Distrib Parallel Databases (2017) 35:55–81
DOI 10.1007/s10619-017-7194-0

M-Grid: a distributed framework for multidimensional indexing and querying of location based data

Shashank Kumar[1] · **Sanjay Madria**[1] · **Mark Linderman**[2]

Published online: 13 March 2017

Abstract The widespread use of mobile devices and the real time availability of user-location information is facilitating the development of new personalized, location-based applications and services (LBSs). Such applications require multi-attribute query processing, scalability for supporting millions of users, real-time querying capability and analyzing large volumes of data. Cloud computing aided a new generation of distributed databases commonly known as key-value stores. Key-value stores were designed to extract values from very large volumes of data while being highly available, fault-tolerant and scalable, hence providing much needed infrastructure to support LBSs. However, complex queries over multidimensional data cannot be processed efficiently as they do not provide means to access multiple attributes. In this paper, we present M-Grid, a unifying indexing and a data distribution framework which enables key-value stores to support multidimensional queries. We organize a set of nodes in a modified P-Grid overlay network which provides efficient data distribution, fault-tolerance and query processing over multidimensional data. To index, we use Hilbert Space Filling Curve based linearization technique which preserves the data locality to efficiently manage multidimensional data in a key-value store. We propose algorithms to dynamically process range and k nearest neighbor (kNN) queries on linearized values. This removes the overhead of maintaining a separate index table.

✉ Sanjay Madria
madrias@mst.edu

Shashank Kumar
sk2z6@mst.edu

Mark Linderman
mark.linderman@us.af.mil

[1] Department of Computer Science, Missouri University of Science and Technology, Rolla, USA

[2] Information Division, Air Force Research Lab, Rome, NY, USA

Our approach is completely independent from the underlying storage layer and can be implemented on any cloud infrastructure. Our experiments on Amazon EC2 show that M-Grid achieves a performance improvement of three orders of magnitude in comparison to MapReduce and four times to that of MD-HBase scheme.

Keywords Location based services · Multidimensional indexing · Peer-to-peer system

1 Introduction

According to the latest report published by International Telecommunication Union [1], the number of mobile subscribers is equal to the world's population at the end of the year 2015 [2]. The increasing need for mobility accompanied with recent advances in wireless and mobile technology have created one of the most promising value added services which are known as location based services (LBSs). LBSs utilizes the ability of mobile devices to provide a user's current geographical location through a mobile network. They allow mobile users to search their environment and give them instant access to personalized and localized content. Currently, a wide variety of LBSs are available to mobile users which include route mapping applications, interactive city guides, location-aware marketing, object tracking and monitoring, and finding objects based upon proximity [3]. LBSs imposes a broad range of requirements on the underlying supporting platform which includes modeling and representing multidimensional data, handling high velocity updates of millions of users and providing real-time analysis on large volumes of data. Apart from these core requirements, LBSs should also provide capability to efficiently process various types of queries to support wide range of applications.

Legacy relational database management systems (RDBMS) can provide efficient and fast complex query processing on multidimensional data by leveraging the built-in indexing features and a rich query processing language. These indexing features are provided by creating an additional index layer on top of a relational store. Examples of such specialized spatial databases include, Oracle Spatial [4] which supports R-tree [5] and Quad-tree[6] based indexing and IBM DB2 spatial extender [7] which provides a three-tiered spatial grid index. However, these centralized systems create a single point of failure, are costly to implement and suffer from scalability bottleneck as the volume of data and the number of users grow.

Cloud-based distributed key-value stores emerged as a new paradigm to provide elastic data management services which can scale according to the demand of different applications. Current key-value stores such as BigTable [8], HBase [9] and Cassandra [10] are designed to be highly available, fault-tolerant and can support millions of users by sustaining high update throughput. However, these key-value stores can only process efficient exact match queries on a single attribute as they lack built-in indexing mechanism to access multiple attributes. For querying over multiple attributes, two design choices are available. The first choice is to create separate indexes, pertaining to each attribute. However, creating multiple index tables will incur huge additional load on the system in terms of managing large volumes of data. Moreover, processing results

from multiple tables will require moving the data from other nodes and performing in-memory aggregation to find the intersection. The second choice is to use MapReduce [11] style parallel processing to scan the entire dataset. However, LBSs require real-time query processing and thus, parallel scanning of the entire dataset is not useful especially for queries with smaller selectivity.

In this paper, we propose M-Grid, a novel data distribution and multidimensional indexing framework to support LBSs on cloud platforms. Because of the characteristics of key-value stores which includes availability, horizontal scalability and a distributed architecture, it became a natural choice to use them as M-Grid's storage back-end. However, the key challenges in developing such an index framework on top of a key-value store are, efficient modeling of multidimensional data and providing it with the ability to process complex multidimensional queries efficiently. M-Grid solves the former by using Hilbert Curve [12] based linearization technique and later by integrating it with a modified P-Grid [13] overlay network. The modification makes the M-Grid a static network by precomputing and assigning the subspaces to the nodes in the network. Hilbert Curve maps multidimensional attributes onto single dimensional while preserving its data locality. On the other hand, P-Grid arranges the nodes in a virtual binary and partitions the multidimensional search space into subspaces. The trie structure acts as the static routing index layer on top of the underlying key-value store and serves as the entry point for multidimensional search. Thus, one of our main contributions is also building this P-Grid overlay network such that data distribution and access path is no more random unlike original P-Grid. M-Grid then processes complex queries by distributing them across the cluster according to P-Grid's prefix-based routing mechanism. One of the practical examples where M-Grid can be used effectively is in Earthquake Disaster Management. In such a scenario, M-Grid can provide scalable knowledge management services to help and locate missing/trapped people by integrating various sensor data and their location coordinates for situational awareness in real-time.

In summary, this paper makes the following novel contributions:

(i) We propose a new data distribution and multidimensional indexing framework, M-Grid, which can efficiently index and process point, range and kNN queries. M-Grid integrates P-Grid overlay network for data distribution and a range partitioned key-value store.
(ii) We leverage Hilbert Space Filling Curve based linearization technique to convert multidimensional data to a single dimension while preserving its data locality better than some others.
(iii) We propose algorithms which can dynamically process point, range and k-nearest neighbor (kNN) queries on linearized values using a modified P-Grid's prefix-based routing mechanism. This removes the overhead of creating and maintaining a separate index table.
(iv) We have performed experiments to measure insert throughput (update follows the same insert algorithm considering it as an append), along with point, range and (kNN) experimental evaluations on Amazon EC2 to show the effectiveness of our framework over MD-Hbase, a state of the art scheme [14].

The rest of the paper is organized as follows: Section 2 presents the state of the art currently in the area of multidimensional data management while Sect. 3 introduces the key background concepts in the area of Hilbert's space-filling curves. Section 4 presents an overview of M-Grid indexing framework. Section 5 describes the design and implementation of M-Grid's storage layer. Section 6 presents the proposed point, range and kNN querying algorithms. Section 7 shows the performance evaluation and we finally conclude the paper in Sect. 8.

2 Related work

Query processing on large data volume has been the center of research since the evolution of cloud computing. This field is predominantly dominated by two classes of scalable data processing systems. The first class uses an underlying key-value store to manage structured data, for example, Google's Bigtable [8], Apache HBase [9], Apache Cassandra [10], Amazon's Dynamo [15] and Yahoo's PNUTS [16]. These systems while being fault-tolerant, highly scalable and available, can efficiently process simple keyword based queries. However, these systems do not provide multi-attribute access as they lack additional secondary indexing capabilities. The second class uses a distributed storage system such as Google's GFS [17] and Apache's HDFS [18] to manage unstructured data. Both of these systems depend on scanning the entire dataset using parallel processing approaches (for e.g. MapReduce [11]) in order to process complex queries such as range and kNN on multidimensional data which incurs high query latency.

To efficiently process multidimensional data, the authors in [19] present a general indexing framework for cloud systems. In their indexing framework, processing nodes are arranged in a BATON overlay network and each node builds a local B^+-tree or hash index on its data. To speed up query processing and data access, a portion of local index is selected and published in the overlay network which forms its global index. Based upon the similar two level index architecture, three more indexing schemes are proposed. The authors in [20] propose RTCAN, which builds a global index by publishing selective local R-tree indexes on C^2 overlay network. EMINC [21] is an indexing framework in which individual slave nodes build a KD-tree [22] on its local data and a global R-tree index is build on a master node. QT-Chord [23] is an indexing framework which builds IMX-CIF Quad-tree over local data and distributes the hashed codes to the Chord overlay network. Apart from these indexing schemes, the work in [24] proposes an in-memory indexing framework PASTIS which uses compressed bitmaps to construct partial temporal indexes. Lastly, the authors in [25] propose VBI indexing framework for P2P systems in which peers are organized in a balanced tree structure to support multidimensional point and range queries. All the aforementioned schemes provide efficient algorithms to process queries. However, such solutions either lack stability in terms of handling data size as the local and global indexes have to be stored in main memory, or are expensive to implement.

