinator With Index (COWI)

We now turn to our coordinator-based distributed query processing algorithm, armed with the knowledge of the importance of accessing only small but relevant cells for query response times. The first solution, coined COWI, identifies relevant cells using a QuadTree (QT) index, without accessing or looking into the whole dataset.

QT accepts as input the entire dataset $\mathcal{D}$ and grid G, and hierarchically divides G into several sub-spaces $\mathcal{C}_i \subset G$ until each sub-space $\mathcal{C}_i$ contains less than α points; i.e., $|\mathcal{D}_i| < \alpha$ and $\mathcal{D} = \bigcup_{i=1}^{p} \mathcal{D}_i$. Each cell $\mathcal{C}_i$ is a leaf node of the QT. We store only the tree structure of the QT in the coordinator's memory. The actual data contained in a cell $\mathcal{C}_i$ are stored in a corresponding row in a HBase table. Every leaf node (cell) $\mathcal{C}_i$ is represented in the coordinator's memory by only the matching row key for the i-th row and the total number of points belonging to this row in the HBase table.

At query time, a query $\mathbf{q}$ traverses the QT starting from the root and descending to the child node that overlaps with $\mathbf{q}$ until it reaches a leaf node, $\mathcal{C}_i$. When a node $\mathcal{C}_i$ encompasses a query $\mathbf{q}$, the minimum distance between $\mathcal{C}_i$ and $\mathbf{q}$ is 0; i.e., $f(\mathbf{q}, \mathcal{C}_i) = 0$.

A. *QT Index Construction*

Initially, using equation (5) the value of α (maximum number of points in a cell) is determined based on the available memory. Then a *summary grid* is created with a fine granularity. The number of points that reside in a cell in the summary grid is counted using MR; therefore, each cell of the summary grid contains two important pieces of information: (i) the coordinates of the cell and (ii) the number of points that lie within the cell. The summary grid is used to construct a QT as follows: (**Step 1**) Assign each cell of the summary grid to a corresponding leaf node, $\mathcal{C}^*$, of the QT that overlaps with the cell completely; (**Step 2**) Increment the count of $\mathcal{C}^*$ by the number of points in the summary grid cell that has been assigned to it; (**Step 3**) If the total number of points in $\mathcal{C}^*$ exceeds α, split $\mathcal{C}^*$ into four children leaf nodes and redistribute all the summary grid's cells that have been assigned to $\mathcal{C}^*$ to the new leaf nodes based on the distance f; see [19]; (**Step 4**) Otherwise store the coordinates of the summary grid cell in $\mathcal{C}^*$. At the end of this process, the QT has its final structure. Then, all the coordinates of the summary grid cells that have been stored in each leaf node are deleted because those coordinates are needed only to identify which summary grid cells must be reassigned to which new leaf nodes when a node splits. At this point, the count of each QT leaf node refers to the total number of points that are going to be stored in the corresponding row in the HBase table.

B. *Query Processing: COWI*

KNN query processing proceeds as follows: (**Step 1**) Identify the closest cell, that is, $\mathcal{C}^* = \underset{\forall C_i \in \mathcal{C}}{\operatorname{argmin}} f(\mathbf{q}, C_i)$, and check if there are enough points (i.e., $\geq k$) in the row corresponding to $\mathcal{C}^*$. If there are not enough points, get the second closest, third closest and so on, until the total number of points that reside in the retrieved rows exceeds k. (**Step 2**) Retrieve all points that reside in the furthest cell $\mathcal{C}_f$ found in Step 1; then, compute the initial kNN answer. (**Step 3**) Use the distance from $\mathbf{q}$ to the k-th point in the initial kNN answer as a radius in order to draw a circle centred at $\mathbf{q}$. Then, retrieve all cells from the QT that overlap with the circle, and store them in a queue based on their distance. (**Step 4**) For each candidate cell in the queue, starting from the closest cell, check if there are points that reside in it and are located in a closer distance to $\mathbf{q}$ than points that are selected as members of the initial kNN answer. If so, add the closer points to the initial kNN answer and remove the furthest points from it. At the end of this process, the kNN answer is the final correct kNN list.

