Distrib Parallel Databases (2013) 31:289–319
DOI 10.1007/s10619-012-7109-z

$\mathcal{MD}$-HBase: design and implementation of an elastic data infrastructure for cloud-scale location services

Shoji Nishimura · Sudipto Das · Divyakant Agrawal · Amr El Abbadi

Published online: 5 September 2012

Abstract The ubiquity of location enabled devices has resulted in a wide proliferation of location based applications and services. To handle the growing scale, database management systems driving such location based services (LBS) must cope with high insert rates for location updates of millions of devices, while supporting efficient real-time analysis on latest location. Traditional DBMSs, equipped with multi-dimensional index structures, can efficiently handle spatio-temporal data. However, popular open-source relational database systems are overwhelmed by the high insertion rates, real-time querying requirements, and terabytes of data that these systems must handle. On the other hand, key-value stores can effectively support large scale operation, but do not natively provide multi-attribute accesses needed to support the rich querying functionality essential for the LBSs.

We present the design and implementation of *$\mathcal{MD}$-HBase*, a scalable data management infrastructure for LBSs that bridges this gap between scale and functionality.

Communicated by Dipanjan Chakraborty.

S. Nishimura
Cloud System Research Laboratories, NEC Corporation, Kawasaki, Kanagawa 211-8666, Japan
e-mail: s-nishimura@bk.jp.nec.com

S. Das (✉) · D. Agrawal · A. El Abbadi
Dept. of Computer Science, University of California at Santa Barbara, Santa Barbara, CA, 93106-5110, USA
e-mail: sudipto@cs.ucsb.edu

D. Agrawal
e-mail: agrawal@cs.ucsb.edu

A. El Abbadi
e-mail: amr@cs.ucsb.edu

Present address:
S. Das
Microsoft Research, Redmond, WA, 98052, USA
e-mail: sudiptod@microsoft.com

Our approach leverages a multi-dimensional index structure layered over a key-value store. The underlying key-value store allows the system to sustain high insert throughput and large data volumes, while ensuring fault-tolerance, and high availability. On the other hand, the index layer allows efficient multi-dimensional query processing. Our optimized query processing technique accesses only the index and storage level entries that intersect with the query region, thus ensuring efficient query processing. We present the design of $\mathcal{MD}$-HBase that demonstrates how two standard index structures—the K-d tree and the Quad tree—can be layered over a range partitioned key-value store to provide scalable multi-dimensional data infrastructure. Our prototype implementation using HBase, a standard open-source key-value store, can handle hundreds of thousands of inserts per second using a modest 16 node cluster, while efficiently processing multi-dimensional range queries and nearest neighbor queries in real-time with response times as low as few hundreds of milliseconds.

Keywords Location based services · Key-value stores · Multi-dimensional data · Real time analysis

1 Introduction

The last few years have witnessed a significant increase in hand-held devices becoming location aware with the potential to continuously report up-to-date location information of their users. This has led to a large number of location based services (*LBS*) which customize a user's experience based on location. Some applications—such as customized recommendations and advertisements based on a user's current location and history—have immediate economic incentives, while some other applications—such as location based social networking or location aware gaming—enrich the user's experience in general. With major wireless providers serving hundreds of millions of subscribers [19], millions of devices registering their location updates continuously is quite common. Database management systems (DBMS) driving these location based services must therefore handle millions of location updates per minute while answering near real time analysis and statistical queries that drive the different recommendation and personalization services.

Location data is inherently multi-dimensional, minimally including a user id, a latitude, a longitude, and a time stamp. A rich literature of multi-dimensional indexing techniques—for instance, K-d trees [1], Quad trees [7] and R-trees [8]—have empowered relational databases (RDBMS) to efficiently process multi-dimensional data. However, the major challenge posed by these location based services is in scaling the systems to sustain the high throughput of location updates and analyzing huge volumes of data to glean intelligence. For instance, if we consider only the insert throughput, a MySQL installation running on a commodity server becomes a bottleneck at loads of tens of thousands of inserts per second; performance is further impacted adversely when answering queries concurrently. Advanced designs, such as staging the database, defining views, database and application partitioning to scale out, a distributed cache, or moving to commercial systems, will increase the throughput; however, such optimizations are expensive to design, deploy, and maintain.

