2014

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

Shashank Kumar

M-GRID : A DISTRIBUTED FRAMEWORK FOR MULTIDIMENSIONAL INDEXING AND QUERYING OF LOCATION BASED BIG DATA

by

SHASHANK KUMAR

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2014

Approved by

Dr. Sanjay Madria, Advisor
Dr. Sriram Chelappan
Dr. Dan Lin

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, handling of high access scalability, support for millions of users, real time querying capability and analysis of 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 value from very large volumes of data while being highly available, fault-tolerant and scalable, hence providing much needed features to support LBSs. However complex queries on multidimensional data cannot be processed efficiently as they do not provide means to access multiple attributes.

In this thesis we present MGrid, a unifying indexing framework which enables key-value stores to support multidimensional queries. We organize a set of nodes in a P-Grid overlay network which provides fault-tolerance and efficient query processing. We use Hilbert Space Filling Curve based linearization technique which preserves the data locality to efficiently manage multi-dimensional 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. Our approach is completely independent from the underlying storage layer and can be implemented on any cloud infrastructure. Experiments on Amazon EC2 show that MGrid achieves a performance improvement of three orders of magnitude in comparison to MapReduce and four times to that of MDHBase scheme.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude and appreciation towards my advisor Dr. Sanjay Madria for his continuous support of my M.S. study and research. I have greatly benefited from his, knowledge, advice and guidance, which was instrumental in the completion of this thesis. I am thankful to him for allowing some extra time to finish my experimental work.

I would also like to thank the members of my dissertation committee, Dr. Sriram Chellappan and Dr. Dan Lin, for their constructive comments and feedback.

I would like to acknowledge the contributions of the current and former members of the research group especially, Dr. Vimal Kumar, Brijesh Kashyap Chejerla, Amartaya Sen and Dr. Roy Cabaniss, with whom I spent countless hours in the lab and engaged in numerous fruitful discussions. I am thankful to my k nearest friends especially Sahil Parikh, Abhinav Saxena, Aditi Sharma, Manish Sharma and Anand Kishore, without them I would have graduated one year earlier.

I would also like to thank Daniel@uzaygezen and Deependra,John@Eucalyptus for helping me in various stages of the project.

Finally, I dedicate this thesis to my father Himmat Singh, mother Manorama, sister Ekta and brother Ashish for their constant support and encouragement in all my professional endeavors. I hope this would make them proud.

TABLE OF CONTENTS

Page

SECTION

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

Cloud computing is a model for enabling ubiquitous, on-demand, utility based access to shared computing resources (hardware and software) where such resources are delivered as a utility based service. The Cloud has provided the IT industry a paradigm shift in the way services are delivered. According to NIST [1] the essential characteristics of this service oriented model includes *on demand self-service* i.e. the ability to obtain, configure and deploy services without the assistance of service provider; *broad network access* which provide device and location independent access to the users; *resource pooling* which enables service provider to pool their resources to serve multiple users at once; *scalability and rapid elasticity* for adding new resources as per the need of the user and a measured pay-per-use model based service. The cloud's service model provides three levels of abstraction. The lowest level we have *Infrastructure as a Service* (**IaaS**) where cloud vendors provide physical resources like servers, storage and networking components. *Platform as a Service* (**Paas**) is the middle level of abstraction which provides an application deployment framework. Applications need to be modeled after the PaaS provider's framework and the provider deploys and scales the application. *Software as a Service* (**Saas**) is the highest level of abstraction where the SaaS provider install and operate domain specific (like supply chain management) application in the cloud. As cloud computing is becoming increasingly popular and useful, our industry is gradually shifting from in-house hosting to cloud hosting and platforms including cloud data storage platforms.

In this chapter, we first start with an introduction of how location based services can benefit from cloud computing model and analyze the challenges of deploying such applications in cloud. Next, we discuss the motivation of our research which aims to provide advance features missing from current cloud data serving systems. Then, we present specific goals and scope of our research and finally, we give an overview of our solution with a summary of our main contributions of this thesis.

1.1. TRADITIONAL ARCHITECTURE OF LOCATION BASED APPLICATIONS

The increasing need for mobility and recent advances in wireless technology have created one of the most promising value added services which are commonly known as Location Based Services (LBS). LBS are provided to mobile users according to their proximity. Such services use the ability to dynamically determine and transmit the location of users by the means of their communication device within a mobile network [2]. These services provide means to search for information about users, physical locations, finding routes to specified destinations, analyze real time traffic etc.

Currently, a wide range of LBS are available for mobile users [3] which includes:

- Mapping applications, providing mapping directions to a vehicle driver
- City guides, providing information for travelers about a given area
- Mobile yellow pages, assisting mobile users to locate the services they need
- Location-aware marketing, triggering advertisements based on proximity to an area

Figure 1.1 shows the 3-tier architecture for location based services which consists of a client tier, a middle tier and a database tier. *Client Tier*: In this tier, the user interacts with the system. User on their mobile devices visualizes web pages which contain spatial information. These web pages display interactive maps, search fields etc. and tracks actions performed by the user. For example, a user can issue a query to find nearby places to its current geographic location. Such queries are

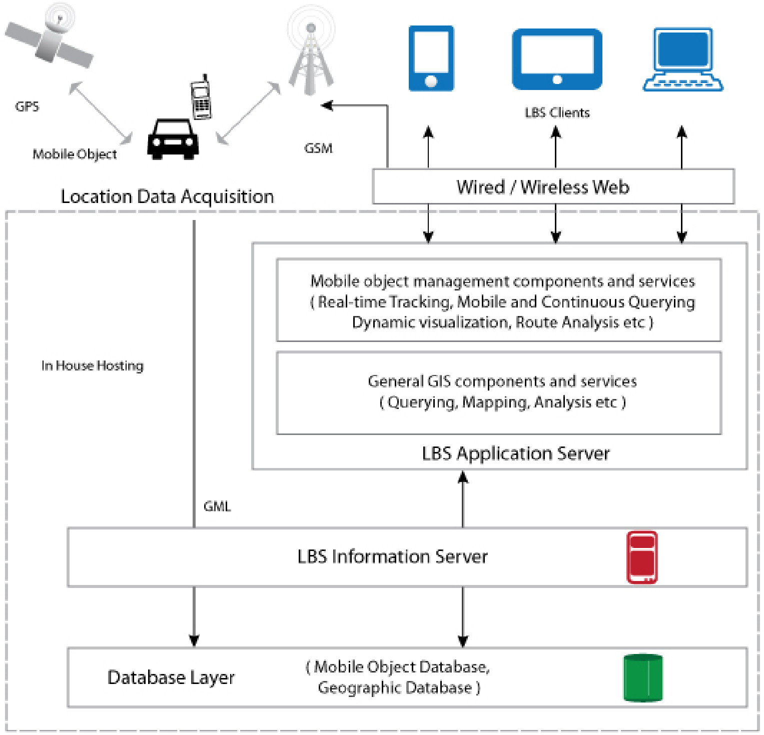
GPS
Mobile Object
GSM
LBS Clients
Location Data Acquisition
Wired / Wireless Web
Mobile object management components and services (Real-time Tracking, Mobile and Continuous Querying Dynamic visualization, Route Analysis etc)
General GIS components and services (Querying, Mapping, Analysis etc)
LBS Application Server
In House Hosting
GML
LBS Information Server
Database Layer
(Mobile Object Database, Geographic Database)

Figure 1.1 Traditional 3-Tier architecture of Location Based Applications

then routed to the LBS. In a typical scenario a web page displays a dynamic map served by the map renderer and other information from the LBS which is situated in the middle tier. *Middle Tier*: This tier provides the core services and functionalities of the system. A web application hosts the Web Server Pages which constitute the access point to the system for the users and the LBS, in which several web services are deployed and exposed on the Internet. This LBS tracks the user sessions, handles the user profiles, logs all of the relevant actions performed by the clients and keeps track of the user location when available. The LBS application server handles user request, process/analyze them and provides the result or services to the user. *Database Tier*: The main function of this tier is to provide the other tiers with spatial information through LBS information server. It consists of a spatial DBMS, which stores spatial datasets (e.g. longitude and latitude points) and data related to the user profiles. The relevant spatial datasets can be imported from an external data provider (e.g.