The above drawbacks were addressed in [14], wherein the authors proposed the MD-HBase indexing scheme. MD-HBase is a highly scalable data store which first converts a multidimensional point to a single dimension binary key by using Z-Order

curve based linearization technique. Then, to partition the linearized space, keys are grouped together according to their longest common Z-value prefix to simulate a KD-trie or Quad-trie. For efficient subspace pruning a single primary index layer is built over a range partitioned key-value store which stores the boundary information of each subspace. Experimental evaluation on a dataset containing four hundred million points demonstrates the scalability and efficiency of their index structure. However, the proposed scheme has two fundamental drawbacks. First, Z-Order curve loosely preserves the locality of data points which increases the range and *k*NN query latency. Secondly, their scheme's throughput performance is also capped by maintaining a separate index table which needs to be updated with each data point insert.

In M-Grid, we combine the best of both the systems by arranging the nodes in an overlay network and using a range partitioned key-value store to manage data without the overhead of maintaining a separate index table. This allows it to scale linearly as the data size grows while sustaining high insert and update rates. Furthermore, M-Grid can also efficiently process point, range and *k*NN queries on secondary attributes which is a key requirement for LBSs.

3 Background

3.1 Linearization using space-filling curve

Linearization is a dimensional reduction method which maps multidimensional attributes onto single dimensional space. Space-filling curve is a linearization technique in which a continuous curve is constructed visiting every point in a n-dimensional hypercube without overlapping itself. The benefits of using them is that, after mapping, neighboring points in n-dimensional space remains close in one dimensional space also. Therefore, space-filling curves are widely used in applications like image processing [26], scientific computing [27] and geographic information systems [28] which require sequential access to datasets. In M-Grid, we use Hilbert space-filling curve [12] to index multidimensional points in the underlying uni-dimensional key-value store to provide query efficiency.

The Hilbert Curve is a continuous space-filling curve which induces a sequential ordering on multidimensional data points. Formally, Hilbert Curve is a one-to-one function:

$$\mathbb{H} : [0, 2^{mn} - 1] \rightarrow [0, 2^{m} - 1]n$$

where n is the number of dimensions in a 2^mx2^m space and n≥2, m>1. This function determines the Hilbert value (H-value) of each point in the original coordinate space where H-value ∈ [0, 2^{mn}-1]. Figure 1 illustrates the coordinates in a 2-dimensional space and its equivalent Hilbert Curve of first order. A curve of order i>1 is constructed in a recursive manner where each vertex of the first order curve is replaced by the curve of order i-1, after rotating and/or reflecting it to fit the new curve [29]. This recursive construction process can also be expressed as a tree structure (Fig. 3) to show the correspondence between the coordinate points (n-points) and their H-values

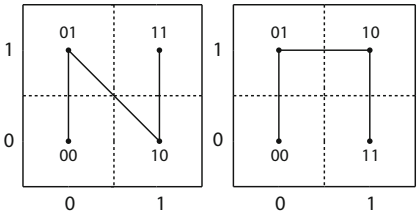


Fig. 1 A 2-d space and its equivalent first order Hilbert curve

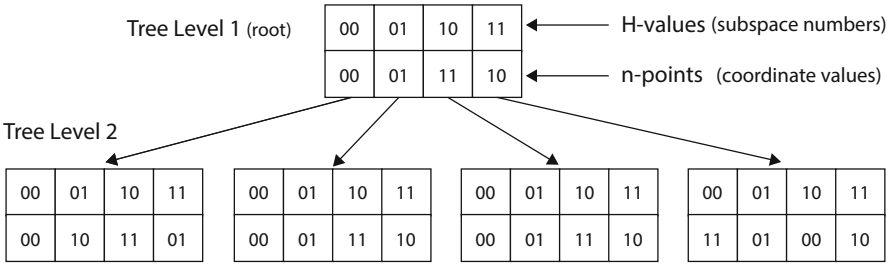


Fig. 2 A tree representation of the second order Hilbert curve in 2 dimension

(subspace numbers) in binary notation [30]. The depth of the tree is equal to the order of the curve and the root node corresponds to the first order curve of Fig. 1. Also, a collection of nodes at any tree level, i, describes a curve of order i. We demonstrate the calculation of H-value using the tree structure shown in Fig. 2 with the example of point (10, 11). In the first step, we concatenate the top bits of the coordinates of point (10, 11) to form the n-point 11. At root node, this n-point corresponds to the subspace 10. In the next step, we descend down one level of the tree to level 2, following the subspace 10 at root. We concatenate the next two bits of the coordinate of the point to form the n-point 01. At tree level 2, the n-points corresponds to the subspace 01. As there are no more levels to descend, the calculation of H-value ends. The final H-value of the point (10, 11) is then formed by concatenating the values of the subspace which we found in each step starting from root node, i.e. 1001. Generating H-value of a point using a tree structure requires the cardinality of each attribute to be equal. However, in LBSs, the cardinality of the attributes can be unequal. Hence, in M-Grid, we compute the H-value using the algorithm presented in [31] which uses logical operations to efficiently compute direct and inverse mapping of a point having unequal attributes on the Hilbert Curve. Currently, this algorithm uses total number of bits (sum of bits in each dimension) less than or equal to 64 bits. If we want to index also attributes timestamp and user id, precision (number of bits) on longitude and latitude can be decreased to accommodate for these other values or possibly increase the total size of bits used or compress the key values.

Beside Hilbert Curve, several space-filling curves such as Z-Order Curve [32] and Gray Order Curve [33] are proposed. Figure 3 shows the illustration of second order Hilbert, Z-Order and Gray Curves for 2 dimensional space. We chose Hilbert Curve to index multidimensional points in M-Grid as it has superior clustering and strong locality preserving properties as compared to other space-filling curves [34–36]. These

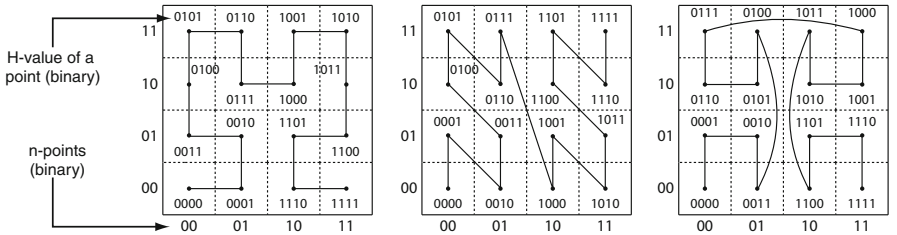


Fig. 3 Illustration of second order Hilbert, Z-order and Grey-order curve for 2 dimensions

properties help M-Grid to achieve efficient clustering of the location points in the database resulting in low query latency.

3.2 Overlay networks

Peer-to-Peer (P2P) overlay networks offer a new paradigm for providing scalability, fault-tolerance and robustness to distributed systems. In P2P networks, all nodes are considered as equal and have symmetrical roles. Each node can either act as a client or a server. The nodes can join or leave the network independently and they share their resources with other participating nodes. P2P networks are suitable for large scale distributed applications due to their cooperative nature and flexible network architecture.

Based upon the search mechanisms used to identify indexed data, P2P networks can be classified as either unstructured and structured. Unstructured P2P networks such as Freenet [37] distribute the data randomly on nodes and uses either a centralized index server or flooding mechanisms for searching. Such searching mechanisms incurs high query latency and therefore, are not suited for large scale data oriented applications such as LBSs. Structured P2P networks such as CHORD [38], BATON [39], CAN [40], PASTRY [41], P-Grid [13] and P-ring [42] use a distributed and scalable access structure to efficiently distribute and search data items. Chord and Pastry only support exact match queries. CAN supports multidimensional queries but it has a high routing cost for low dimensional data. BATON, P-Grid and P-ring supports one dimensional range queries. However, except P-Grid, none of the other P2P networks have a truly decentralized architecture. Also, P-Grid supports prefix-based routing which is integral to our querying algorithms.