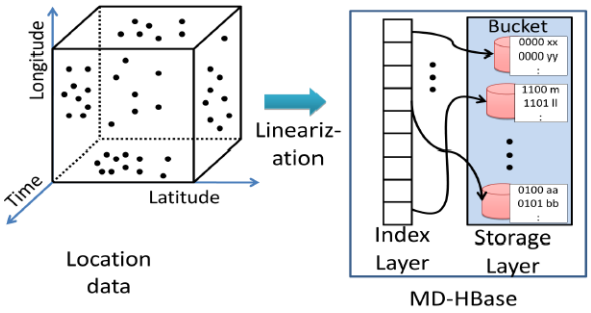


Fig. 1 Architecture of $\mathcal{MD}$-HBase

On the other hand, key-value stores, both in-house systems such as Bigtable [3] and their open source counterparts like HBase [10], have proven to scale to millions of updates while being fault-tolerant and highly available. However, key-value stores do not natively support efficient multi-attribute access, a key requirement for the rich functionality needed to support LBSs. In the absence of any filtering mechanism for secondary attribute accesses, such queries resort to full scan of the entire data. MapReduce [5] style processing is therefore a commonly used approach for analysis on key-value stores. Even though the MapReduce framework provides abundant parallelism, a full scan is wasteful, especially when the selectivity of the queries is high. Moreover, many applications require near real-time query processing based on a user's current location. Therefore, query results based on a user's stale location is often useless. As a result, a design for batched query processing on data periodically imported into a data warehouse is inappropriate for the real-time analysis requirement.

RDBMSs provide rich querying support for multi-dimensional data but are not scalable, while key-value stores can scale but cannot handle multi-dimensional data efficiently. Our solution, called *$\mathcal{MD}$-HBase*, bridges this gap by layering a multi-dimensional index over a key-value store to leverage the best of both worlds.[1] We use linearization techniques such as Z-ordering [13] to transform multi-dimensional location information into a one dimensional space and use a range partitioned key-value store (HBase [10] in our implementation) as the storage back end. Figure 1 illustrates $\mathcal{MD}$-HBase's architecture showing the index layer and the data storage layer. We show how this design allows standard and proven multi-dimensional index structures, such as K-d trees and Quad trees, to be layered on top of the key-value stores with minimal changes to the underlying store and negligible effect on the operation of the key-value store. The underlying key-value store provides the ability to sustain a high insert throughput and large data volumes, while ensuring fault-tolerance and high availability. The overlaid index layer allows efficient real-time processing of multi-dimensional range and nearest neighbor queries that comprise the basic data analysis primitives for location based applications. The index layer performs first-level filtering by obviating scanning data buckets that do not contain any data points that match the query. We further minimize the query processing time by using a technique to

[1] The name $\mathcal{MD}$-HBase signifies adding multi-dimensional data processing capabilities to HBase, a range partitioned key-value store.

recursively sub-divide multi-dimensional range queries so that only the index entries that intersect with the query region are scanned, thus eliminating any false positive scans in the index layer. This elimination of false positives is particularly significant for queries where one or more dimensions have long intervals or are unbounded. Since linearization loosely preserves locality, linearized lower and upper bounds of such queries typically encapsulate large numbers of sub-spaces that do not intersect with the query region.

We evaluate different implementations of the data storage layer in the key-value store and evaluate the trade-offs associated with these different implementations. In our experiments, $\mathcal{MD}$-HBase achieved more than 200 K inserts per second on a modest cluster spanning 16 nodes, while supporting real-time range and nearest neighbor queries with response times less than one second. Assuming devices reporting one location update per minute, this small cluster can handle updates from 10–15 million devices while providing between one to two orders of magnitude improvement over a MapReduce or Z-ordering based implementation for query processing. Moreover, our design does not introduce any scalability bottlenecks, thus allowing the implementation to scale with the underlying key-value data store. Nishimura et al. [14] presented a preliminary design and implementation of $\mathcal{MD}$-HBase. This paper extends Nishimura et al. [14] by presenting an optimized query processing technique with no false positive scans in the index layer, additional implementation details of the index and storage layer, and a detailed evaluation demonstrating the benefits of the optimized query processing technique.