OpenStreetMap, web crawlers etc.) which can be stored in the spatial DBMS and updated on a regular basis.

1.2. DEPLOYING LOCATION BASED APPLICATIONS IN THE CLOUD

The architecture described in section 1.1 though provides much needed flexibility in terms of application development, it still suffers from many drawbacks. First, LBS requires to handle millions of data insert and update requests per second, concurrently. Traditional single master server architecture therefore can become a bottleneck especially database servers and hence provide poor fault tolerance to the system. Next, in order to serve high query throughput and to provide low latency, LBS requires massive parallel processing of requests. Although traditional systems backed by *Relational Database Management Systems* (**RDBMs**) do provide a level of parallelism at the lowest level (query, bulk insert and update), it is still insufficient as the load on the system increases. Not just that, the increase in the load as the number of user increases presents many threatening problems to the shared database architecture. The biggest problem it faces is of *scaling*. In traditional systems, there are two ways to achieve scalability either by *scale-up* or by *scale-out*. *Scale-up* is achieved by using larger and more powerful servers which simply have more processing cores, memory and faster storage disks. However it has to be noted that in a master-slave architecture, we need to *scale-up* both master and slave servers otherwise the slaves will fail to keep up with masters update rate and the reverse is also true. *Scale-up* incurs almost twice or sometimes thrice the cost of initial hardware setup cost and therefore it is not a viable option. *Scale-out* is achieved by increasing the overall capacity of the system by adding more secondary servers. This method also does not provide a feasible solution because scaling the database layer while preserving the strong consistency and referential integrity is not viable [4]. Another problem with *scale-out* is that Location Based Applications typically handle more read queries than write queries which adds further CPU and I/O load on the system. Apart from *scale-up* or *scale-out*, other solutions to provide scalability and load balancing includes *caching* and *sharding*. *Caching* can reduce the load to a level as the system can now serve requests by keeping data and objects in-memory. However maintaining consistency across the servers now becomes a problem as the cache has to be invalidated and refreshed periodically. *Sharding* refers to the process of logically separating the data into horizontal partitions across database servers. It is a very costly operation as the whole process has to be meticulously planned and the storage layout has to be rewritten. The characteristics of cloud computing model in which clusters of commodity servers are used to perform computing tasks with a utility based pay-per-use model, has now became a feasible and an acceptable solution to all the above mentioned problems of the traditional application development architecture. Figure 1.2, shows the best design practice for LBS in a cloud environment. The single master/slave architecture can now be easily elastically scaled-out by leasing virtual machines from cloud vendors as per the need. The low cost, ease of application setup and elastic scalability provided by cloud computing has paved way to a numerous cloud based applications like Facebook, Foursquare, and NetFlix etc. and cloud vendors like Amazon Web Services, Rackspace, HP Cloud Services and Oracle Cloud Services etc.

1.3. MOTIVATION AND CHALLENGES

One of the crucial characteristic which LBS exhibit is the ability to handle massive data generated by millions of service subscribers at once. For example, Foursquare has a user base of 30 million people around the world, Yelp has more than 100 million subscriber with 86 million monthly unique visitors etc. Irrespective of the cloud's abstraction, data forms the central and critical part of LBS.

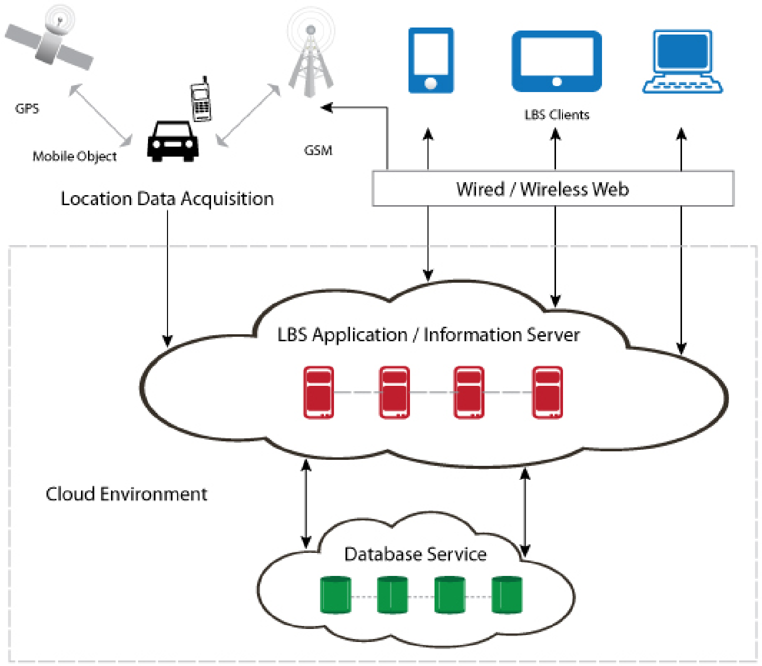

Figure 1.2 Location Based Application architecture in cloud environment

Data processing in cloud can be divided into two categories which are *Online Transaction Processing* (**OLTP**) and *Online Analytical Processing* (**OLAP**). OLTP database is modeled for business transactional data processing applications which are mission-critical in nature for example an online banking application. Such applications require real time processing of transactions where data is changing at the same time and to provide low latency and high throughput. The OLAP databases, on the other hand, are modeled to provide business insights like decision making and planning through data mining. Such databases forms the part of an organizations massive data warehouses. Databases in data mining applications require to handle more complex and longer running queries and also are more read oriented than write oriented. As LBS require to process queries in real time with short latency and high query throughput, our research will be focused on OLTP databases designed for elastically scalable could computing infrastructure.

Relational Database Management Systems (**RDBMs**) were invented by E. F. Codd of IBM research labs in 1970 [5] to handle the transactional workload of OLTP applications. The key characteristics which RDBMs provides includes, a relational data model which represents data in terms of tuples and declarative schemas, *concurrency controls* by providing guarantee for *Atomicity* i.e. either all of the transactions will commit or none will, *Consistency* i.e. data only validated by pre-defined rules will be saved, *Isolation* i.e. multiple transactions should not interfere with each other, *Durability* i.e. transactions will not be reversed upon their completion (ACID), support for *normalization* of data for removing data duplicity and a powerful and rich query and data manipulation language. Today there are number of RDMBs available which includes both commercial (Oracle SQL, IBM DB2 and Microsoft SQL) and open source (MySQL and PostGres).

Although RDBMs have been proved highly successful for traditional transactional based systems, they are inefficient for cloud infrastructures. One of the biggest problems of RDBMs is scaling. The relational architecture does not allow to scale-out the database to many nodes as the requirement of an application to handle traffic increases. Scaling-out RDBMs while providing ACID guarantee is expensive due to distributed synchronization among database server. This can also be well explained by CAP theorem [6] which states that it is impossible for a distributed system to provide consistency, availability and partition tolerance simultaneously. Partition tolerance is essential for LBS built on cloud infrastructure as network partitions are inevitable. Choosing between availability and consistency, consistency is neglected as the LBS is expected to be remain online at all the times [7]. Apart from that, LBS need to handle Big Data which is a term to denote data sets which grows so rapidly in a small period of time, that it cannot be managed on a single system because of storage, CPU cycles and memory constraints. Other LBS characteristics which makes RDMS a poor design choice are, LBS queries are more read oriented than write oriented and LBS queries are attribute focused rather than entity focused. In summary, data management system of LBS should be able to scale out on demand, should provide high availability and fault-tolerance and should be easy to administer.