3.3 Prefix-Grid (P-Grid) overlay network

P-Grid is a scalable, self-organized structured P2P overlay network based on a distributed hash table (DHT). Its access structure is based upon a virtual distributed binary-trie. The canonical-trie structure is used to implement prefix-based routing strategy for exact match and range queries. P-Grid assigns each node n a binary bit string which represents its position in the overall trie and is called $path(n)$ of the node. This path contains the sequence from leaf to the root. An illustration of P-Grid trie can be seen in Fig. 4. To store a data item, P-Grid uses a locality preserving hash function to convert the data item's identifier to a binary key κ, where $\kappa \in [0, 1[$. The

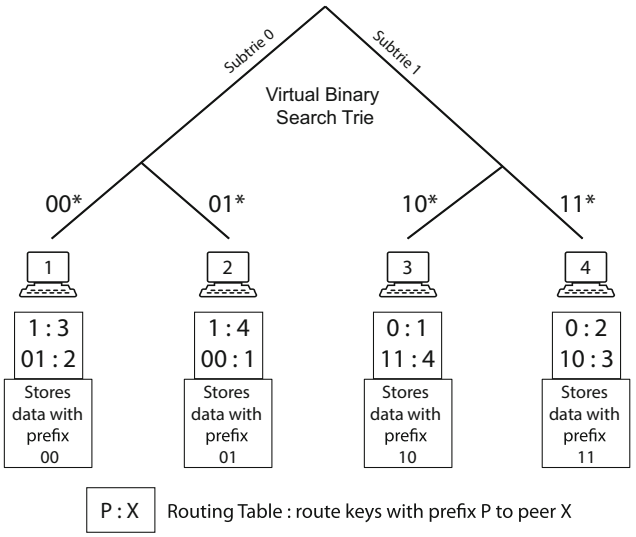


Fig. 4 An example P-Grid trie

data item is then routed to the node whose *path* has the longest common prefix with κ. For example, the *path* of node 2 in Fig. 4 is 10, therefore it stores all the data items whose keys begin with 10.

P-Grid employs a completely decentralized and parallel construction algorithm which can build the overlay network with short latency. The construction process is strictly based on local peer interactions which is done by initiating random walks on pre-existing unstructured overlay network. Each node in P-Grid maintains a routing table which stores the information about the paths of other nodes in the network. Specifically, for each bit position, it maintains the address of at least one node that has a path with the opposite bit at that position. This information is stored in the routing table in the form of $[path(n), \text{FQDN}(n)]$ where $\text{FQDN}(n)$ is the fully qualified domain name of the node. Details of the construction algorithm can be found in [43].

3.3.1 Searching in P-Grid

P-Grid utilizes a simple but efficient strategy to process exact match and range queries [44]. For executing an exact match query, the query is mapped to a key and routed to the responsible node whose path is in a prefix relationship with the key. For example, in Fig. 4, a query for 1111 is issued to node 2 which is responsible for storing the keys starting with 01. As node 2 cannot satisfy the query request, it searches its routing table and forwards the query to node 4, which has the longest common prefix of 1 with the query. Node 4, upon getting the request, searches its local storage to find the data item associated with the key 1111. If the key exists, node 4 sends an acknowledgement message to node 6 which can then request the data. The complexity of the exact match process is $O(\log \Pi)$, where Π is the number of messages exchanged and is independent of how the P-Grid is structured.

P-Grid processes a range query in a parallel and concurrent manner. The intuition behind the query processing strategy is to divide the P-Grid trie in subtries and selectively forward the query to only those nodes of the subtries whose paths intersects with the query. For example, in Fig. 4, node 1 issues a range query, having 1000 as the lower bound and 1101 as the upper bound. Node 1 splits the P-Grid trie in 2 subtries i.e. 01 and 1. Node 1 forwards the query for subtrie 1 to node 3. The subtrie 01 of node 2 does not intersect with the query and therefore is ignored. Node 3, after getting the request, repeats the same process and forwards the query to node 4. The search cost of the range query process is independent of the size of range of the query but depends on the number of data items in the result set.

4 M-Grid data distribution and indexing framework

The M-Grid indexing framework constitutes a federation of shared-nothing cluster of nodes leased from the cloud. Our primary goal in designing M-Grid is to support LBSs by having a truly decentralized P2P architecture which can be scaled accordingly. M-Grid achieves this by adopting a simple two tiered architecture. The upper tier is based on the P-Grid's overlay network which is responsible for routing queries and assigning sub-spaces to the computing nodes, whereas, the lower tier utilizes the underlying key-value store (HBase) to maintain data, depending on the type of data model being used (Sect. 5.2). A node in M-Grid serves two purposes. It is a node in P-Grid because it maintains subspace information about all other nodes in its routing table. The same node is a part of HBase cluster which store the actual data. A HBase cluster comprises of HBase Master and several Region Servers/Data Nodes.

Our architecture splits the query processing in two phases. In the first phase, the node responsible for storing the subspace is identified by searching the routing table. The routing table holds the references of all the other nodes which are at an exponential distance from its own position in the search space. This is achieved by arranging the node in a virtual binary-trie structure. In the second phase, the query is forwarded to the responsible nodes which processes it locally. Although P-Grid efficiently divides the search space in a self-organizing manner, the cost associated with its maintenance protocol is very high. P-Grid dynamically assigns new subspaces to the nodes by extending their paths for distributing load in the network. This operation is very costly for LBSs as they manage large volumes of data, and, dynamically changing the assignment will lead to moving of data from one node to another. Furthermore, P-Grid is a probabilistic data structure which uses best-effort strategy for processing queries. Thus, after issuing a query, it is not possible for a node to calculate the exact number for response messages it has to expect for getting the complete result. However, M-Grid processes kNN queries by iteratively performing range searches and with each iteration, the system has to wait until it receives all the results for further processing which is not viable in P-Grid. M-Grid solves these problems by making the following changes in the original architecture of P-Grid:

(i) It creates a balanced network by associating only one node with each leaf of the virtual trie. This assigns each node to a unique subspace.

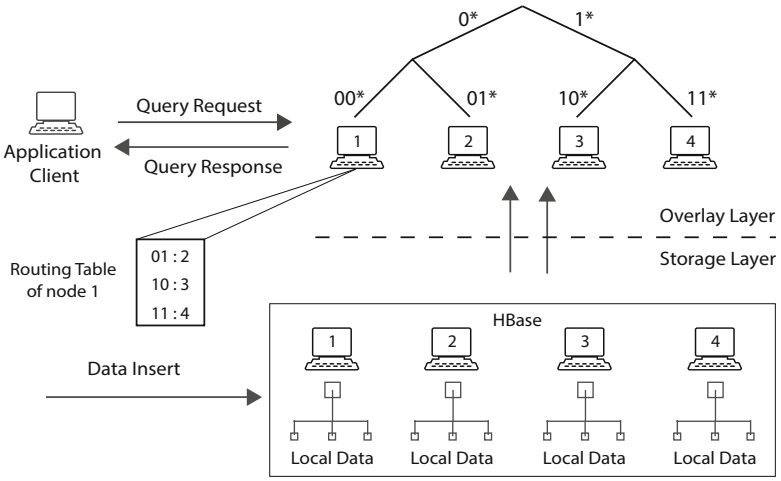


Fig. 5 M-Grid's system architecture

(ii) It provides the ability to start a P-Grid network from a predefined prefix to handle data skewness.
(iii) It modifies the maintenance protocol so that, after network stabilization, nodes do not extend their paths.
(iv) For efficient query processing, each node stores the information about all the other nodes in the network. Consequently, the cost of routing queries in terms of messages is reduced from $O(\log\Pi)$ to 3 in the worst case scenario.

The resultant high-level overview of our architecture is shown in Fig. 5. We construct M-Grid using the bottom-up approach in which, nodes are first arranged in an HBase cluster and then join the overlay network. The construction is done in an off-line procedure and has a small one time set-up cost. Data insertion can be done at any node. To insert the data, we first calculate the H-value of the multidimensional point and insert it according to the data models presented in Sect. 5.2.

5 Data storage layer

M-Grid is a storage platform independent framework which allows us to use any key-value store as per the need of the application. We use Apache HBase [9] to store the H-value of a multidimensional point which we use as the unique *rowkey*. In this section, we describe the overview of HBase and the two data models, *Table per Node* and *Table Share*, which we used to store data in M-Grid.

5.1 Apache HBase

Apache HBase is a distributed, non-relational key-value datastore modeled after Google's BigTable [8] and built on top of HDFS [18]. It is designed to provide

high scalability, partition tolerance and row-level consistency which makes it suitable for big data applications such as LBSs. A table in HBase is composed of multiple rows and columns. Each row is identified by a unique primary *rowkey*. The columns are grouped into column families where each column family is identified by a pair of user defined *prefix:qualifier*. The column *prefix* is static and needs to be defined while creating the table whereas the *qualifiers* can be added dynamically while inserting the data. Thus, we need to specify at least two attributes in order to get a value from a table which are the *rowkey* and the column family identifier.

The physical architecture of HBase consists of a master server and a collection of slaves called region servers. Each region server contains multiple regions and each region stores a sorted continuous range of *rowkeys* which belong to a table. HBase provides autosharding, which means that when the size of a region exceeds a predefined threshold, it dynamically splits the region into two sub regions. This allows HBase to achieve horizontal scalability as the volume of data grows. Despite having a Master/Slave architecture, the role of a Master server is limited to handle administrative operations like monitoring the cluster, assigning regions to regional servers and creating, modifying or deleting a table. The read and write operations are provided directly from the region servers even if the Master server fails.