Contributions This paper makes the following significant contributions:

- We propose the design of $\mathcal{MD}$-HBase that uses linearization to implement a scalable multi-dimensional index structure layered over a range-partitioned key-value store.
- We demonstrate how this design can be used to implement a K-d tree and a Quad tree, two standard multi-dimensional index structures.
- We propose an optimization to eliminate any false positive scans in the index layer by sub-dividing the input query recursively into sub-queries exactly as the space is partitioned by the index structure.
- We present three alternative implementations of the storage layer in the key-value store and evaluate the tradeoffs associated with each implementation.
- We provide a detailed evaluation of our prototype using synthetically generated location data and analyze $\mathcal{MD}$-HBase's scalability and efficiency.

Organization Section 2 provides background on location based applications, linearization techniques, and multi-dimensional index structures. Section 3 describes the design and implementation of the index layer. Section 4 presents an optimized query processing technique with no false positives. Section 5 describes the implementation of the data storage layer. Section 6 presents a detailed evaluation of $\mathcal{MD}$-HBase by comparing the different design choices. Section 7 surveys the related literature and Sect. 8 concludes the paper.

2 Background

2.1 Location based applications

Location data points are multi-dimensional with spatial attributes (e.g., longitude and latitude), a temporal attribute (e.g., timestamp), and an entity attribute (e.g., user's ID). Different applications use this information in a variety of ways. We provide two simple examples that illustrate the use of location data for providing location aware services.

Application 1 Location based advertisements and coupon distribution: Consider a restaurant chain, such as McDonalds, running a promotional discount for the next hour to advertise a new snack and wants to disseminate coupons to attract customers who are currently near any of their restaurants spread throughout the country. An LBS provider issues multi-dimensional range queries to determine all users within 5 miles from any restaurant in the chain and delivers a coupon to their respective devices. Another approach to run a similar campaign with a limited budget is to limit the coupons to only the 100 users nearest to a restaurant location. In this case, the LBS provider issues nearest neighbor queries to determine the users. In either case, considering a countrywide (or worldwide) presence of this restaurant chain, the amount of data analyzed by such queries is huge.

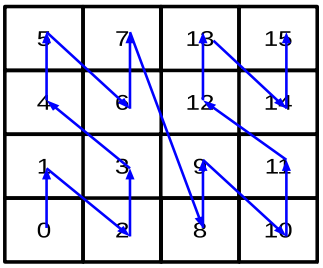
Application 2 Location based social applications: Consider a social application that notifies a user of his/her friends who are currently nearby. The LBS provider in this case issues a range query around the user's current location and intersects the user's friend list with the results. Again, considering the scale of current social applications, an LBS provider must handle data for tens to hundreds of millions of its users to answer these queries for its users spread across a country.

Location information has two important characteristics. First, it is inherently skewed, both spatially and temporally. For instance, urban regions are more dense compared to rural regions, while business and school districts are dense during weekdays, while residential areas are dense during the night and weekends. Second, the time dimension is potentially unbounded and monotonically increasing. We later discuss how these characteristics influence many of the design choices.

2.2 Linearization techniques

Linearization [16] is a method to transform multi-dimensional data points to a single dimension and is a key aspect of the index layer in $\mathcal{MD}$-HBase. Linearization allows leveraging a single-dimensional database (a key-value store in our case) for efficient multi-dimensional query processing. A space-filling curve [16] is one of the most popular approaches for linearization. A space filling curve visits all points in the multi-dimensional space in a systematic order. Z-ordering [13] is an example of a space filling curve. Figure 2 illustrates a z-order in a 2D space, transforming the space into a 1D order as depicted by the numbers in the blocks. Z-ordering loosely

Fig. 2 Z-ordering



preserves the locality of data-points in the multi-dimensional space and is also easy to implement.