Recent years saw the emergence of key-value stores (also referred as NOSQL stores) which are modeled to scale-out and provide all the essential features necessary for cloud based applications. Examples of such key-value stores includes Google's BigTable [8], Yahoo's PNUTS [9] and Amazon's Dynamo [10] and their various open source counterparts such as Apache's HBase [11], Facebook's Cassandra [12] etc. Figure 1.3 shows the difference between the traditional RDBMs and the Key-Value stores. Different key-value stores provide different data models for example, HBase is designed to provide availability and partition tolerance whereas Cassandra is designed to provide Consistency and partition tolerance. Irrespective of the data model, the property which is common in all these key-value stores is the key-value abstraction in which data is viewed and stored as independent key-value pairs and the access is supported only at the granularity of single keys. Such abstraction naturally allows efficient horizontal data partition and elastic scalability. This abstraction though satisfy the needs of many present applications, a large number of current web applications need more than a single atomic key access pattern like LBS. The data in LBS is inherently multidimensional which mainly compromises of longitude, latitude, time, user id etc. and therefore it requires a multi-key access.

In key-value stores, data can only be queried based on the key, so a specific keyword or value must be known to perform a search. LBS require to handle mainly three types of queries - point, range and k nearest neighbor (kNN). These queries cannot be implemented efficiently because RDBMs like additional indexes are not available and any such query on a particular key-value store would require scanning of all the keys at a minimum to produce results, essentially making this approach not feasible. Without proper indexing method, even for a simple point query, we need to

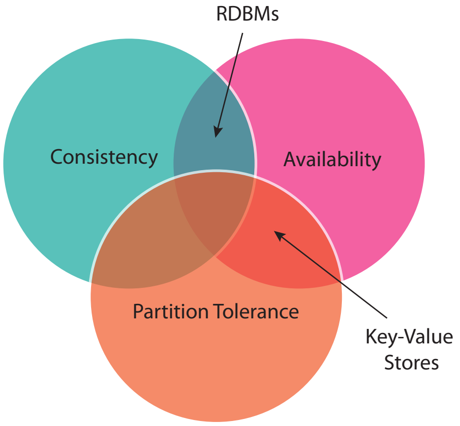

Figure 1.3 RDBMs versus Key-Value stores

scan the whole data set repeatedly and produce the necessary second indexes in an off-line batch manner. Problems with this approach are that the secondary index is not up-to-date and newly inserted data cannot be queried until they are indexed.

Our research goal is motivated by the fact that LBS requires multidimensional indexing capabilities to support its rich real-time querying functionality and scalability which no cloud data serving system supports currently.

1.4. RESEARCH GOAL AND SCOPE

Our ultimate research goal is to build an efficient multidimensional index structure that can be built on an underlying key-value store to provide advance query capabilities for location based applications on cloud infrastructure.

We have also aimed to provide our indexing structure the ability to support skewed data set, a robust mechanism for fault tolerance, replication and consistency management and dynamic provisioning. The thesis focuses on the following lines:

- Distributed Indexing: the design of an efficient multidimensional index structure for location based applications on cloud friendly key-value store.
- Load Balancing: the capability of efficiently handling skewed data.

- Rich Query Functionality: the ability to efficiently process point, range and kNN queries.
- Elastic Scalability: the capability to extend the index structure in the presence of dynamic workload.

1.5. SOLUTIONS OVERVIEW

In this paper, we propose MGrid, a novel distributed multidimensional indexing framework to support LBSs on cloud platform. Because of the characteristics of key-value stores which includes availability, horizontal scalability and a distributed architecture, it became a natural choice for us to use them as MGrid's storage back-end. However, the key challenges in developing an index framework on top of a key-value store are, efficient modeling of multidimensional data and providing it the ability to process complex multidimensional queries. MGrid solves the former by using Hilbert Curve[13] based linearization technique and later by integrating PGrid overlay network. Hilbert Curve maps multidimensional attributes onto single dimensional while preserving its data locality. On the other hand, PGrid arranges the nodes in a virtual binary trie and partitions the multidimensional search space into subspaces. MGrid then processes complex queries by distributing them across the cluster according to P-Grid's prefix based routing mechanism.

In summary, this thesis makes the following contributions:

1. We propose a new multidimensional indexing framework, MGrid, which can efficiently process point, range and kNN queries. MGrid integrates PGrid overlay network and a range partitioned key-value store.
2. We leverage Hilbert Space Filling Curve based linearization technique to convert multidimensional data to a single dimension while preserving its data locality.

3. We propose algorithms which can dynamically process queries on linearized values using PGrid's prefix based routing mechanisms. This removes the overhead of creating and maintaining a separate index table,

4. We performed extensive experimental evaluations on Amazon EC2 to show the effectiveness and efficiency of our framework.

1.6. OUTLINE OF THESIS

- **Section 2** given background information that forms the basis of our research

- **Section 3** presents literature review on previously done related work

- **Section 4** describes the design of our proposed distributed multidimensional indexing framework.

- **Section 5** presents the results of our experimental evaluation

2. BACKGROUND

In this chapter, we present background information for our research. We present the idea of linearization using space filling curves and present Hilbert Space Filling Curve in detail. We also discuss basic techniques for replication management and review peer-to-peer (P2P) overlay networks that are commonly used to facilitate distributed search. We finish this chapter by presenting the overview of P-Grid overlay network which we use to efficiently route queries in order to process complex multidimensional queries.

2.1. LINEARIZATION USING SPACE FILLING CURVE

Linearization is a dimensional reduction method which maps multi-dimensional 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 [14], scientific computing [15] and geographic information systems [16] which require sequential access to datasets. In MGrid, we use Hilbert space-filling curve [13] to index multidimensional points in the underlying uni-dimensional key-value store to promote query efficiency.

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

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

where n is the number of dimensions in a 2^mx2^m space and $n \geq 2$, $m > 1$. This function

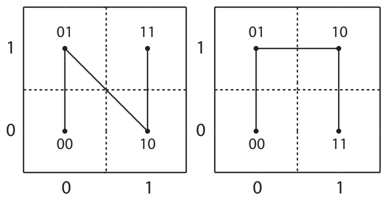

Figure 2.1 A 2-d space and its equivalent First order Hilbert Curve

determines the Hilbert value (H-value) of each point in the original coordinate space where H-value ∈[0, 2^{mn}-1]. Fig. 2.1 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 appropriately rotating and/or reflecting it to fit the new curve [17]. This recursive construction process can also be expressed as a tree structure (Fig. 2.3) to show the correspondence between the coordinate points (n-points) and their H-values in binary notation [18]. 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. 2.1. Also, a collection of nodes at any tree level, i, describes a curve of order i. 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 MGrid, we compute the H-value using the algorithm presented in [19] which uses logical operations to efficiently compute direct and inverse mapping of a point having unequal attributes on the Hilbert Curve.

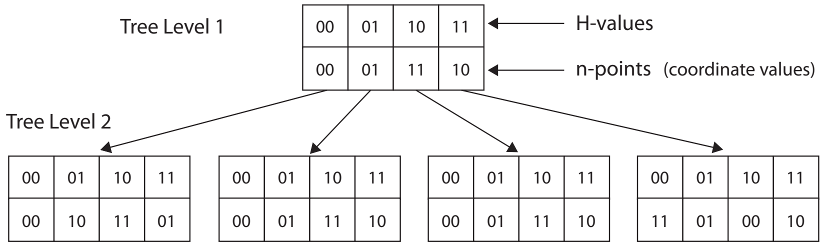

Figure 2.2 A tree representation of the second order Hilbert Curve in 2 dimension

Beside Hilbert Curve, several space-filling curves such as Z-Order Curve [20] and Gray Order Curve [21] are proposed. Fig. 2.3 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 MGrid as it has superior clustering and strong locality preserving properties as compared to other space-filling curves [22] [23] [24]. These properties help MGrid to achieve efficient clustering of the location points in the database resulting in low query latency.