5.2 Storage models

5.2.1 Table per node (TPN) model

In this model, each node is responsible for maintaining their own separate table. When a node joins M-Grid, it creates a table in HBase by the name of its own Fully Qualified Domain Name (FQDN). The nodes stores the *rowkeys* locally according to the subspace they are responsible for. For example, in Fig. 4, node 3 which has a path '10' will store all the *rowkeys* which has a '10' prefix. This model efficiently maps the key space to the responsible node allowing parallel and independent query operations. As the *rowkeys* are stored locally, this model provides low access latency. However, the insert operation is expensive since the prefix of a *rowkey* needs to be checked for finding the responsible node before its insertion.

5.2.2 Table share (TS) model

In this model, all the nodes share a single table to manage data *rowkeys*. This model allows us to efficiently insert keys directly in the table without checking their prefixes. Thus, this model can sustain high insert throughput. However, as each table is distributed across the server, this model has high access latency. An important observation to note here is that when we employ TPN, the overlay layer is used for both data partitioning and routing the queries where as in the case of TS model, the overlay layer is used just for the purpose of routing queries.

6 Query processing

6.1 Data insert and point query

Data insert and point query can be executed by using the P-Grid's search mechanism to forward the insert or query request to the responsible node but it involves additional routing cost. Our algorithm (Algorithms 1 and 2) efficiently inserts the data and process the point query respectively, by leveraging the key-value store's ability to provide direct data access. We modify the data insert and point query algorithm with respect to two storage model described in section (V-B). In Algorithm 1, to insert a point, we first compute the binary H-value(*rowkey*) of the point (line 1). Next, for Table per Node model, insert operation is split into two phases. In the first phase, we search the routing table ρ, to find the name (FQDN) of the node whose path has the longest common prefix with the *rowkey* (line 2). This model stores the data in a table whose name is set to the name of the node, hence this step is sufficient to find the name of the table responsible for storing the point. In the second phase, we insert *rowkey* by the standard insert operation on that table (line 3). For Table Share model, we can easily insert *rowkey* in the predefined shared table (line 5). The steps for inserting a point p are shown below:

Algorithm 1 Data Insert (point p, value v)

1: $rowkey \leftarrow computeH\text{-}value(p)$
// Table per Node //
2: $n.Table$ = PrefixMatchingBinarySearch(ρ, $rowkey$)
3: $n.Table$.insert($rowkey$, value v)
4: **return** true
// Table Share //
5: $sharedTable$.insert($rowkey$, value v)
6: **return** true

Given a d-dimensional point $p = (p_1, ..., p_d)$, our point query strategy tries to identify the value v associated with p. To process the query, we first compute the H-value of the point to calculate the *rowkey* (line 1). Next, similar to our insert algorithm, for Table per Node model, the query processing is split into two phases. In the first phase, we search the routing table ρ, to find the name (FQDN) of the node whose path has the longest common prefix with the *rowkey* (line 2). In the second phase, we retrieve v by the key-lookup operation on that table. For Table Share model, we can easily retrieve v by simple key-lookup operation on shared table.

6.2 Range query processing

A range query is a type of query that retrieves all records where some attribute is between an upper and lower boundary. Formally, a range query is a hyper-rectangular region formed by lower and upper bound coordinates, $(l_1, l_2, .., l_n)$ and $(u_1, u_2, .., u_n)$ with $min_i \leq l_i \leq u_i \leq max_i$. Another common query in LBSs is "Get all values

Algorithm 2 Point Query Processing(p)

Input query point p
Output value v associated with p
1: $rowkey \leftarrow computeH\text{-}value(p)$
// Table per Node //
2: $n.Table$ = PrefixMatchingBinarySearch($\rho, rowkey$)
3: **return** ($v \leftarrow lookup(rowkey, n.Table)$)
// Table Share //
4: **return** ($v \leftarrow lookup(rowkey, sharedTable)$)

around (x longitude, y latitude) within z radius. This type of query can also be easily translated to a range query after calculating the enclosing square as the bounding box. P-Grid's trie-based partitioning divides the linearized space into equal size subspaces and assigns subspaces to the nodes according to their paths. The range query region intersects with one or more subspaces. A naïve range query strategy will try to retrieve all the points contained in the query region by searching between the subspaces which the query lower and upper bound intersects. This querying strategy works with other space-filling curves such as Z-order which loosely preserves the data locality but not in Hilbert Curve as in each curve, the orientation of subspaces is different (Fig. 3). For example, consider the range query Q1 as shown in Fig. 6. Its lower bound and upper bound coordinates are A (01,01) and F (11,10). The equivalent H-value range of this query is $\langle 0010, 1011 \rangle$. A level two binary-trie partitions the space into equal size four quadrants namely 00, 01, 10 and 11. The first subspace to be searched is determined by the H-value of the lower bound which is 00. All the subsequent subspaces which lie between the lower and upper subspaces needs to be searched in order to get the points which are contained in the range query. In this example, the naïve querying strategy will search the 00, 01 and 10 subspaces. The subspace 11 though intersects with the query will be skipped.

The authors in [45] present "best effort query processing techniques" using Z-curve and CAN protocol which are not suitable for LBSs. Our range query algorithm (Algorithm 3) is based upon the method described in [46,47]. The intuition behind the algorithm is to find the boundaries of only those subspaces which the query region intersects. Thus, the original query range is divided into many smaller sub-ranges. Our algorithm divides the range query processing in two phases as described below:

(i) In the first phase, we divide the original range query into smaller sub-queries, one for each subspace which the query region intersects (line 5). We perform this by calculating the lowest H-value of the point in each subspace lying within the query region. We call that point as the next-match and the function which calculates it as the calculate-next-match().
(ii) In the second phase, we process each sub-query according to P-Grid's search mechanism which forwards the sub-query to all the nodes whose path intersects with the upper and the lower bound of the sub-query.

Subspaces can be viewed as logically ordered by the lowest H-value of a point in a subspace and we call it as the subspace-key. For example, the subspace-key of subspace 11 in Fig. 6 is 1100. In general terms, a subspace-key is also the point where the Hilbert Curve enters in a subspace. Subspaces which intersect with the

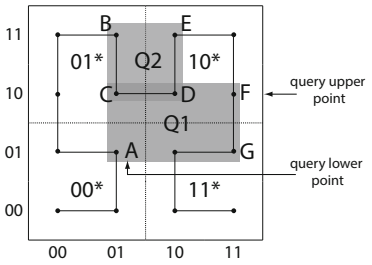


Fig. 6 Example of a range query on points mapped to the second order Hilbert curve in 2 dimensions

query region are iteratively identified in ascending subspace-key order by calculate-next-match() function. In the first iteration, the calculate-next-match() tries to identify the lowest H-value of any point lying within the query region. The first subspace in which the next-match lies is identified by giving the value of 0 as the input. In the second iteration, the calculate-next-match() tries to find the lowest H-value of a point which is equal or minimally greater than the subspace-key of the successor to the subspace searched in the previous iteration. The process is effected by a variable current-subspace-key which stores the current value of subspace-key in each iteration. For finding the intersecting subspaces, calculate-next-match() iteratively performs a binary search on the node which will be explained later. To illustrate the operation of calculating sub-ranges using calculate-next-match() function, consider an example range query Q2 as shown in Fig. 6.

(i) The range query Q2, is defined by providing the lower and upper bound coordinates C (01,10) and E (10,11) respectively. The H-value equivalent of this range query is ⟨0111, 1001⟩.
(ii) The current-subspace-key is initially set to the subspace-key of subspace 00, i.e. to 0000.
(iii) The calculate-next-match() function is called and it determines that the H-value of point C is the first next-match to the query, i.e. 0111.
(iv) The current-subspace-key is set to the subspace-key of the successor subspace, i.e., subspace 10. Its subspace-key is the H-value of point D, i.e. 1000.
(v) The calculate-next-match() is called and it determines that the next-match to the current-subspace-key to be the H-value of point D, i.e. the current-subspace-key is its own next-match.
(vi) The current-subspace-key is set to the subspace-key of the successor subspace, i.e., subspace 11. Its subspace-key is the H-value of point G, i.e. 1100.
(vii) The calculate-next-match() is called and it determines that there is no higher next-match to the current-subspace-key. The query process therefore terminates.