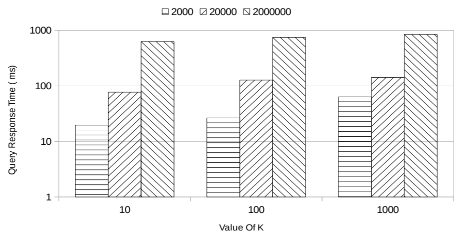


Figure 3: Three rows with different values of α and k.

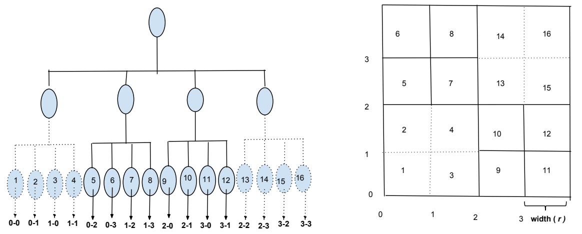


Figure 4: Conceptual nodes are connected by dotted line.

VI. Coordinator with No Index (CONI)

Tree-based indexing approaches might produce large index tree that might not fit in memory, especially when the value of α is small. Some times the storage cost of the index exceeds the dataset itself [22]. Therefore, tree-based approaches such as COWI, can not scale well with extremely big datasets because decreasing the storage cost of the tree requires increasing the value of α. However, as shown in Fig.3 when the value of α increases from 2,000, to 20,000 and to 200,000, significant increase in query processing time are observed for different values of k.

In order to avoid the scalability issue of COWI, we propose the coordinator with no index in-memory (CONI) approach. Storing some parts of the index tree in a key-value store table rather than in main memory makes CONI robust and scalable. The main advantage of this approach is, therefore, that both scalability and efficiency are achieved without sacrificing one for the other. Please note that CONI does not store any index in memory.

Indexing: In CONI there are two different indexing processes. (i) A dataset is indexed using a QT based on a small number of α that is determined only with respect to the desired performance of kNN as explained in VI-A and the contents of each leaf node are stored in a row in a key value table (coined the *data-table*). (ii) The row keys of the data-table are indexed separately (as if they are a separate dataset) and are stored in a row in a key value table (coined the *meta-table*). A row in meta-table contains several row keys of the data-table, each of which points to a row in data-table.

As a QT divides highly populated regions rigorously, the final structure of the QT might not be a balanced tree. In order the coordinator to be able to identify relevant rows of the meta-table without maintaining index in memory, the QT of the meta-table has to be a balanced tree, which means that all its leaf nodes must have equal width and length (see VI-A for more detail). To balance the QT of the meta-table, as shown in fig. 4, conceptual nodes (nodes that are connected by dot lines) are added. Thus, the QT has two types of nodes: actual nodes and conceptual nodes. Each actual leaf node contains several unique row keys of the data-table, whereas each conceptual leaf node only contains copies of row keys (of the data-table) inherited from its (actual) parent node. In general, CONI manages:

- to store a small number of points α per row data-table regardless of β and the size of the dataset;
- stores the index in meta-table
- does not store index in memory

It is important to note that, instead of using a second key-value "meta-table", one could think of using secondary storage at the coordinator for this. However, this would violate several tenets of big data systems as the index contents then would not be highly available, easily recoverable, and easily accessible. Hence we opted to use a second table in HBase for this purpose. At query time, CONI has an additional (relatively small) overhead cost compared to the COWI approach. This is due to the fact that, during query execution, CONI has to access two data store tables: the meta-table and data-table.

A. *How CONI Works*

In CONI the value of α is determined based on two principles: (i) in HBase it is more efficient to retrieve fewer 'fatter' rows than many 'thin' rows [13]; (ii) query $\mathbf{q}$ or the value of k may not be necessarily known in advance. However, we can still select a value for α that is large enough to accommodate a reasonable expectation that all kNN points will be included in a single data cell. For example, if k is expected to be up to 1000, α should be a (small) multiple of this value. The tension here is that we wish to have α as small as possible, but certainly larger than k-values that typical users are interested in; e.g, for k=1 to 1000 as mentioned in [18]. Accordingly, CONI determines the value of α based on the most frequently used value of k or the maximum value of k queried for so far. For our experiments, we used a value of $\alpha = 2000$.

Remark: Recall that the lower left coordinate of a cell is used as a row key. All cells that corresponds to the rows of the meta-table has equal height and width (balanced QT). According to definition 10, if the height or width of each dimension is divided into equal intervals then for any random number q the starting point of the i^{th} interval in which the random number lies can be found by $\lfloor \frac{q}{r} \rfloor \cdot r$ where r is the width or height of the intervals. Therefore, by applying definition 10, for any given point we can identify the row key of the meta-table, in which the point resides as in the following example.