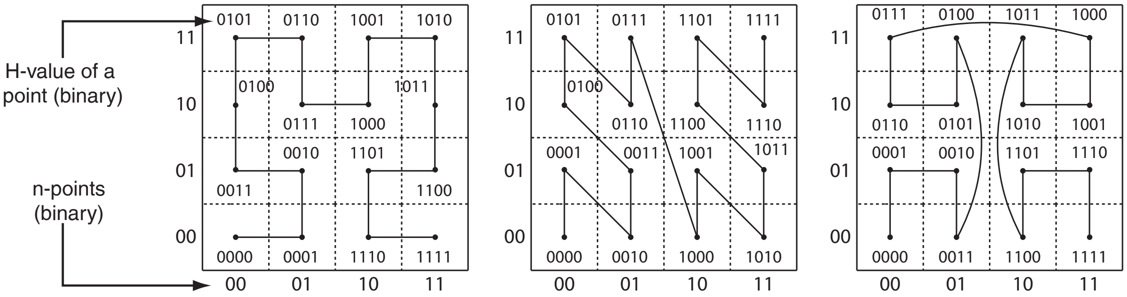

Figure 2.3 Illustration of second order Hilbert, Z-Order and Grey-Order Curve for 2 dimensions

2.2. 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 [25] distributes 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 [26], BATON [27], CAN [28], PASTRY [29], P-Grid [30] and P-ring [31] uses a distributed and scalable access structure to efficiently distribute and search data items. Chord and Pastry only supports 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.

2.3. PREFIX-GRID (PGRID) 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. PGrid 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. 2.4. To store a data item, PGrid uses a locality preserving hash function to convert the data item's identifier to a binary key κ, where $\kappa \in [0, 1[$. The 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. 2.4 is 10, therefore it stores all the data items whose keys begin with 10.

P-Grid employs a completely decentralized, parallel and distributed construction algorithm which can construct 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 atleast 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 [$path(n)$, FQDN(n)] where FQDN(n) is the fully qualified domain name of the node. Details of the construction algorithm can be found in [32].

2.3.1. Searching in P-Grid. P-Grid utilizes a simple but efficient strategy to process exact match and range queries [33]. 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. 2.4, 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.

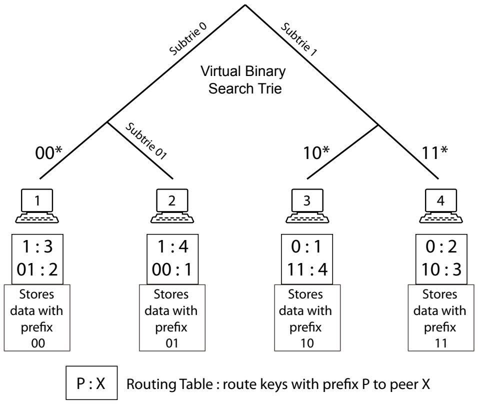
Subtrie 0
Subtrie 1
Virtual Binary Search Trie
Subtrie 01
00*
10*
11*
1
2
3
4
1 : 3
01 : 2
1 : 4
00 : 1
0 : 1
11 : 4
0 : 2
10 : 3
Stores data with prefix 00
Stores data with prefix 01
Stores data with prefix 10
Stores data with prefix 11
P : X
Routing Table : route keys with prefix P to peer X

Figure 2.4 An example P-Grid trie

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 forwarding the query to only those nodes of the subtries whose paths intersects with the query. For example, in Fig 2.4, 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 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.

2.3.2. Comparison between P-Grid and Other Overlay Networks.

Table shows the comparison between P-Grid and other popular overlay networks CHORD and CAN.

Table 2.1 Comparison between P-Grid and other p2p networks

	Min Routing	Search Method	Search Cost
Chord	Binary Tree	Equality	O(logn)
CAN	Grid	Equality	O($n^{1/d}$)
P-Grid	Binary Trie	Prefix	O(logn)

3. RELATED WORK

Query processing on large data volume has been the center of research in the computer science community since the evolution of cloud computing. This field is predominantly dominated by two classes of scalable data processing systems. The first class uses underlying key-value store to manage structured data, for example, Google's Bigtable [34], Apache HBase [11], Apache Cassandra [12], Amazon's Dynamo [10] and Yahoo's PNUTS [9]. 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 [35] and Apache's HDFS [36] to manage unstructured data. Both the systems relies on scanning the entire dataset using parallel processing approaches (for e.g. Mapreduce [8]) in order to process complex queries such as range and kNN on multidimensional data which incurs high query latency.

To address this problem, authors in [37] presents a general framework for efficient processing of multidimensional data on cloud systems. In their index 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. Authors in [38] proposes RTCAN, which builds a global index by publishing selective local R-tree indexes on C^2 overlay network. EMINC [39] is an indexing framework in which individual slave nodes builds a KD-tree [40] on its local data and a global R-tree index is build on a master node. QT-Chord [41] is an indexing framework which builds IMX-CIF Quad-tree on local data and distributes the hashed codes to the Chord overlay network. Lastly, the work in [42] proposes an in-memory indexing framework PASTIS which uses compressed bitmaps to construct partial temporal indexes. All the aforementioned schemes provides 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.

In MGrid, we combine the best of both the systems by arranging the nodes in a 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, MGrid can also efficiently process point, range and kNN queries on secondary attributes which is a key requirement for LBSs.

In this chapter, we survey the various proposed solutions in this area. Our goal is to critically examine the current state-of-the art and propose our advancements. We end this chapter by providing a comprehensive list of pros and cons of the current proposed solutions in a tabular form. .

3.1. SYSTEM ARCHITECTURE: MASTER-SLAVE VERSUS P2P BASED SYSTEM DESIGN

The current proposed solutions can be broadly classified into two categories based on the design of system architecture they employ. The first category is of approaches which employ master-slave architecture. In 2009, Xiangyu Zhang et al. proposed an indexing framework called **EMINC** [39] for cloud data processing. The global multi-dimensional index of this platform is built by first building local indices on each individual slave nodes using KD tree and then selectively publishing the set of KD tree nodes on the master nodes and maintaining them as R tree. Query processing can be done by choosing all the nodes in the cluster as candidates of the query as knowledge about data distribution on each slave node is not maintained 3.2. In the record retrieving phase, each node utilizes the local KD-tree index to get records on that node.

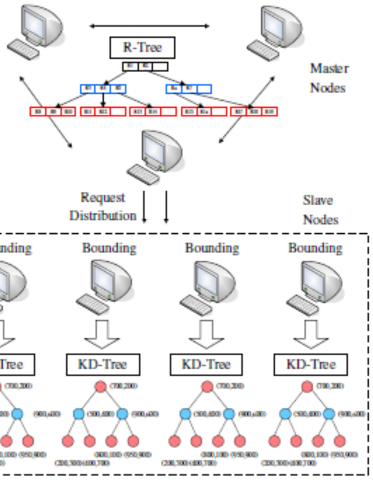

Figure 3.1 EMINC index structure consists of a R-tree in master nodes and one KD-tree on each slave node [39]

Distributing local indices on slave nodes without maintaining any meta-data leads to inefficiency of query processing which can be improved by maintaining bounding information of each dimension on each node and prune irrelevant nodes during query processing. To prune irrelevant nodes, node cube for each slave node is constructed. A node cube is a sequence of value intervals and each interval represents the value range of one indexed attribute on this node. After building a cube for each slave node, the cubes on master nodes is maintained as an R-tree. With EMINC, the authors uses bounding technique to filter unnecessary queries but it has some limitations and could be further extended using Extended EMINC (EEMINC). In EEMINC data records on one slave node will be represented by multiple node cubes.

As the slave node accumulates more and more data update operations, node cubes may need to be updated (reshaping) since the data distribution within a node cube may be sparse or uneven again. The reshaping process is similar to the process of cutting the original single cube into several small cubes by using the techniques like

Random cutting, Equal cutting and Clustered based cutting. To decide when to do the reshaping, a Cost Estimation based update strategy is employed. Experimental evaluations demonstrated that such a framework can execute point and range query efficiently. However, the framework has some limitations as the distribution of data records on slave nodes depends on the method used for cutting the original single node cube. If the method is not chosen properly, data records on a slave node will be represented by several node cubes and hence the performance of the framework will deteriorate.