To find the next-match, we determine the lowest subspace which intersects with the current query region by using the binary search algorithm. This algorithm iteratively determines the lowest subspace which intersects with the current query region at any node of the tree (where a node of a tree represents a collection of sub-spaces ordered by their H-values). In each iteration, we will discard half of the subspaces and descends down the correct branch of the tree until we find the next-match at the leaf level. Also, this descent is an iterative process where with each iteration, we restrict

the user defined search space with the bounds of the subspace being searched. The new bounds are collectively called at current-query-region which is initially set as the original query region. We start with computing the lower and upper n-points by concatenating the bits at position k (k is the level of tree) of the lower and upper bounds of the current query region. Once we have these n-points, we determine whether the query regions intersects with the lower half or (and) upper half of the subspaces. To do so we use a function, h_to_c(). Solving this function using the H-values of a subspace will give us its n-points. If the H-values of a sub-set of subspaces are in the following range:

$$[\text{lowest}, ..., \text{max-lower}, \quad \text{min-higher}, ..., \text{highest}]$$

then all subspaces whose H-values are in the lower sub-range [lowest,...,max-lower] have same value (either 0 or 1), for their coordinates in one specific dimension, i. Whereas subspaces having their H-values in the higher sub-range have the opposite value in the same dimension, i. To find the value of i, we compute a n-point variable called partitioning dimension (pd) by performing the *xor* ($\oplus$) operation:

$$\text{pd} : \text{h\_to\_c(max-lower)} \oplus \text{h\_to\_c(min-higher)}$$

In order to find the exact value of this dimension i (0 or 1), we calculate a variable j as:

$$\text{j} : \text{pd} \wedge \text{h\_to\_c(max-lower)}$$

If j evaluates to '00', it indicates that the value at the ith dimension is 0, otherwise 1. We then compare the value of j with that of the previously obtained lower n-point and upper n-point of the current-query-region. If the values (0 or 1) at dimension i, of lower and/or upper n-points is the same as that of the value at the ith dimension of j, then the current query region intersects with the nodes.

We extend our previous example to show how two next-matches, i.e. 0111 and 1100, are calculated for the query region Q2 with the help of the tree representation of the Hilbert Curve as shown in Fig. 2 in the following steps:

Step 1: Tree Level 1 (root): The current-subspace-key is initialized as the subspace-key of subspace 00, i.e. to 0000. Since we are at root level, the lower and upper bounds of current-query-region are same as original query region, i.e. (01,10) and (10,11). The n-points enclosing the current-query-region at this level are formed from the top bits taken from its coordinates. Thus, the lower n-point is 01 and the upper n-point is 11. In order to find the lowest subspace intersecting with the current-query-region at root, the binary search proceeds as follows:

Step 1.1: The first iteration of binary search determines whether the query region intersects with the lower subspaces (00 and 01) in the following manner. First, pd is calculated as h_to_c(01) $\oplus$ h_to_c(10) which evaluates to 01 $\oplus$ 11, i.e 10. This implies that lower subspaces 00 and 01 have the same coordinate value at x dimension and higher subspaces 10 and 11 have the opposite coordinate value at the same x

dimension. Secondly, j is calculated as h_to_c(01) $\wedge$ pd which evaluates to 01 $\wedge$ 10, i.e. 0. This implies that lower subspaces have the value of 0 for their coordinate in the x dimension and higher subspaces have the value of 1 for their coordinate at x dimension. This is also confirmed by Fig. 2. Since the lower n-point also has the value of 0 for its x coordinate, the current-query-region must intersect with the lower subspaces. We also note that, since the upper n-point has the value of 1 for its x coordinate, the higher subspaces 10 and 11 also intersect with the current-query-region and if the next-match is not found in lower subspaces, it will be found in one of higher subspaces.

Step 1.2: The second iteration of binary search now determines the lowest subspace, among 00 and 01 subspaces, intersecting with current-query-region. First, pd is calculated as h_to_c(00) $\oplus$ h_to_c(01) which evaluates to 00 $\oplus$ 01, i.e 01. Secondly, j is calculated as h_to_c(01) $\wedge$ 00, i.e. 0. This implies that subspace 00 has a value of 0 and subspace 01 has a value of 1 for their y coordinate. Since the lower and upper n-point have a value of 1 for its y coordinate, subspace 01 is the lowest among the lower subspaces (00 and 01) which intersects with the current-query-region. Binary search at root node shows that subspace 01 is the lowest subspace which intersect with the current-query-region. The next-match is modified to 01.

Step 2: Tree Level 2: The search for next-match now descends one level down to level 2 following the subspace 01 in the root node. The current-query-region is restricted to subspace 01 which has the lower and upper bound coordinates of (00,00) and (01,01). The current-query-region is then calculated as the intersection of original query bounds with the 01 subspace bounds ((01,10) $\cap$ (00,10) and (10,11) $\cap$ (01,11)). Query lower bound coordinates which are less than the restricted search space equivalents are increased and upper bound coordinates which are greater than the restricted search space equivalents are decreased. The current-query-region is then bounded by the points (01,10) and (01,11). Similar to the previous steps, the first iteration of binary search finds that the current-query-region intersects only with the higher subspaces. The second iteration of binary search finds that the subspace 10 is the lowest subspace intersecting with the current-query-region. The next-match is modified to 0110. Since there are no more levels to descend, calculate-next-match() terminates and the search for the next-match is now complete.

Step 3: Tree Level 1: In the next step, current-subspace-key is set to the subspace-key of the subspace following the one just searched, i.e. 1000. A binary search of root node finds that subspace 10 is the lowest subspace intersecting with the current-query-region, i.e. (01,10) and (10,11). The next-match is modified to 10.

Step 4: Tree Level 2: The search for next-match now descends one level down to level 2 following the subspace 10 in the root node. The current-query-region is then restricted to bounds (10,10) and (10,11). The binary search determines that 00 is the lowest subspace intersecting with the query region. The next-match to the current-subspace-key is determined to be the H-value of point D (10,10), i.e. 1000, current-subspace-key is its own match. As we are at the leaf level, the search for next-match is now complete. After getting the required next-matches, we calculate the sub-ranges in the following manner. The lower bound of a sub-query is set as the next-match. The upper bound is set as the subspace-key of the successor subspace minus one, if its not the last logical

subspace. If the subspace is the last logical subspace, then the upper bound is set as the H-value of the last point on the curve. Thus, for the previous example, we get the sub-ranges as (0110,0111) and (1000,1011). After calculating the required sub-ranges, we use P-Grid's search mechanism to forward the sub-queries to the responsible node (line 7). For example, in Fig. 5, the sub-query (0110,0111) will be forwarded to node 1 and sub-query (1000,1011) will be forwarded to node 2. Upon getting the request, each node will search their local storage and return only those points which intersect with the sub-query to the node which has issued the query.

The complexity of the range query algorithm depends on two factors, the order of the curve which determined by the number of bits in the coordinate value of each dimension and the number of dimensions. Also, of the operations performed during each iteration, none has a complexity which exceeds O(n) where n is the number of bits in coordinate value of each dimension. Thus, the overall complexity of the range querying algorithm is as O(kn) where k is the number of iterations and where n is the number of bits in coordinate value of each dimension.

Algorithm 3 Range Query Processing (q_l, q_h)

Input: query lower point q_l, query higher point q_h
Output: result set R_q

1: $R_q \leftarrow \phi$
2: $S_r \leftarrow \phi$
3: $H_l \leftarrow computeH\text{-}value(q_l)$
4: $H_h \leftarrow computeH\text{-}value(q_h)$
5: $S_r \leftarrow calculateSubRanges(H_l, H_h)$
6: **for each** $s \in S_r$ **do**
7: $\quad R_q \leftarrow PGrid.Search(s)$
8: **end for each**
9: **return** R_q

6.3 *k*NN query processing

Given a set of points N in a d-dimensional space S and a query point $q \in S$, our query processing algorithm returns a set of $k \in N$ points which are closer to q according to some distance function. It is challenging to execute kNN query efficiently in overlay networks as we do not have any prior knowledge of data distribution among the nodes. Recent solutions proposed in [48–50] uses different distributed data structures built on decentralized P2P systems but such solutions are not scalable. [51,52] proposed solutions based on MapReduce framework to process k nearest neighbor query on large volumes of data. However, such methods incur high query latency.

To alleviate these problems, we present a simple query processing strategy. Our kNN query processing algorithm iteratively performs range search with an incrementally enlarged search region until k points are retrieved. Algorithm 3 illustrates the steps of our algorithm. In line 2, we first construct a range r, centered at the query point q and with initial radius $\delta = D_k/k$, where D_k is the estimated distance between the query point q and its k^{th} nearest neighbor. D_k can be estimated by using the equation [53]:

$$D_k \approx \frac{2\sqrt[d]{\Gamma(\frac{d}{2}+1)}}{\sqrt{\pi}} \left(1 - \sqrt{1 - \sqrt[d]{\frac{k}{N}}}\right) \tag{1}$$

where $\Gamma(x+1) = x\Gamma(x)$, $\Gamma(1) = 1$ and $\Gamma(\frac{1}{2}) = \frac{\pi}{2}$, d is the dimensionality and N is the cardinality.

After getting the required lower (q_l) and upper (q_u) bounds of the range query in line 5 and 6, we perform a parallel range search in line 7 to get desired k points in the result set. If k points are not retrieved for the first time, we increase the range (line 11) and repeat the process from line 5 to 12. The complexity of our algorithm depends on the two factors, the data distribution among the nodes and the value of k.