Linearization alone is however not enough for efficient query processing; a linearization based multi-dimensional index layer, though simple in design, results in inefficient query processing. For instance, a range query on a linearized system is decomposed into several linearized sub-queries; however, a trade-off exists between the number of sub-queries and the number of false-positives. A reduction in the number of linearized sub-queries results in an increase in the overhead due to the large number of false-positives. On the other hand, eliminating false-positives results in a significant growth in the number of sub-queries. Furthermore, such a naïve technique is not robust to skew inherent in many real life applications.

2.3 Multi-dimensional index structures

The Quad tree [7] and the K-d tree [1] are two of the most popular multi-dimensional indexing structures. They split the multi-dimensional space recursively into *subspaces* in a systematic manner and organize these subspaces as a search tree. A Quad tree divides the n-dimensional space into 2^n subspaces along all dimensions whereas a K-d tree alternates the splitting of the dimensions. Each subspace has a maximum limit on the number of data points in it, beyond which the subspace is split. Two approaches are commonly used to split a subspace: a trie-based approach and a point-based approach [16]. The trie-based approach splits the space at the mid-point of a dimension, resulting in equal size splits; while the point-based technique splits the space by the median of data points, resulting in subspaces with equal number of data points. The trie-based approach is efficient to implement as it results in regular shaped subspaces. On the other hand, the point based approach is more robust to skew.

In addition to the performance issues, trie-based Quad trees and K-d trees have a property that allows them to be coupled with Z-ordering. A trie-based split of a Quad tree or a K-d tree results in subspaces where all z-values in any subspace are continuous. Figure 3 provides an illustration using a K-d tree for two dimensions; an example using a Quad tree is very similar. Different shades denote different subspaces and the dashed arrows denote the z-order traversal. As is evident, in any of the subspaces the Z-values are continuous. This observation forms the basis of the indexing layer of $\mathcal{MD}$-HBase. Compared to a naïve linearization based index structure or a B+-Tree index built on linearized values, K-d and Quad trees capture data distribution statistics. They partition the target space based on the data distribution

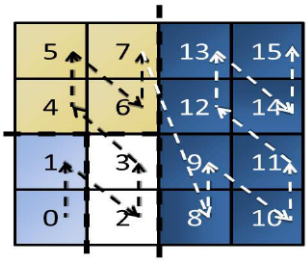


Fig. 3 Space splitting in a K-d tree

and limit the number of data points in each subspace, resulting in more splits in hot regions. Furthermore, Quad trees and KD trees maintain the boundaries of sub-spaces in the original space. This allows efficient pruning of the space as well as reducing the number of false positive scans during query processing and therefore is more robust to skew. Moreover, when executing the best-first algorithm for kNN queries, the fastest practical algorithm, a B+ tree based index cannot be used due to the irregular sub-space shape. $\mathcal{MD}$-HBase therefore uses these multi-dimensional index structures instead of a simple single-dimension index based on linearization.

3 Multi-dimensional index layer

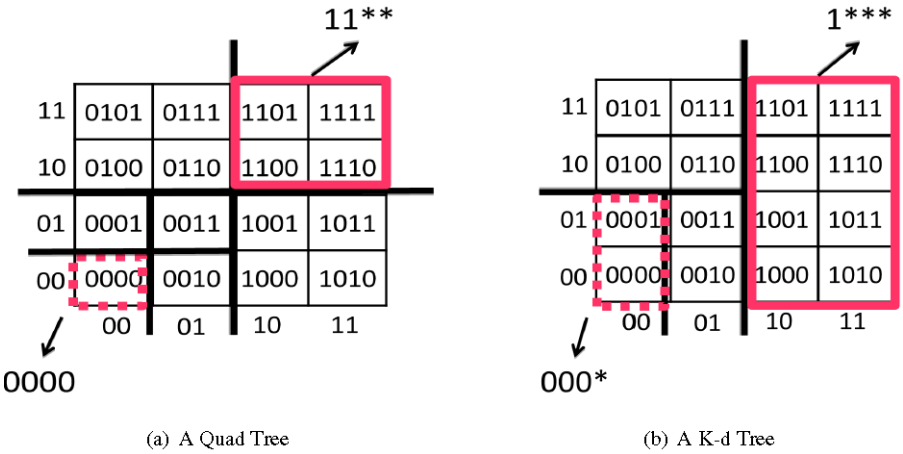
We now present the design of the multi-dimensional index layer in $\mathcal{MD}$-HBase. Specifically, we show how standard index structures like K-d trees [1] and Quad trees [7] can be adapted to be layered on top of a key-value store. The indexing layer assumes that the underlying data storage layer 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 and timestamp. We use the trie-based approach for space splitting. The index partitions the space into conceptual *subspaces* that are in-turn mapped to a physical storage abstraction called *bucket*. The mapping between a conceptual subspace in the index layer and a bucket in the data storage layer can be one-to-one, many-to-one, or many-to-many depending on the implementation and requirements of the application. We develop a novel naming scheme for subspaces to simulate a trie-based K-d tree and a Quad tree. This naming scheme, called *longest common prefix naming*, has two important properties critical to efficient index maintenance and query processing; a description of the naming scheme and its properties are discussed below.