Based on the similar system architecture, Shoji Nishimura et al. in 2011 proposed MD-HBase indexing scheme [43]. This approach uses the z-order space filling linearization technique to convert multi-dimensional space into a linear space. The key for the key/value data store is the z-value of the multi-dimensional data point. A global index is created of the linearized multi-dimensional data points. The design of multi-dimensional index layer is as follows. The indexing layer assumes that the underlying data storage layer stores the items sorted by their key and range-partitions the key space. The keys correspond to the z-value of the dimensions being indexed; for instance the location, user-id and timestamp. The author uses the trie-based approach (splits the space at the mid-point of a dimension, resulting in equal size splits) for space splitting. The index partitions the space into conceptual subspaces that are in-turn mapped to a physical storage abstraction called bucket. The authors proposed a novel naming scheme for subspaces to simulate a trie-based KD-tree and

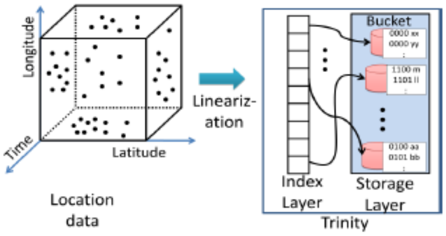

Figure 3.2 MD-HBase uses linearization technique to support multidimensional queries [43]

a Quad-tree, called as longest common prefix naming. In this naming scheme, keys which share the longest common prefix with other keys are stored in the same bucket.

To execute a point query, the key is first searched in the global index of sorted subspaces. The search for the subspace finds the entry that has the maximum prefix matched with the z-value of the query point; this entry corresponds to the highest value smaller that the z-value of the query point. After finding the corresponding bucket-id, data item is searched locally in that bucket.

For range query, the z-value of the lower bound determines the first subspace to scan. The search is continued until the subspace which corresponds to the upper bound is identified. All the subspaces which are between the lower and the upper bound are potential candidate subspaces and should be searched to get the points which satisfy the query. However, this technique has a drawback. As the number of data points increases the number of false positives also increases. Thus pruning those irrelevant subspaces incur delay in query processing.

The process of finding nearest neighbor is based upon the best first algorithm which consists of two steps, subspace search expansion and subspace scan. During subspace search expansion, the search region is incrementally expanded and then the subspaces are sorted in the region in order of the minimum distance from the queried point. The next step scans the nearest subspace that has not already been scanned and sorts points in order of the distance from the queried point.

The maximum numbers of points which can be stored in a bucket is determined by the bucket size of the underlying storage layer. Since the index layer is decoupled from the data storage layer, a subspace split when an overflow occurs in the data storage layer is handled separately. A split in the index layer relies on the first property of the prefix naming scheme which guarantees that the subspace name is a prefix of the names of any enclosed subspace. A subspace split in the index layer therefore corresponds to replacing the row corresponding to the old subspace's name with the names of the new subspaces. Experimental evaluations were conducted using open source key-value store called HBase and shows that this method is effective even while handing skewed datasets.

Both these frameworks though provides efficient query processing suffers from a major drawback. As the number of users subscribed to LBS increases, the number of concurrent requests and the amount of data to be processed increases exponentially. As the size of the global index depends on the size of the data stored locally, the master server proves to be a single point of failure and thus the master-slave architecture is inefficient for handing LBS.

To overcome the aforementioned drawbacks of a master-slave architecture, the next category of indexing frameworks employs overlay routing protocols to arrange nodes in a cluster. In 2010, Sai Wu et al. improved their previous work done in [37] and proposed CG-Index (Cloud-Global Index) [38]. In this approach, on a shared-nothing cluster, data is first partitioned along the primary key and then the partitions are distributed to nodes in the cluster which are organized in a Baton overlay structure. Each node builds a B+ tree index on its local data which facilitates search for a secondary key.

However, due to the absence of any centralized coordinator, to perform any query, the query has to be flooded on all nodes where the local search can be performed in parallel. This naive strategy is very costly and also not scalable. To overcome this drawback, the authors proposed to build a global primary index over the local B+ tree. As shown in figure 4.3, some of the B+ tree nodes (shown in red) are first published and then indexed on the cluster nodes based upon the Baton's default routing mechanism.

To process a query, the B+ tree nodes which overlaps the query are first identified in the CG-Index and then the query is processed locally on those B+ -tree nodes in parallel. The provide eventual consistency and to handle updates, the authors proposed lazy update strategy in which after a predefined time threshold, all updates are committed together on the corresponding nodes. Also, to guarantee the robustness of their index structure, replicas of both the CG-Index and B+ tree nodes are maintained in the cluster. This technique though efficient has several drawbacks. The proposed CG-Index can just index single column and thus cannot support queries referring to multiple attributes. Another problem is that B+ tree based index cannot support KNN queries due to irregular sub space shape.

In our research, we architect an indexing framework which combines the best of both these system model. Our proposed indexing framework M-Grid arranges the serving nodes in P-Grid overlay structure so that the system can be scaled to handle millions of user requests simultaneously and avoid the risk of being a bottleneck. To support queries on multiple attributes, we used h-curve based linearization technique which has the best clustering property. Furthermore, we use H-Base key-value store to demonstrate the effectiveness of our scheme.

3.2. COMPARISON MATRIX

The table 3.1 compares the three papers reviewed in the earlier section.

Table 3.1 A Review of Presented Indexing Schemes

Categories	CG-Index	EMINC	MDHBase
Supports multi-dimensional queries	No	Yes	Yes
Data Storage Layer Impacted by Approach	No	No	yes
Uses structured overlay	Yes	No	No
Users master nodes	No	Yes	No
Base indexing approach	CG-Index Global B+ -tree index for all the compute nodes in the network	local K-d tree index for each slave node	Global Index Quad or kd index of linearized data points
Scalability	increases with the increase in number of processing node	linear with the number of nodes	table per bucket and table sharing designs showed low scalability
High Throughput	No	No	Yes

4. THE MGRID INDEX FRAMEWORK

The MGrid indexing framework constitutes a federation of shared-nothing cluster of nodes leased from the cloud. Our primary goal in designing MGrid is to support LBSs by having a truly decentralized and a distributed architecture which can be scaled according to the need of the application. MGrid achieves this by adopting a simple two tiered architecture. The upper tier is based on the P-Grid 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 in our implementation) to maintain data, depending on the type of data model being used (section 4.3).

4.1. OVERVIEW

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, MGrid 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. MGrid 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.

(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 reduces from $O(\log\Pi)$ to 3 in the worst case scenario.

The resultant high-level overview of our architecture is shown in Fig. 4.1. We construct MGrid using the bottom-up approach in which, nodes are first arranged in an HBase cluster and then joins 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 multi-dimensional point and insert it according to the data models presented in section 4.3.

4.2. DATA STORAGE LAYER

MGrid 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 [11] to store the H-value of a multidimensional point which we use as the unique *rowkey*. In

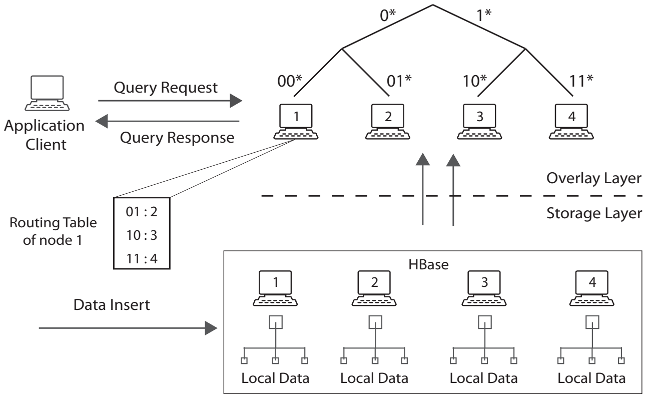

Figure 4.1 MGrid's System Architecture

this section, we describe the overview of HBase and the two data models, *Table per Node* and *Table Share*, we used to store data in MGrid.