Example: Assume a random point (2.05, 1.8) lies in the 10^{th} cell in Fig. 4. Suppose the cell in which the random point is contained is not known in advance and we are interested to find the row key of the cell. As shown in definition 10 applying $\lfloor \frac{q}{r} \rfloor \cdot r$ where r on each dimension of the point, the row key of the relevant cell can be determined as follows: given $r = 1$ the lower left x-coordinate of the relevant cell is $\lfloor (2.05/1) \rfloor \cdot 1$ and the lower left y-coordinate of the corresponding cell is $\lfloor (1.8/1) \rfloor \cdot 1$ that is, (2,1), which is exactly the lower left coordinate of the 10^{th} cell in figure 4.

In CONI, kNN query processing proceeds as follows: (**Step 1**) Identify the closest cell (winner meta-cell) of the meta-data table to the query point using $\lfloor (q \div r) \rfloor \cdot r$, and if there are not enough points in the data rows that lie within the winner meta-cell, using algorithm 1, get the second closest meta cell by assuming $\rho = 2 \cdot r$, third closest meta cell by considering $\rho = 3 \cdot r$ and so on until the total number of points that resides within those cell exceeds k (N.B.: due to space limitations we omit the proof of algorithm 1 but interested readers can consult Euclid's Elements, Book IV, Proposition 7, from which we adapt the algorithm). (**Step 2**) Retrieve all data rows keys that were identified in Step 1 and sort them in ascending order based on $f(\mathbf{q}, C_i)$ see definition 4. (**Step 3**) From the list of row keys that are identified in Step 2, select the short-listed candidates the closest least number of row keys which contain k or more points in total. Retrieve all points that reside in the furthest row from the short-listed candidates and compute the initial kNN answer based on the Euclidean distance. (**Step 4**) Use the Euclidean distance from $\mathbf{q}$ to the k-th point in the initial kNN answer as a radius in order to draw a circle centred at $\mathbf{q}$. Then, retrieve all points that reside in data-rows, whose keys are in turn stored in meta-rows, which overlap the circle using algorithm 1. (**Step 5**) Check if there are points that are retrieved in Step 3 and 4 and are located at a closer distance to $\mathbf{q}$ than those points that are selected as members of the initial kNN answer. If so, add them to the kNN answer and remove the furthest points. At the end, the kNN answer will contain the correct set of points.

VII. Performance Evaluation

A. *Experimental set up*

We now provide a comprehensive experimental study of the performance of our approaches (COWI and CONI variants) compared against the state of the art: SHadoop (SH) [8] representing the best solution from the Hadoop ecosystem, and Simba [23] representing the best solution from the Spark ecosystem. We used the publicly available code for both systems. Experiments were ran on a 5-node

Algorithm 1: Retrieve cells that intersect a given range circle

```
Input: query point $\mathbf{q}$, radius $\rho$, cell width $r$
Output: the set $\mathcal{P}$ of candidate cells
$\mathcal{P} = \emptyset$; // initialize candidate priority queue
double xmin = $\mathbf{q}$[0] - $\rho$
double xmax = $\mathbf{q}$[0] + $\rho$
double ymin = $\mathbf{q}$[1] - $\rho$
double ymax = $\mathbf{q}$[1] + $\rho$
for $i$ = $xmin$; $i \leq xmax$; $i$ += $r$ do
    for $j$ = $ymin$; $j \leq ymax$; $j$ += $r$ do
        rowKey = $\lfloor (Point(i,j)/r) \rfloor \cdot r$
        cell = Cell(rowKey, $r$)
        distance = $f(\mathbf{q}$, cell)
        if distance $\leq \rho$ then
            $\mathcal{P}$.put(cell, distance);
        end
    end
end
return $\mathcal{P}$;
```

cluster; each node is a Dell R720 server with 4 Intel Xeon(R) CPUs (8 cores each), 64GB RAM, and 9TB of disk space.

Datasets. We experiment with datasets of various sizes in terms of the number of two-dimensional $(d = 2)$ points.