3.1 Longest common prefix naming scheme

If the multi-dimensional space is divided into equal sized subspaces and each dimension is enumerated using binary values, then the z-order of a given subspace is given by interleaving the bits from the different dimensions. Figure 4 illustrates this property. For example, the Z-value of the sub-space (00, 11) is represented as 0101. We name each subspace by the longest common prefix of the z-values of points contained in the subspace. Figure 5 provides an example for partitioning the space in a trie-based Quad tree and K-d tree. For example, consider a Quad tree built on the 2D

Fig. 4 Binary Z-ordering

11	0101	0111	1101	1111
10	0100	0110	1100	1110
01	0001	0011	1001	1011
00	0000	0010	1000	1010
	00	01	10	11



(a) A Quad Tree (b) A K-d Tree

Fig. 5 The longest common prefix naming scheme

space. The subspace at the top right of Fig. 5(a), enclosed by a thick solid line, consists of z-values 1100, 1101, 1110, and 1111 with a longest common prefix of 11**; the subspace is therefore named 11**. Similarly, the lower left subspace, enclosed by a broken line, only contains 0000, and is named 0000. Now consider the example of a K-d tree in Fig. 5(b). The subspace in the right half, enclosed by a solid line, is represented by the longest common prefix 1, while the subspace in the bottom left consisting of 0000 and 0001 is named as 000*. This naming scheme is similar to that of Prefix Hash Tree (PHT) [15], a variant of a distributed hash table.[2]

$\mathcal{MD}$-HBase leverages two important properties of this longest common prefix naming scheme. First, if subspace A encloses subspace B, the name of subspace A is a prefix of that of subspace B. For example, in Fig. 5(a), a subspace which contains 0000–0011 encloses a subspace which contains only 0000. The former subspace's name, 00**, is a prefix of the latter's name 0000. Therefore, on a split, names of the new subspaces can be derived from the original subspace name by appending bits depending on which dimensions were split.

Second, the subspace name is enough to determine the region boundaries on all dimensions. This property derives from the fact that the z-values are computed by in-

[2] http://en.wikipedia.org/wiki/Distributed_hash_table.