4.2.1. Apache HBase. Apache HBase is a distributed, non-relational key-value datastore modeled after Google's BigTable [34] and built on top of HDFS [36]. 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 *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 atleast 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 auto-sharding, which means that when the size of a region exceeds a pre-defined 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 region 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.

4.3. STORAGE MODELS

4.3.1. Table per Node (TPN) Model. In this model, each node is responsible for maintaining their own separate table. When a node joins MGrid, it creates a table in HBase by the name of its own FQDN. The nodes stores the *rowkeys* locally according to the subspace they are responsible for. For example, in Fig. 2.4, 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.

4.3.2. Table Share (TS) Model. In this model, all the nodes share a single table to manage data *rowkeys*. This model allow 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.

5. QUERY PROCESSING

In this section, we present how MGrid inserts data in the network and it executes multi-dimensional point, range and kNN query processing.

5.1. DATA INSERT & 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 (Algorithm 1 & 2) efficiently insert 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 point query algorithm with respect to two storage model described in section (4.3). 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:

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

Algorithm 1 Data Insert (point p)

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

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.

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 ($v \leftarrow lookup(key, n.Table)$)
   // Table Share //
4: return ($v \leftarrow lookup(key, sharedTable)$)
```

5.2. RANGE QUERY PROCESSING

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 $min_i \leq l_i \leq u_i \leq max_i$. 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. 2.3). For example, consider the range query Q1 as shown in Fig. 5.1. Its lower bound and upper bound coordinates are A (01, 01) and F (11, 10). The equivalent H-value range of this query is ⟨0010, 1011⟩. 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 lies 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.

Our range query algorithm (Algorithm 3) is based upon the method described in [44] and [45]. 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.

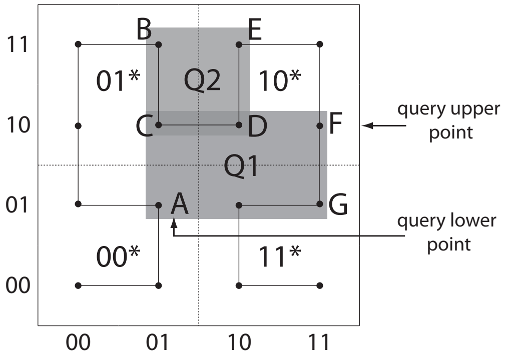

Figure 5.1 Example of a Range Query on points mapped to the Second Order Hilbert Curve in 2 dimensions

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. 5.1 is 1100. In general terms, a subspace-key is also the point where the Hilbert Curve enters in a subspace. Subspaces which intersects with the 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. 5.1.

(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 $\langle 0111, 1001\rangle$

(ii) The current-subspace-key is initially set to the subspace-key of subspace 00, ie 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, ie 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, ie the current-subspace-key is its own next-match.

(vi) The current-subspace-key is set to the subspace-key of the successor subspace, ie 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 sub-space which intersects with the current query region by using the binary search algorithm. This algorithm iteratively determines the lowest sub-space 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 sub-spaces 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 sub-spaces. To do so we use a function, h_to_c(). Solving this function using the H-values of a sub-space will give us its n-points. If the H-values of a sub-set of sub-spaces are in the following range:

[lowest,..., max-lower, min-higher,...,highest]

then all sub-spaces 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 sub-spaces 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 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:

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

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 2.2

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) ∩ (00,10) and (10,11) ∩ (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 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. 4.1, 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). Thus the overall complexity of the range querying algorithm is as O(kn) where k is the number of iterations.

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:     $R_q \leftarrow PGrid.Search(s)$
8: end for each
9: return $R_q$
```

5.3. KNN 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 [46] [47] [48] uses different distributed data structures built on decentralized P2P systems but such solutions are not scalable. [49] and [50] 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 [51]:

$$D_k \approx \frac{2\sqrt[d]{\Gamma(\frac{d}{2}+1)}}{\sqrt{\pi}}(1-\sqrt{1-\sqrt[d]{\frac{k}{N}}}) \qquad (5.1)$$