We use ten synthetic datasets in our experiments. The first dataset contains around 600 million 2-d points with a total size of 20GB. The second dataset contains around 1 billion points with a total size of around 35GB. The third dataset contains circa 7 billion points, with a total size of 250GB. The fourth dataset contains around 29 billion points, with a total size of 1TB. The fifth dataset contains circa 100 billion points with a total size of 3.5TB. Note that in terms of total storage space, the fifth dataset is the largest dataset that our cluster could accommodate (as data needs to be stored in both HBase and HDFS). In the same way as in SH [8] all the above datasets are generated in an area of $1M \cdot 1M$ units and all the points are generated based on uniform distribution. We also generated another five datasets that have the same sizes, as explained above, but using a multi-modal distribution to generate data points.

As the performance results and conclusions remain the same across the different distributions, for space reasons, we report only the results for the uniformly distributed datasets.

As in [8] we randomly select 10^4 query points from the input files and issued over the datasets for different values of $k \in \{10, 100, 1000\}$.

Performance metrics. We measure the query response time in milliseconds (ms). Each method executes all queries sequentially and we compute the average query response time. We also considered three other qualitative measures: (i) average number of rows (cells) retrieved per query, (ii) average number of data points accessed per query, and (iii) the time that is required at the coordinator to process the contents of the retrieved rows and produce the final kNN answer. These three measures are all important to consider in order to showcase the scalability of the proposed design.

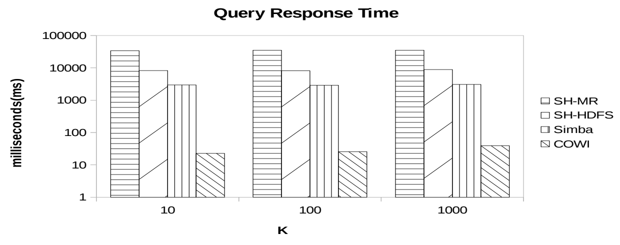


Figure 5: Dataset: 600 Million data points (20GB)

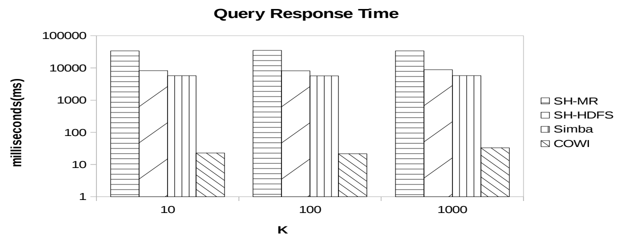


Figure 6: Dataset: 1 Billion data points (35GB)

SHadoop can employ two m-d alternative indexing methods: Grid and R-Tree. However, using the available code, indexing the datasets using MapReduce for the R-Tree index was not possible, as the MR processes would *hang* repeatedly during indexing of our datasets. For this reason, we compared our approach against SHadoop with Grid for the uniformly distributed datasets (however, please note that for the uniformly distributed datasets, as the SHadoop authors point out, the grid-file indexing is performing fine). Additionally, by examining the code of SHadoop, we discovered that it can execute a query using both MR and without MR; that is, by retrieving files directly from HDFS without using MR jobs. Thus, we compare the performance of our approach against both variants of SHadoop: SH with MR (SH-MR) and SH with HDFS without MR (SH-HDFS).

Similarly, we used the publicly available Simba code. However, creating an index for datasets bigger than 1 billion points was not possible; Simba repeatedly crashed while creating the index for the bigger datasets. For that reason, Simba is compared against our approach using only the 25GB and 35GB datasets, each of which contains 600 million and 1 billion data points respectively. Please note that this is enough to showcase the superiority of our approach against Simba: even for smaller datasets Simba is shown to be up to two orders of magnitude slower than CONI/COWI (even when the latter run over the bigger datasets).

B. Performance assessment

As shown in Figures 5 and 6 we compare the kNN query response time of COWI against SH-MR, SH-HDFS and

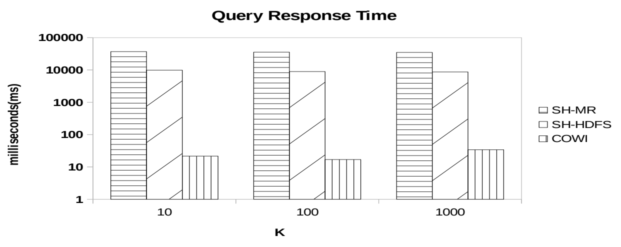


Figure 7: Dataset: 7.3 Billion data points (250GB)