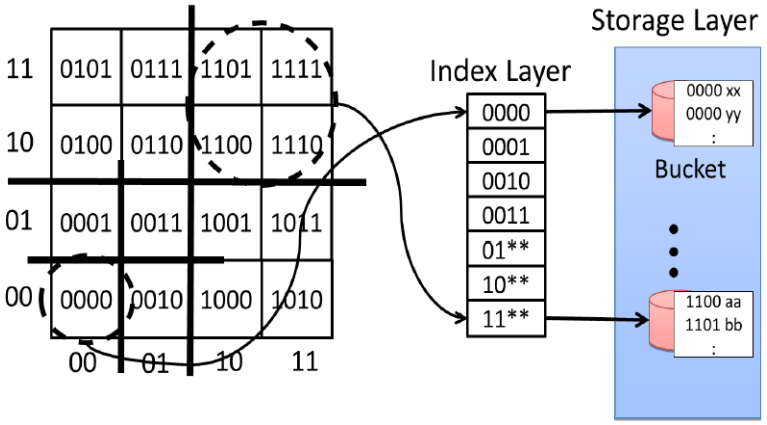


Fig. 6 Index layer mapping a trie-based Quad tree

terleaving bits from the different dimensions and thus improves the pruning power of a range query. Determining the range boundary consists of the following two steps: (i) given the name, extract the bits corresponding to each dimension; and (ii) complete the extracted bits by appending 0 s for the lower bound and 1s for the upper bound values, both bound values being inclusive. For instance, let us consider the Quad tree in Fig. 5(a). The subspace at the top right enclosed by a solid lined box is named 11. Since this is a 2D space, the prefix for both the vertical and horizontal dimensions is 1. Therefore, using the rule for extension, the range for both dimensions is [10, 11]. Generalizing to the n-dimensional space, a Quad tree splits a space on all n dimensions resulting in 2^n subspaces. Therefore, the number of bits in the name of a subspace will be an exact multiple of the number of dimensions. This ensures that reconstructing the dimension values in step (i) will provide values for all dimensions. But since a K-d tree splits on alternate dimensions, the number of bits in the name is not guaranteed to be a multiple of the number of dimensions, in which case the prefix for different dimensions have different lengths. Considering the K-d tree example of Fig. 5(b), for the lower left subspace named 000$*$, the prefix for the horizontal dimension is 00 while that of the vertical dimension is 0, resulting in the bounds for the horizontal dimension as [00, 00] and that of the vertical dimension as [00, 01].

3.2 Index layer

The index layer leverages the longest common prefix naming scheme to map multi-dimensional index structures into a single dimensional substrate. Figure 6 illustrates the mapping of a Quad tree partitioned space to the index layer; the mapping technique for K-d trees follows similarly. We now describe how the index layer can be used to efficiently implement some common operations and then discuss the space splitting algorithm for index maintenance.

Algorithm 1 Subspace Lookup

1: /* $\langle q \rangle$ be the query point. */
2: $Z_q \leftarrow ComputeZ\text{-}value(q)$
3: $Bkt_q \leftarrow PrefixMatchingBinarySearch(Z_p)$

Algorithm 2 Insert a new location data point

1: /* $\langle p \rangle$ be the new data point. */
2: $Bkt_p \leftarrow LookupSubspace(p)$
3: $InsertToBucket(Bkt_p, p)$
4: **if** $(Size(Bkt_p) > MaxBucketSize)$ **then**
5: $\quad SplitSpace(Bkt_p)$

3.2.1 Subspace lookup and point queries

Determining the subspace to which a point belongs forms the basis for point queries as well as for data insertion. Since the index layer comprises a sorted list of subspace names, determining the subspace to which a point belongs is efficient. Recall that the subspace name determines the bounds of the region that the subspace encloses. 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. A prefix matching binary search, which substitutes exact matching comparison to prefix matching comparison for the termination condition of the binary search algorithm, is therefore sufficient. Algorithm 1 provides the algorithm for subspace lookup. To answer a point query, we first lookup the subspace corresponding to the z-value of the point. The point is then queried in the bucket to which the subspace maps.

3.2.2 Insertion

The algorithm to insert a new data point (shown in Algorithm 2) is similar to the point query algorithm. It first looks up the bucket corresponding to the subspace to which the point belongs, and then inserts the data point in the bucket. Since there is a maximum limit to the number of points a bucket can hold, the insertion algorithm checks the current size of the bucket to determine if a split is needed. The technique to split a subspace is explained in Sect. 3.2.5.