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 is 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 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
$Q_{result} \leftarrow \phi$
$\delta \leftarrow estimateRadius(k)$
$r \leftarrow \delta$
while true do
    $q_l \leftarrow q - r$
    $q_h \leftarrow q + r$
    $Q_{result} \leftarrow RangeSearch(q_l, q_h)$
    if $|Q_{result}| \geq k$ then
        return top $k$ results of $Q_{result}$
    else
        $r \leftarrow r + \delta$
    end if
end while
```

6. EXPERIMENTAL EVALUATION

We implemented MGrid 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 [52] 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 in the dataset and set the path of the nodes according to the common prefix of those values. This helped us to efficiently distribute the dataset among the nodes. 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 MGrid'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 MGrid's performance against MDHBase [43] indexing scheme's Table per Bucket (MDH-TPB) and Table Share (MDH-TS) data model*. Furthermore, we compared the performance of range queries with MapReduce.

*We could not evaluate the performance of MDHBase 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.

6.1. PERFORMANCE OF INSERT

The growing trend in LBSs are characterized by their need for scalability. We evaluated MGrid's scalability using YCSB [53] benchmarking tool. Fig. 6.1 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 less 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 (840x60) million users. Whereas, the TPN model achieved a peak throughput of approximately 660K inserts per second and can handle around 38-42 (660x60) million users. Moreover, the performance of both designs exceeds MDHBase by over 4 times and the gap becomes larger as the number of nodes increases. The reason behind MHDBase's low scalability is the cost associated with splitting the index layer which blocks other operations until its completion. In MGrid, 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.

6.2. PERFORMANCE OF POINT & RANGE QUERY

Multidimensional point and range queries are the most frequent queries in LBSs. MGrid processes the point query by directly querying the HBase table. On

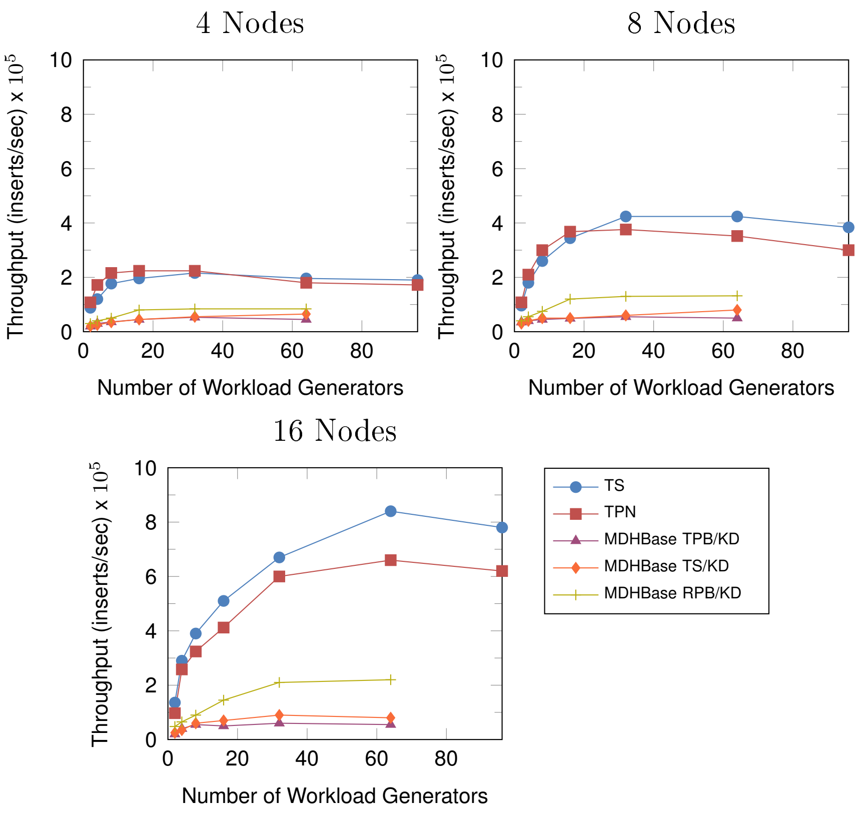

Figure 6.1 Performance of Insert Throughput as a Function of Load on the System

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. Fig. 6.2 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.

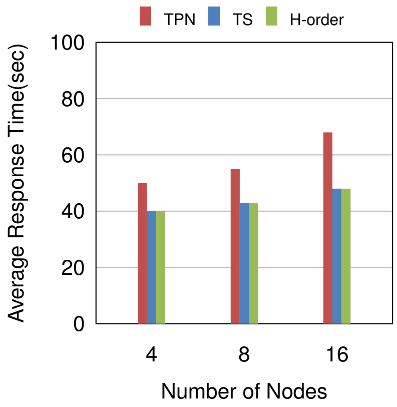

Figure 6.2 Performance of Point Query (D=3)

Fig. 6.3 and 6.4 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. 6.3). 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. 6.8, 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 Fig. 6.3 and 6.4 we perform the set of experiments done for 2-d dataset on a 3-d dataset. Fig. 6.4 shows the performance

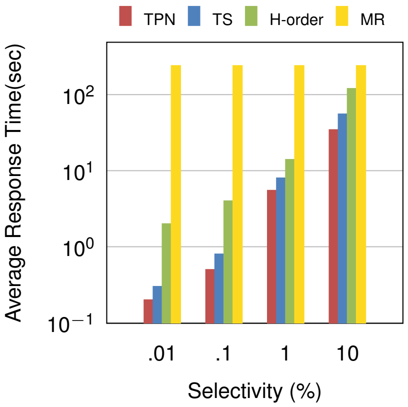

Figure 6.3 Performance of Range Query (Nodes=4, D=2)

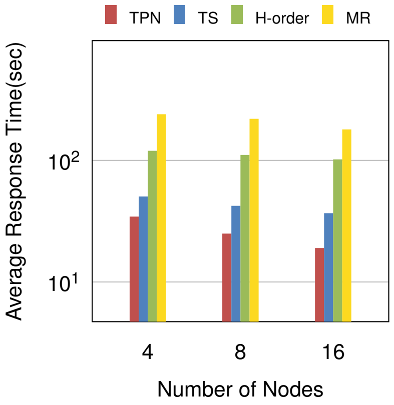

Figure 6.4 Performance of Range Query (Selectivity=10%, D=2)

of 3-d range query as a 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 MDHBase'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 MDHBase 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.

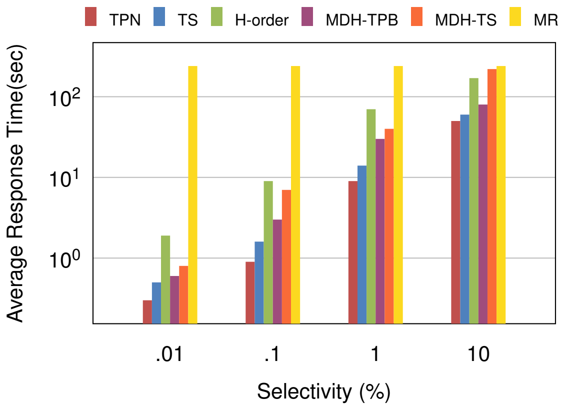

Figure 6.5 Performance of Range Query (Nodes=4, D=3)

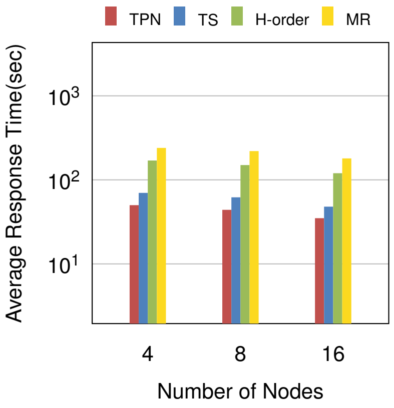

Figure 6.6 Performance of Range Query (Selectivity=10%, D=3)

6.3. PERFORMANCE OF KNN QUERY

MGrid processes the k nearest neighbor (kNN) query iteratively. We first estimate the distance between the query point and its k^{th} nearest neighbor using (5.1), 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. Fig. 6.7 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 queries with larger search space is processed using more nodes. Our obtained results are validated in Fig. 6.8, 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 Fig. 6.9 and 6.10 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 MDHBase and H-order designs in Fig 6.9. The average response time of our models increase with the increase in value of parameter k, which validates the results depicted in Fig. 6.7. However, 3-d kNN queries take more time to process as compared to 2-d since the complexity of performing range queries increases with

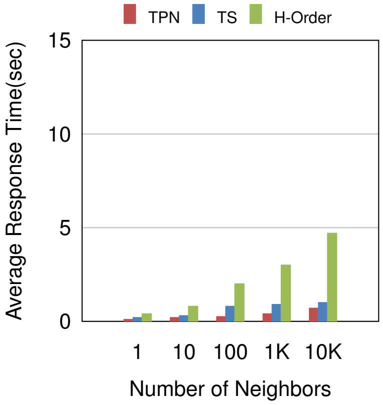

Figure 6.7 Performance of kNN Query (Nodes=4, D=2)

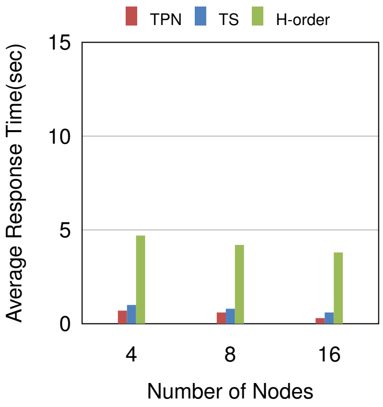

Figure 6.8 Performance of *k*NN Query (k=10K, D=2)

number of dimensions. For k=1, the TPN and TS models gives superior performance with an average response time of approximately 500ms and 700ms respectively, in contrast to H-order, MDHBase TPB and TS models being approximately 800ms, 2000ms amd 3000ms 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 *k*NN 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. 6.10, we compare the effect of varying the number of nodes on the performance of 3-d *k*NN 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 6.8). We expect the performance of our designs for *k*NN processing to be better on uniform dataset as the equation 5.1 provides more accurate estimation of initial search range for uniform dataset. Thus, the *k*NN processing will require less number of range search iterations to retrieve k results.

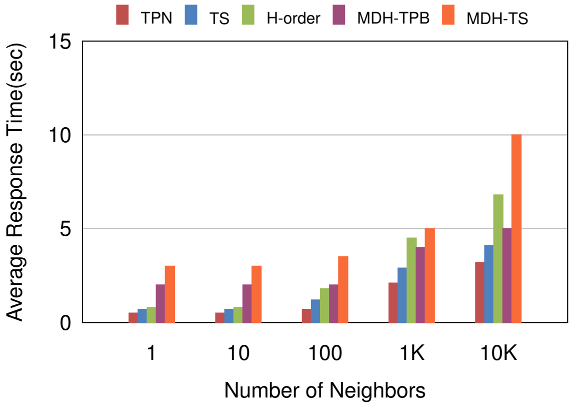

Figure 6.9 Performance of kNN Query (Nodes=4, D=3)

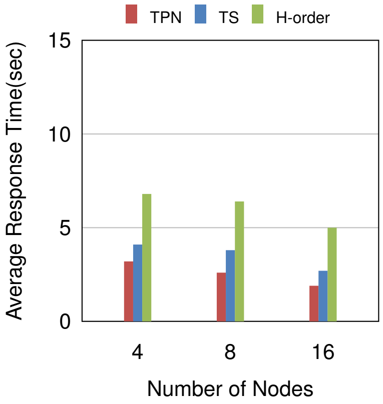

Figure 6.10 Performance of kNN Query (k=10K, D=3)

7. CONCLUSIONS

In this thesis we present and evaluate MGrid, a multidimensional indexing framework for location aware services on cloud platform. MGrid is a scalable, completely decentralized and platform independent indexing framework which can efficiently process point, range and nearest neighbor queries. MGrid first arranges the nodes leased from cloud in a P-Grid overlay network which virtually partitions the whole space in a binary tri-structure. Next, for efficient storage and retrieval of multidimensional data, we exploit 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 develop 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 MGrid achieves almost four times better performance than its previous counterpart. In future we wish to extend our framework by providing it the ability to create different index structures on-the-fly based upon users choice and to support wider variety of queries including skyline and spatial-joins.

BIBLIOGRAPHY

[1] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011.

[2] Gavin Mcardle, Andrea Ballatore, Ali Tahir, and Michela Bertolotto. An open-source web architecture for adaptive location-based services, 2006.

[3] Agnes Voisard Jochen Schiller. *Location-Based Services - The Morgan Kaufmann Series in Data Management Systems*. Morgan Kaufmann, 2014.

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

[5] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, 1970.

[6] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, 2002.