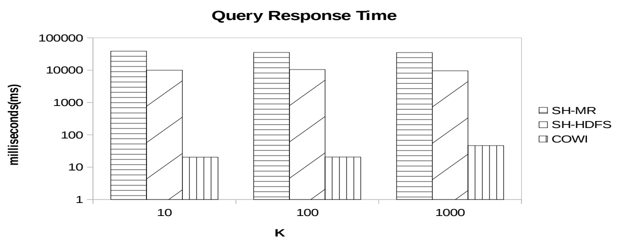


Figure 8: Dataset: 29.1 Billion data points (1TB)

Simba on the 20GB and 35GB datasets. We measured kNN query response time in milliseconds (ms) for different values of k. In our approach, query response time varies from 22ms-32ms. In SH-MR, the query response time is from 34,000ms – 35,000ms, in SH-HDFS 8,100ms – 8900ms, and in Simba 3,000ms – 5,000ms. The results concerning the performance of Simba and SH approaches are in line with those reported by the authors of Simba, with respect to the relative performance of SH and Simba. However, these results clearly indicate that COWI achieves query performance gains up to two orders of magnitude compared to Simba. We wish to stress that the query response time of Simba can only increase (to more than 5 seconds) when the dataset sizes increase (i.e., for the larger datasets that could not be indexed and thus could not be tested here). Also note that COWI achieves query performance gains of more than two orders of magnitude, compared against both implementations of SH.

The same conclusions hold for the 250GB and 1TB datasets, shown in Figure 7 and 8 respectively. Also, note that all approaches show excellent scalability; i.e., a very small increase in query response time occurs despite an increase of about 2 orders of magnitude in the dataset size.

To further stress-test our approach, quantifying also how the size of a cell affects query times, we increased the size of the dataset to 3.5 TB. With COWI, we store 10^6 data elements per cell, whereas using CONI we manage to reduce the row size to 2,000. As shown in Fig. 9, query processing time of COWI increases more than 10×(to between 526ms and 595.87ms), with CONI query processing time is within the range of 91.4ms - 185ms. Thus, CONI continues to offer gains of orders of magnitude, in addition to those benefits of CONI stemming from not requiring memory resources at the coordinator.

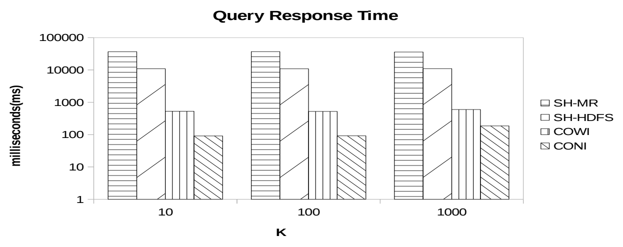


Figure 9: Dataset: 100 Billion data points (3.5TB)

Please note that the above experiments showcase that both of our key design choices pay off significantly. Namely, avoiding Hadoop-MR-based and Spark-based approaches and ensuring surgical access to only small but relevant data items (again unlike Simba, SH or Aqwa) can improve query response time considerably. The above experiments and comparisons actually quantify the relevant costs associated with each design choice.

In order to evaluate the scalability of our approach, we also measure the number of rows (cells) and total number of points accessed, on average, per query. This is fundamental for any coordinator-based approach, as the network bandwidth of the coordinator can be saturated and the same holds for the CPU processing data items retrieved from the data store.

As shown in Fig. 10, the average number of rows that are accessed by both of our approaches remains tamed with increased size datasets. As expected, it increases with larger k values. In our smallest dataset (20GB), when the value of $k = 10$ and $\alpha = 2000$, COWI accesses on average 1.17 rows (cells) per query. For the same values of k and α, when the dataset size increases to 250GB and 1TB, on average only 1.17 and 1.176 rows per query are accessed, respectively. Thus, COWI scales very well in terms of the average number of rows accessed per query with increasing dataset sizes. When the value of k increases to 1000 and $\alpha = 2000$, on average 2.8, 2.89 and 2.909 rows are accessed per query for 20GB, 250GB and 1TB dataset sizes, respectively.