Algorithm 4 k Nearest Neighbors (q, k)

Input: query point q, number of nearest neighbors k
Output: k nearest neighbors

```
1: Q_result ← φ
2: δ ← estimateRadius(k)
3: r ← δ
4: while true do
5:     q_l ← q − r
6:     q_h ← q + r
7:     Q_result ← RangeSearch(q_l, q_h)
8:     if |Q_result| ≥ k then
9:         return top k results of Q_result
10:    else
11:        r ← r + δ
12:    end if
13: end while
```

7 Experimental evaluation

We implemented M-Grid on Amazon EC2 with a cluster size of 4, 8, and 16 nodes. Each of these nodes is a medium instance of EC2 consisting of 4 virtual cores, 15.7 GB memory, 1.6 TB HDD configured as a RAID-0 array and Centos 6.4 OS. The nodes are connected via a 1 GB network link. The data storage layer was implemented using Hadoop 1.2.1 and HBase 0.94.10. Experiments for point, range and kNN queries were carried out on a synthetic dataset containing 400 million points. This dataset was generated using a network based generator of moving objects [54] which simulated the movement of 40,000 objects on the road map of San Francisco bay area. Each object moved 10,000 steps and reported its location (longitude, latitude) at successive timestamps. The dataset follows a skewed distribution since the generator uses a real world road network. We ran a simple MapReduce (MR) job to compute the minimum and maximum values of points (bounds of the regions) in the dataset and set the path of the nodes according to the common prefix of the H-value of those points. This helped us to efficiently distribute the dataset among the nodes using the rowkeys by matching the predefined subspace. Note that even if the data distribution changes and target moves,

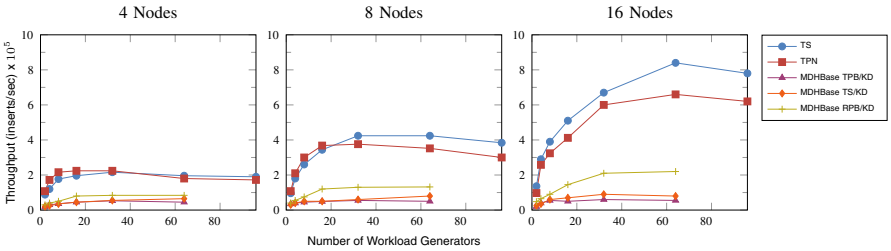


Fig. 7 Effect of varying the number of workload generators

rowkeys will be inserted into the correct node using the matching predefined subspace. We also kept the precision on longitude and latitude values as 1 meter by 1 meter.

We performed extensive experimentations on 2-d and 3-d datasets to show the effectiveness of M-Grid's TPN and TS data models. Index layer using Hilbert Curve (H-order) without the overlay layer was implemented as the baseline. We also evaluated M-Grid's performance against MD-HBase [14] indexing scheme's Table per Bucket (MDH-TPB) and Table Share (MDH-TS) data model.[1] Furthermore, we compared the performance of range queries with MapReduce.

7.1 Performance of insert

The growing trend in LBSs are characterized by their need for scalability. We evaluated M-Grid's scalability using YCSB [55] benchmarking tool. Figure 7 depicts the performance of insert throughput as a function of load on the system on a cluster having 4, 8 and 16 nodes. We varied the number of workload generators from 2 to 96 where each workload generated 10,000 inserts per second based on Zipfian distribution. We ran the workload generator simultaneously on different nodes and aggregated the results. For TS model, the insert throughput scales almost linearly as the number of workload generators increases in accordance to the horizontal scalability provided by HBase. However, TPN model's insert throughput first increases and then decreases as a result of the insertion trend; for a small number of workload generator, TPN model efficiently uses a systems's resources to insert the data simultaneously in different tables. For a location update interval of 60 seconds, the TS model achieved a peak throughput of approximately 840K inserts per second and can handle around 48–52 (840×60) million users. Whereas, the TPN model achieved a peak throughput of approximately 660K inserts per second and can handle around 38–42 (660×60) million users. Moreover, the performance of both designs exceeds MD-HBase by over 4 times and the gap becomes larger as the number of nodes increases. The reason behind MD-HBase's low scalability is the cost associated with splitting the index layer which blocks other operations until its completion. In M-Grid, there is no splitting cost associated with insert operation as the TPB design stores all the data on the responsible node and the TS design allow us to pre-split the table before insertion.

[1] We could not evaluate the performance of MD-HBase for all the experiments as the authors have only published results for 3-d dataset on a 4 nodes cluster size except for insert throughput experiment.

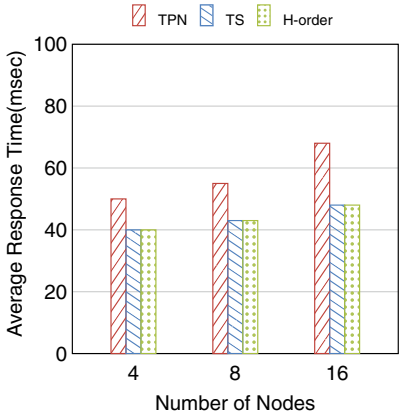


Fig. 8 Performance of point query (D = 3)

7.2 Performance of point and range queries

Multidimensional point and range queries are the most frequent queries in LBSs. M-Grid processes the point query by directly querying the HBase table. On the other hand, range queries are processed by first dividing it into multiple sub-queries and then simultaneously forwarding each sub-query to the responsible node by using the overlay layer. Figure 8 shows the effect of varying the number of nodes on the performance of 3-d point queries for TPN, TS and H-order models. When we increase the number of nodes, the average response times increases for all the models except for TPN model. The TS and H-order models have the same response time as they both use the same querying strategy. However, the response time of the TPN model is longer than the other models because of the cost associated with searching the routing table to find the relevant node. We also found that the response time for processing 2-d point queries is approximately equal to the processing of 3-d point queries.

Figures 9 and 10 shows the performance of 2-d range queries for TPN, TS, H-order, and MR models with different selectivity and node size respectively. The query response time of TPN, TS and H-order models increases almost linearly as we increase the number of nodes (Fig. 9). On the contrary, the response time of MR remains constant as it performs a full scan of the dataset to execute the query and thus, its response time is independent of the selectivity. The performance of TS and TPN model exceeds that of other models, especially for queries with larger selectivity. Since range queries with larger search area will intersect with more subspaces resulting in several sub-queries. However, the increase is not exponential since sub-queries are executed in parallel. The results are corroborated from Fig. 10, which depicts the effect on average range query response time by increasing the number of nodes and keeping selectivity as 10%. The average query response time decreases as we increase the number of nodes since an increase in the number of nodes results in efficient distribution of data. Furthermore, the performance of TPN model is superior than that of TS model because TPN model stores all the data locally on the nodes whereas TS model distributes the data across the clusters. In Figs. 11 and 12 we perform the set of experiments done for 2-d dataset on a 3-d dataset. Figure 10 shows the performance of 3-d range query as a

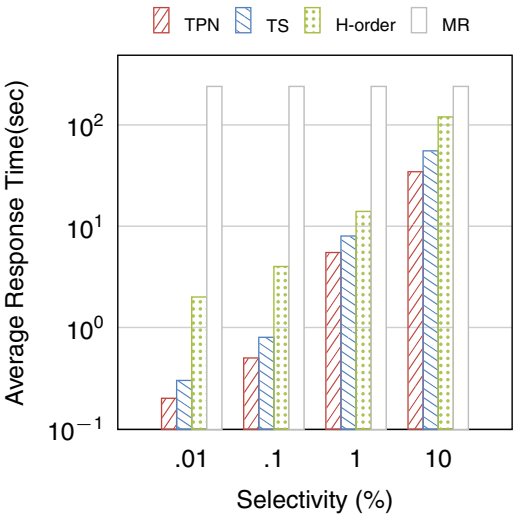


Fig. 9 Performance of range query (nodes = 4, D = 2)

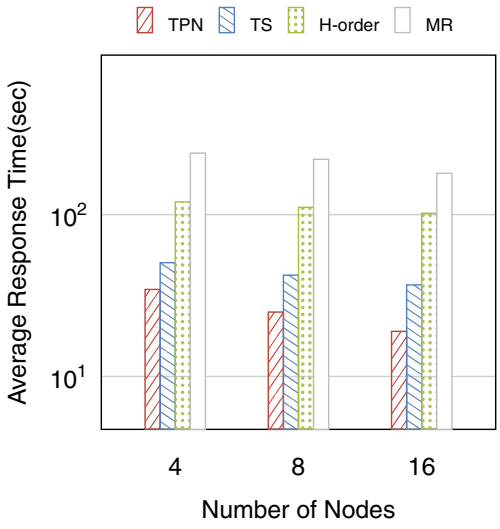


Fig. 10 Performance of range query (selectivity = 10%, D = 2)

function of selectivity on a 4 node cluster. When we increase the selectivity, the average query response time of our models increases. In this experiment, we also compared the results of our models with MD-HBase's TPB and TS data model in addition to H-order and MR models. The results of our models shows better performance even for larger selectivity. This is because, in MD-HBase uses additional index layer for pruning result sets whereas in our schemes there is no such overhead. Also, both of our designs show three order of magnitude improvement over MapReduce model. The results obtained from 2-d datasets are much better than that of 3-d dataset, since the complexity of our range processing algorithm depends on the number of dimensions and on the order of the curve, i.e. the number of bits in each dimension.