[7] Daniel J. Abadi. Data management in the cloud: Limitation and opportunities. Technical report, 2009.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, pages 107–113, 2008.

[9] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, pages 1277–1288, 2008.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, 2007.

[11] http://hbase.apache.org/. [Last Accessed 2014-08-20].

[12] http://cassandra.apache.org/. [Last Accessed 2014-08-20].

[13] D Hilbert. Ueber stetige abbildung einer linie auf ein flashenstuck. *Mathematishe annalen*, 32:459–460, 1893.

[14] Fei Li, Rongguo Chen, Chenghu Zhou, and Mingbo Zhang. A novel geo-spatial image storage method based on hilbert space filling curves. In *Geoinformatics, 2010 18th International Conference on*, pages 1–4, 2010.

[15] M. Pavanakumar and K.N. Kaushik. Revisiting the space-filling curves for storage, reordering and partitioning mesh based data in scientific computing. In *High Performance Computing (HiPC), 2013 20th International Conference on*, pages 362–367, 2013.

[16] Chunyang Hu, Yongwang Zhao, Xin Wei, B. Du, Yonggang Huang, Dianfu Ma, and Xuan Li. Actgis: A web-based collaborative tiled geospatial image map system. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 521–528, 2010.

[17] A. R. Butz. Alternative algorithm for hilbert's space-filling curve. *IEEE Trans. Comput.*, 20:424–426, 1971.

[18] T. Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *Information Theory, IEEE Transactions on*, 15(6):658–664, 1969.

[19] C.H. Hamilton and A Rau-Chaplin. Compact hilbert indices for multi-dimensional data. In *Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007. First International Conference on*, pages 139–146, 2007.

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

[21] F. Gray. Pulse code communication. 1953.

[22] B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering, IEEE Transactions on*, 13:124–141, Jan 2001.

[23] DAVID J. ABEL and DAVID M. MARK. A comparative analysis of some two-dimensional orderings. *International journal of geographical information systems*, 4:21–31, 1990.

[24] Mohamed F. Mokbel, Walid G. Aref, and Ibrahim Kamel. Performance of multi-dimensional space-filling curves. In *Proceedings of the 10th ACM International Symposium on Advances in Geographic Information Systems*, GIS '02, pages 149–154, 2002.

[25] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, pages 46–66, 2001.

[26] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 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*, SIGCOMM '01, pages 149–160, 2001.

[27] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 661–672, 2005.

[28] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31:161–172, 2001.

[29] Antony I. T. Rowstron and Peter Druschel. 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*, Middleware '01, pages 329–350, 2001.

[30] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-grid: A self-organizing structured p2p system. *SIGMOD Rec.*, 32:29–33, 2003.

[31] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. P-ring: An efficient and robust p2p range index structure. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 223–234, 2007.

[32] Karl Aberer, Anwitaman Datta, Manfred Hauswirth, and Roman Schmidt. Indexing data-oriented overlay networks. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 685–696, 2005.

[33] Anwitaman Datta, Manfred Hauswirth, Renault John, Roman Schmidt, and Karl Aberer. Range queries in trie-structured overlays. In *Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*, P2P '05, pages 57–66, 2005.

[34] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, 2006.

[35] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, 2003.

[36] http://hadoop.apache.org/. [Last Accessed 2014-08-20].

[37] Sai Wu and Kun-Lung Wu. An indexing framework for efficient retrieval on the cloud. *IEEE Data Eng. Bull.*, 32(1):75–82, 2009.

[38] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multi-dimensional data in a cloud system. In *Proceedings of the 2010 ACM*

SIGMOD International Conference on Management of Data, SIGMOD '10, pages 591–602, 2010.

[39] Xiangyu Zhang, Jing Ai, Zhongyuan Wang, Jiaheng Lu, and Xiaofeng Meng. An efficient multi-dimensional index for cloud data management. In *Proceedings of the First International Workshop on Cloud Data Management*, CloudDB '09, pages 17–24, 2009.

[40] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[41] Linlin Ding, Baiyou Qiao, Guoren Wang, and Chen Chen. An efficient quad-tree based index structure for cloud data management. In *Web-Age Information Management*, volume 6897 of *Lecture Notes in Computer Science*, pages 238–250. 2011.

[42] Rolando Blanco Suprio Ray and Anil K. Goel. Supporting location-based services in a main-memory database. *Proceedings of the IEEE International Conference on Mobile Data Management (MDM)*, 2014.

[43] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *Proceedings of the 2011 IEEE 12th International Conference on Mobile Data Management - Volume 01*, MDM '11, pages 7–16. IEEE Computer Society, 2011.

[44] J. K. Lawder. Querying multi-dimensional data indexed using the hilbert space-filling curve. *SIGMOD Record*, 30:2001, 2001.

[45] https://code.google.com/p/uzaygezen/. [Last Accessed 2014-08-20].

[46] Yuzhe Tang, Jianliang Xu, Shuigeng Zhou, Wang-Chien Lee, Dingxiong Deng, and Yue Wang. A lightweight multidimensional index for complex queries over dhts. *IEEE Trans. Parallel Distrib. Syst.*, 22:2046–2054, 2011.

[47] Egemen Tanin, Deepa Nayar, and Hanan Samet. An efficient nearest neighbor algorithm for p2p settings. In *Proceedings of the 2005 National Conference on Digital Government Research*, dg.o '05, pages 21–28, 2005.

[48] Jun Gao. Efficient support for similarity searches in dht-based peer-to-peer systems. In *In IEEE International Conference on Communications (ICCâĂŹ07*, 2007.

[49] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proc. VLDB Endow.*, 5:1016–1027, 2012.

[50] Aleksandar Stupar, Sebastian Michel, and Ralf Schenkel. Rankreduce - processing k-nearest neighbor queries on top of mapreduce. In *In LSDS-IR*, 2010.

[51] Yufei Tao, Jun Zhang, Dimitris Papadias, and Nikos Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE Trans. on Knowl. and Data Eng.*, 16:1169–1184, 2004.

[52] Thomas Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6:153–180, 2002.

[53] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, 2010.

VITA

Shashank Kumar was born in Moradabad, India in 1986. His initial schooling took place in four different schools. He earned his bachelorâĂŹs degree in Computer Science & Engineering from Uttar Pradesh Technical University at Lucknow in 2009. After his graduation, he started his own company, Webvity which specialized in web development and headed numerous prestigious projects for Government of India.

Shashank came to the Missouri University of Science and Technology in 2011, where he earned his Masters of Science in Computer Science, in Aug, 2014. While there, he worked as a research assistant with Dr. Sanjay Madria, focusing on cloud database management and query optimization.