Last but not least, when the dataset size is 3.5TB and $\alpha = 2000$, CONI accesses on average 2.16, 2.73, 3.29 rows for k equal to 10,100 and 1000, respectively. On the other hand, COWI for $\alpha = 100000$ accesses on average 1.01, 1.07 and 1.24 rows when k is 10, 100 and 1000, respectively. On average COWI accesses fewer rows than CONI in the 3.5TB dataset. This is because CONI has to access more rows of the meta-table in order to identify the closest cell, $\mathcal{C}^*$. With this result we start to see and quantify the tensions between CONI and COWI: CONI can reduce the value for α, but at the expense of needing to access additional meta-table rows from HBase. On the other hand, COWI needs no additional HBase accesses, as the results above show, but must use a much higher value for α.

Fig. 10 shows that in both COWI and CONI the average

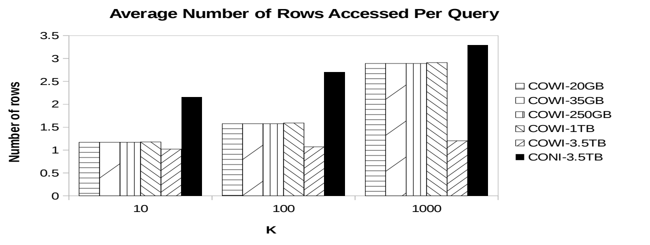


Figure 10: Average number of rows accessed per query.

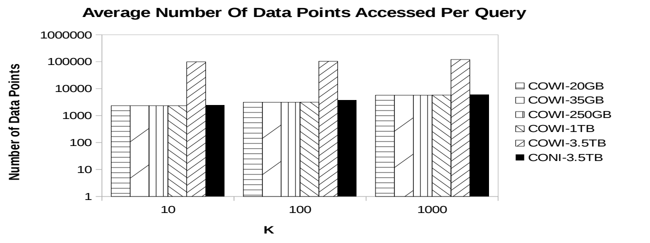


Figure 11: Average number of data points accessed per query.

number of rows accessed per query increases by a very small value when the dataset increases significantly. Similarly to the average number of rows accessed per query, the average number of points accessed per query is not significantly affected by the size of the underlying dataset in our solutions. As illustrated in Fig. 11, for k=10 and $\alpha = 2000$ COWI has accessed on overage 2,340, 2,340 and 2,339 points when the dataset size is 20GB, 250GB and 1TB, respectively. Simultaneously, CONI accessed on average 2,471 elements per query when $k = 10$ and $\alpha = 2,000$ when the dataset is 3.5TB. However, when using the largest 3.5TB dataset, COWI accessed 99,312 data elements for $k = 10$. This is because the parameter α must now be set to a much greater value (e.g., $\alpha = 100,000$) retrieving thus many more data points with every cell accessed. This demonstrates that when $\alpha >> k$, many irrelevant points (i.e., that do not contribute to the kNN list) are accessed as a result query response time is affected negatively; see Fig. 9. Fundamentally, the results in Fig. 11 show with COWI or CONI, on average, relatively the same number of points per query are accessed across widely varying dataset sizes.

Finally, it is also fundamental to scalability, in addition to the above two qualitative measures, to see how processing time at the coordinator (needed to process retrieved data points) is affected. Figure 12 shows the relevant results. It is again clear that dataset size increases have a very small effect on the time the coordinator must devote to data crunching. Note that, as expected, the processing time at the coordinator with COWI increases significantly with the largest dataset, as α assumes greater values, as this, in turn, leads to a very large number of points that must be (communicated to and) processed by the coordinator to produce the final query answer.

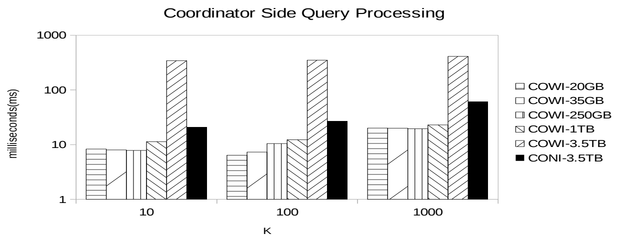


Figure 12: Coordinator Processing Time.