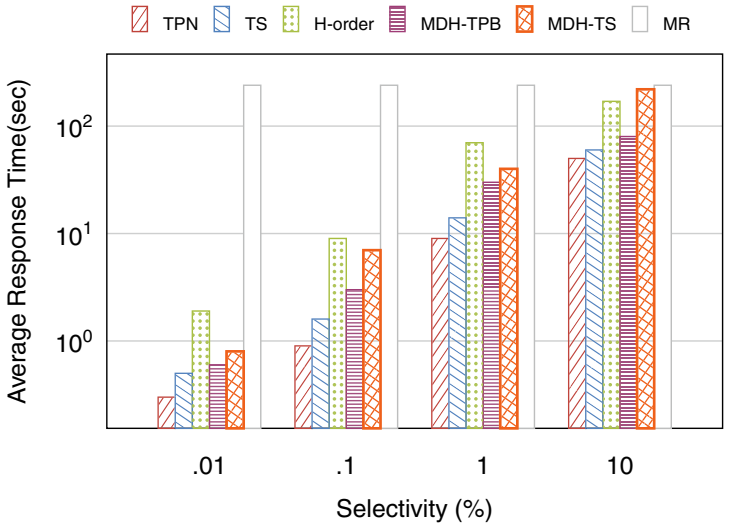
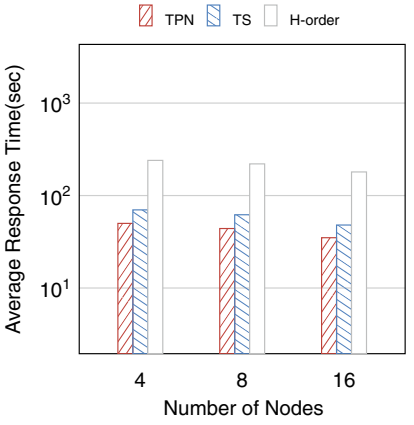


Fig. 11 Performance of range query (nodes = 4, D = 3)

Fig. 12 Performance of range query (selectivity = 10%, D = 3)



7.3 Performance of *k*NN query

M-Grid processes the k nearest neighbor (kNN) query iteratively. We first estimate the distance between the query point and its k^{th} nearest neighbor using (1), which becomes the initial search radius. Then, we perform a range search to retrieve k results. If the k results are not returned, we increase the search space and perform the range search again. Thus, the performance of kNN computation is directly correlated to the performance of our range search. Figure 13 shows the performance of kNN queries for TPN, TS and H-order models on a 2-d dataset by varying the value of k from 1 to 10K on a 4 node cluster. The average response time of kNN query increases for all the models when the value of k increases, since the query space increases as we increase k. However, this increase in average response time is not exponential because range

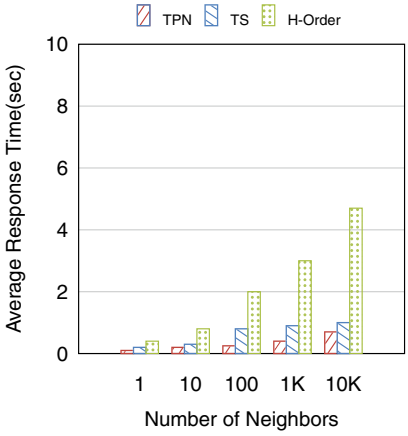


Fig. 13 Performance of kNN query (nodes = 4, D = 2)

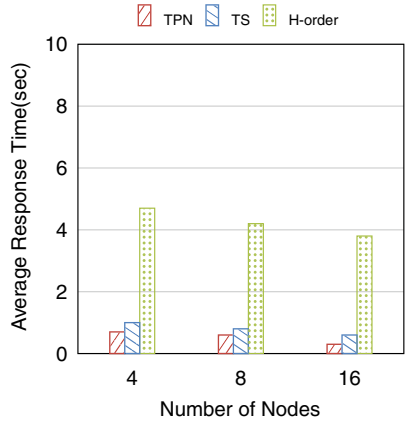


Fig. 14 Performance of kNN query (k = 10K, D = 2)

queries with larger search space are processed using more nodes. Our obtained results are validated in Fig. 14, where we set the value of k to 10K but increase the number of nodes from 4 to 16. The results of this experiment shows that the average query response time decreases as the number of nodes in the cluster increase because larger range queries will intersect more subspaces and thus more nodes will be involved. However, the decrease is again not exponential because after issuing a range query, the system waits until it receives results from all of the nodes involved. In both the experiments, the TPN and TS models show a performance improvement of 4 to 5 times as compared to H-order design. In Figs. 15 and 16 we performed the set of experiments of 2-d dataset on a 3-d dataset. We show the effect of varying the parameter k on a 4 node cluster and compare the results with MD-HBase and H-order designs in Fig. 15. The average response time of our models increase with the increase in value of parameter k, which validates the results depicted in Fig. 13. However, 3-d kNN queries take more time to process as compared to 2-d since the complexity of performing range queries increases with number of dimensions. For $k = 1$, the TPN and TS models gives superior performance with an average response time of approximately 500ms and

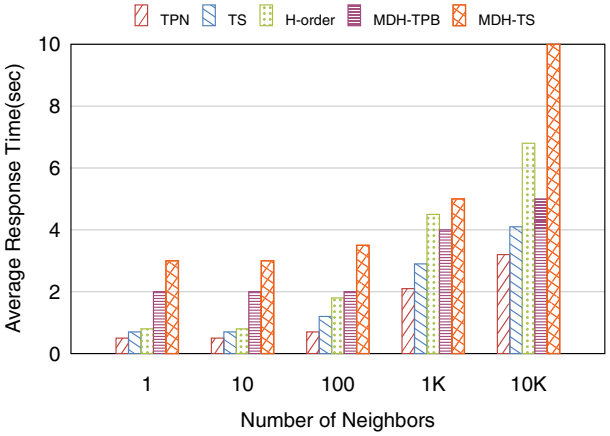
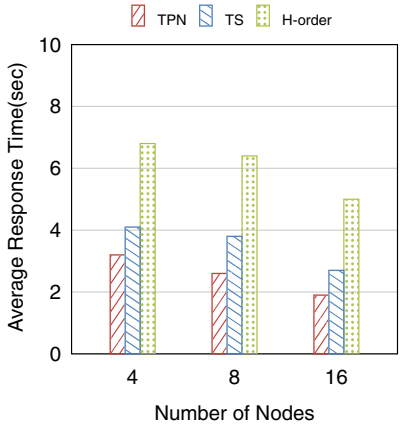


Fig. 15 Performance of kNN query (nodes = 4, D = 3)

Fig. 16 Performance of kNN query (k = 10K, D = 3)



700ms respectively, in contrast to H-order, MD-HBase TPB and TS models being approximately 800, 2000 and 3000 ms, respectively. In our experiments, we also observed that for k<100, the search space does not expand large enough to intersect more than two subspaces. For k>100, the kNN query processing results in range searches with larger radius which intersects with more than two subspaces. Thus, the performance of H-order model degrades for k>100 while that of TPN and TS models continue to show better performance. In Fig. 16, we compare the effect of varying the number of nodes on the performance of 3-d kNN queries by setting the value of k as 10K. The results for this experiment are consistent with those of the experiments performed for 2-d dataset (Fig. 14). We expect the performance of our designs for kNN processing to be better on uniform dataset as the Eq. 1 provides more accurate estimation of initial search range for uniform dataset. Thus, the kNN processing will require less number of range search iterations to retrieve k results.

8 Conclusion and future work

In this paper, we presented and evaluated M-Grid, a multidimensional data distribution and indexing framework for location aware services on cloud platform. M-Grid is a scalable, completely decentralized and platform independent indexing framework which can efficiently process point, range and nearest neighbor queries. M-Grid first arranges the nodes leased from cloud in a modified P-Grid overlay network which virtually partitions the whole space in a binary-trie structure. Next, for efficient storage and retrieval of multidimensional data, we exploited Hilbert Space Filling Curve based linearization technique to convert multidimensional data into one dimensional binary keys. This technique allowed us to map the keys to the peers according to their paths while preserving data locality. We designed and developed algorithms to dynamically process range and nearest neighbor queries which allowed us to remove the limitation of creating and maintaining a separate index table. We conducted extensive experiments using a cluster size of 4, 8 and 16 modest nodes on Amazon EC2. Our results shows that M-Grid achieves almost four times better performance than its previous counterpart MD-Hbase. In future, we wish to extend our framework by providing it the ability to handle data skewness in a dynamic way and to support wider variety of multidimensional queries including skyline and spatial-joins. We also want to develop efficient subspace splitting algorithms for load balancing and efficient data replication algorithms for making M-Grid more fault-tolerant. We also want to evaluate P-Ring p2p network [42] to check if it can address all the requirements of our model, or possible modify and how it compares with P-Grid in performance.