VIII. Conclusions

In this paper, we propose a novel approach to process kNN queries. This approach centres on two key features: First, it is based on a coordinator-based distributed query processing algorithm. This goes against state of the art approaches, which are based on scalable data-parallel processing engines, such as Hadoop/MR and Spark. The key point put forward here is that scalability should not come at the expense of query processing efficiency; using Hadoop/MR or Spark based solutions may achieve scalability but unnecessarily sacrifice efficiency. We have shown that computing a kNN query should and could be a matter of a few tens of milliseconds and not several (tens of) seconds. Second, we have paid attention to the data organization, storage, and indexing in a way that allows surgical accesses to only relevant data points. Why should an algorithm retrieve from storage, communicate, and process millions of other data items, when processing a 10NN query? We have investigated the relations of the cell size with key scalability factors, such as the size of available memory at the coordinator. We have provided two versions of our approach, COWI and CONI, while respecting the need of maintaining small cell sizes, depending on available memory on the coordinator and dataset sizes. We have conducted performance evaluations of COWI/CONI and compared against the state of the art (Hadoop-based) SH and (Spark-based) Simba solutions. The results showcased and quantified performance improvements of two to three orders of magnitude. We also studied all fundamental factors affecting the the scalability of our proposed approach, showing that the overall query processing times scale excellently with dataset sizes: We studied measures, such as the number of cells and data points retrieved, communicated, and processed, as they depend on dataset sizes, as well as the processing times at the coordinator. All those results further substantiate the scalability of our approach.

References

[1] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop GIS: A high performance spatial data warehousing system over MapReduce. *PVLDB*, 6(11):1009–1020, 2013.

[2] A. M. Aly, A. S. Abdelhamid, A. R. Mahmood, W. G. Aref, M. S. Hassan, H. Elmeleegy, and M. Ouzzani. A demonstration of aqwa: Adaptive query-workload-aware partitioning of

big spatial data. *Proc. VLDB Endow.*, 8(12):1968–1971, Aug. 2015.

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

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[5] J. L. Bentley, D. F. Stanat, and E. H. Williams. The complexity of finding fixed-radius near neighbors. *Information processing letters*, 6(6):209–212, 1977.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[7] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[8] A. Eldawy and M. F. Mokbel. A demonstration of spatialhadoop: An efficient mapreduce framework for spatial data. *Proc. VLDB Endow.*, 6(12):1230–1233, Aug. 2013.

[9] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[10] L. George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[12] D. Han and E. Stroulia. Hgrid: A data model for large geospatial data sets in hbase. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 910–917, Washington, DC, USA, 2013. IEEE Computer Society.

[13] Y.-T. Hsu, Y.-C. Pan, L.-Y. Wei, W.-C. Peng, and W.-C. Lee. Key formulation schemes for spatial index in cloud data managements. In *Proceedings of the 2012 IEEE 13th International Conference on Mobile Data Management (Mdm 2012)*, MDM '12, pages 21–26, Washington, DC, USA, 2012. IEEE Computer Society.

[14] L. Jiang, B. Li, and M. Song. The optimization of hdfs based on small files. In *Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on*, pages 912–915. IEEE, 2010.

[15] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(34):1–7, 2016.

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

[17] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi. Mdhbase: Design and implementation of an elastic data infrastructure for cloud-scale location services. *Distrib. Parallel Databases*, 31(2):289–319, June 2013.

[18] Y. Park, M. Cafarella, and B. Mozafari. Neighbor-sensitive hashing. *Proceedings of the VLDB Endowment*, 9(3):144–155, 2015.

[19] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. *SIGMOD Rec.*, 24(2):71–79, May 1995.

[20] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[21] D. Vohra. Apache hbase primer, 2016.

[22] J. Wang, W. Liu, S. Kumar, and S.-F. Chang. Learning to hash for indexing big dataa survey. *Proceedings of the IEEE*, 104(1):34–57, 2016.

[23] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1071–1085, New York, NY, USA, 2016. ACM.

[24] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[25] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on*, pages 34–41. IEEE, 2015.

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

[27] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[28] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[29] S. Zhang, L. Miao, D. Zhang, and Y. Wang. A strategy to deal with mass small files in hdfs. In *Proceedings of the 2014 Sixth International Conference on Intelligent Human-Machine Systems and Cybernetics - Volume 01*, IHMSC '14, pages 331–334, Washington, DC, USA, 2014. IEEE Computer Society.

[30] Y. Zhang and D. Liu. Improving the efficiency of storing for small files in hdfs. In *Proceedings of the 2012 International Conference on Computer Science and Service System*, CSSS '12, pages 2239–2242, Washington, DC, USA, 2012. IEEE Computer Society.