Acknowledgements This project is supported partially from an AFRL grant, and NSF Grants IIP-1238321 and CNS-1461914.

References

1. http://www.itu.int/
2. Union, I.T.: The world in 2015: Ict facts and figures. [Online]. Available: https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2015.pdf (2015)
3. McMahon, M., Steketee, C.: Investigation of proposed applications for lbs enabled mobile handsets. In:ICMB '06. International Conference on Mobile Business, 2006, pp. 26–26 (2006)
4. [Online]. Available: http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/index.htm
5. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '84, pp. 47–57 (1984)
6. Finkel, R., Bentley, J.: Quad trees a data structure for retrieval on composite keys. Acta Informatica **4**, 1–9 (1974)
7. http://www.ibm.com/software/products/en/db2spaext
8. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, vol. 7, ser. OSDI '06, pp. 15–15 (2006)
9. http://hbase.apache.org/
10. http://cassandra.apache.org/
11. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)

12. Hilbert, D.: Ueber stetige abbildung einer linie auf ein flashenstuck. Mathematishe annalen **32**, 459–460 (1893)
13. Aberer, K., Cudré-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., Schmidt, R.: P-grid: a self-organizing structured p2p system. SIGMOD Rec. **32**, 29–33 (2003)
14. Nishimura, S., Das, S., Agrawal, D., Abbadi, A.E.: Md-hbase: design and implementation of elastic infrastructure for cloud-scale location services. Distrib. Parallel Databases **31**, 289–319 (2014)
15. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. **41**, 205–220 (2007)
16. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. **2**(1), 1277–1288 (2008)
17. Ghemawat, S., Gobioff, H., Leung, S.-T.: The google file system. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, ser. SOSP '03, pp. 29–43 (2003)
18. http://hadoop.apache.org/
19. Wu, S., Wu, K.-L.: An indexing framework for efficient retrieval on the cloud. IEEE Data Eng. Bull. **32**(1), 75–82 (2009)
20. Wang, J., Wu, S., Gao, H., Li, J., Ooi, B.C.: Indexing multi-dimensional data in a cloud system. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '10, pp. 591–602 (2010)
21. Zhang, X., Ai, J., Wang, Z., Lu, J., Meng, X.: An efficient multi-dimensional index for cloud data management. In: Proceedings of the First International Workshop on Cloud Data Management, ser. CloudDB '09, pp. 17–24 (2009)
22. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
23. Ding, L., Qiao, B., Wang, G., Chen, C.: An efficient quad-tree based index structure for cloud data management. In: Web-Age Information Management. Lecture Notes in Computer Science, vol. 6897, pp. 238–250 (2011)
24. Suprio Ray, R.B., Goel, A.K.: Supporting location-based services in a main-memory database. In: Proceedings of the IEEE International Conference on Mobile Data Management (MDM), (2014)
25. Jagadish, H., Ooi, B.-C., Vu, Q.H., Zhang, R., Zhou, A.: Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In: Data Engineering, 2006. in: ICDE '06 Proceedings of the 22nd International Conference on, pp. 34–34 (2006)
26. Li, F., Chen, R., Zhou, C., Zhang, M.: A novel geo-spatial image storage method based on hilbert space filling curves. In: 2010 18th International Conference on Geoinformatics, pp. 1–4 (2010)
27. Pavanakumar, M., Kaushik, K.: Revisiting the space-filling curves for storage, reordering and partitioning mesh based data in scientific computing. In: 2013 20th International Conference on High Performance Computing (HiPC), pp. 362–367 (2013)
28. Hu, C., Zhao, Y., Wei, X., Du, B., Huang, Y., Ma, D., Li, X.: Actgis: A web-based collaborative tiled geospatial image map system. In: 2010 IEEE Symposium on Computers and Communications (ISCC), pp. 521–528 (2010)
29. Butz, A.R.: Alternative algorithm for hilbert's space-filling curve. IEEE Trans. Comput. **20**, 424–426 (1971)
30. Bially, T.: Space-filling curves: their generation and their application to bandwidth reduction. IEEE Trans. Inf. Theory **15**(6), 658–664 (1969)
31. Hamilton, C., Rau-Chaplin, A.: Compact hilbert indices for multi-dimensional data. In: First International Conference on Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007, pp. 139–146 (2007)
32. Morton, G.: A computer oriented geodetic data base and a new technique in file sequencing. *International Business Machines Company*, [Online]. Available: http://books.google.com/books?id=9FFdHAAACAAJ (1966)
33. Gray, F.: Pulse code communication. (1953)
34. Moon, B., Jagadish, H., Faloutsos, C., Saltz, J.: Analysis of the clustering properties of the hilbert space-filling curve. Knowl. Data Eng. IEEE Trans. **13**, 124–141 (2001)
35. Abel, D.J., Mark, D.M.: A comparative analysis of some two-dimensional orderings. Int. J. Geogr. Inf. Syst. **4**, 21–31 (1990)

36. Mokbel, M.F., Aref, W.G., Kamel, I.: Performance of multi-dimensional space-filling curves. In: Proceedings of the 10th ACM International Symposium on Advances in Geographic Information Systems, ser. GIS '02, pp. 149–154 (2002)
37. Clarke, I., Sandberg, O., Wiley, B., Hong, T.W.: Freenet: A distributed anonymous information storage and retrieval system. In: International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability, pp. 46–66 (2001)
38. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01, pp. 149–160 (2001)
39. Jagadish, H.V., Ooi, B.C., Vu, Q.H.: Baton: A balanced tree structure for peer-to-peer networks. In: Proceedings of the 31st International Conference on Very Large Data Bases, ser. VLDB '05, pp. 661–672 (2005)
40. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. SIGCOMM Comput. Commun. Rev. **31**, 161–172 (2001)
41. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, ser. Middleware '01, pp. 329–350 (2001)
42. Crainiceanu, A., Linga, P., Machanavajjhala, A., Gehrke, J., Shanmugasundaram, J.: P-ring: An efficient and robust p2p range index structure. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '07, pp. 223–234 (2007)
43. Aberer, K., Datta, A., Hauswirth, M., Schmidt, R.: Indexing data-oriented overlay networks. In: Proceedings of the 31st International Conference on Very Large Data Bases, ser. VLDB '05, pp. 685–696 (2005)
44. Datta, A., Hauswirth, M., John, R., Schmidt, R., Aberer, K.: Range queries in trie-structured overlays. In: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing, ser. P2P '05, pp. 57–66 (2005)
45. Rosch, P., Sattler, K., von der Weth, C., Buchmann, E.: Best effort query processing in dht-based p2p systems. In: 21st International Conference on Data Engineering Workshops, 2005, pp. 1186–1186 (2005)
46. Lawder, J.K.: Querying multi-dimensional data indexed using the hilbert space-filling curve. SIGMOD Rec. **30**, 2001 (2001)
47. https://code.google.com/p/uzaygezen/
48. Tang, Y., Xu, J., Zhou, S., Lee, W.-C., Deng, D., Wang, Y.: A lightweight multidimensional index for complex queries over dhts. IEEE Trans. Parallel Distrib. Syst. **22**, 2046–2054 (2011)
49. Tanin, E., Nayar, D., Samet, H.: An efficient nearest neighbor algorithm for p2p settings. In: Proceedings of the 2005 National Conference on Digital Government Research, ser. dg.o '05, pp. 21–28 (2005)
50. Gao, J.: Efficient support for similarity searches in dht-based peer-to-peer systems. In: In IEEE International Conference on Communications (ICC07 (2007)
51. Lu, W., Shen, Y., Chen, S., Ooi, B.C.: Efficient processing of k nearest neighbor joins using mapreduce. Proc. VLDB Endow. **5**, 1016–1027 (2012)
52. Stupar, A., Michel, S., Schenkel, R.: Rankreduce - processing k-nearest neighbor queries on top of mapreduce. In: In LSDS-IR, (2010)
53. Tao, Y., Zhang, J., Papadias, D., Mamoulis, N.: An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. IEEE Trans. Knowl. Data Eng. **16**, 1169–1184 (2004)
54. Brinkhoff, T.: A framework for generating network-based moving objects. Geoinformatica **6**, 153–180 (2002)
55. Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing, ser. SoCC '10, pp. 143–154 (2010)