3.2.3 Range query

A multi-dimensional range query is one of the most frequent queries for location based applications. Algorithm 3 provides the pseudo code for range query processing. Let $\langle q_l, q_h \rangle$ be the range for the query, q_l is the lower bound and q_h is the upper bound. The z-value of the lower bound determines the first subspace to scan. All sub-sequent subspaces until the one corresponding to the upper bound are potential candidate subspaces. Let $\mathbb{S}_q$ denote the set of candidate subspaces. Since the z-order loosely preserves locality, some subspaces in $\mathbb{S}_q$ might not be part of the range. For

Algorithm 3 Range query

```
1: /* ⟨q_l, q_h⟩ be the range for the query. */
2: Z_low ← ComputeZ-value(q_l)
3: Z_high ← ComputeZ-value(q_h)
4: 𝕊_q ← {Z_low ≤ SubspaceName ≤ Z_high}
5: ℝ_q ← φ /* Initialize result set to empty set. */
6: for each Subspace S ∈ 𝕊_q do
7:    ⟨S_l, S_h⟩ ← ComputeBounds(S)
8:    if (⟨S_l, S_h⟩ ⊆ ⟨q_l, q_h⟩) then
9:       ℝ_q ∪ ScanBucketForSpace(S)
10:   else if (⟨S_l, S_h⟩ ∩ ⟨q_l, q_h⟩) then
11:      for each point p ∈ S do
12:         if (p ∈ ⟨q_l, q_h⟩) then
13:            ℝ_q ∪ p
14: return ℝ_q
```

example, consider the Quad tree split shown in Fig. 6. Consider the range query $\langle[01, 10], [11, 11]\rangle$. The z-value range for this query is [0110, 1111] which results in $\mathbb{S}_q$ equal to $\{01^{**}, 10^{**}, 11^{**}\}$. Since the query corresponds to only the top half of the space, the subspace named 10^{**} is a false positive. But such false positives are eliminated by a scan of the index. As the subspace name only is enough to determine the boundary of the region enclosed by the subspace, points in a subspace are scanned only if the range of the subspace intersects with the query range. This check is inexpensive and prunes out all the subspaces that are not relevant. For subspaces that are contained in the query, all points will be returned, while subspaces that only intersect with the query require further filtering. The steps for query processing are shown in Algorithm 3.

3.2.4 Nearest neighbor query

Nearest neighbor queries are also an important primitive operation for many location based applications. Algorithm 4 shows the steps for k nearest neighbors query processing in $\mathcal{MD}$-HBase. The algorithm is based on the best-first algorithm where the subspaces are scanned in order of the distance from the query point [16].

The algorithm consists of two steps: *subspace search expansion* and *subspace scan*. During subspace search expansion we incrementally expand the search region and sort subspaces in the region in order of the minimum distance from the query point. Algorithm 4 increases the search region width to the maximum distance from the query point to the farthest corner of the scanned subspaces. The next step scans the nearest subspace that has not already been scanned and sorts points in order of the distance from the query point. If the distance to the k-th point is less than the distance to the nearest unscanned subspace, the query process terminates.

3.2.5 Space split

Both K-d and Quad trees limit the number of points contained in each subspace; a subspace is split when the number of points in a subspace exceeds this limit. We determine the maximum number of points by the bucket size of the underlying storage

Algorithm 4 k Nearest Neighbors query

```
1: /* q be the point for the query. */
2: PQ_subspace ← CreatePriorityQueue()
3: PQ_result ← CreatePriorityQueue(k) /* queue capacity is k. */
4: width ← 0 /* region size for subspace search */
5: 𝕊_scanned ← φ /* scanned subspaces */
6: loop
7:    /* expand search region. */
8:    if PQ_subspace = φ then
9:       𝕊_next ← SubspacesInRegion(q, width) − 𝕊_scanned
10:      for each Subspace S ∈ 𝕊_next do
11:         Enqueue(S, MinDistance(q, S), PQ_subspace)
12:   /* pick the nearest subspace. */
13:   S ← Dequeue(PQ_subspace)
14:   /* search termination condition */
15:   if KthDistance(k, PQ_result) ≤ MinDistance(q, S) then
16:      return PQ_result
17:   /* scan and sort points by the distance from q. */
18:   for each Point p ∈ S do
19:      Enqueue(p, Distance(q, p), PQ_result)
20:   /* maintain search status. */
21:   𝕊_scanned ∪ S
22:   width ← max(width, MaxDistance(q, S))
```

Algorithm 5 Subspace name generation

```
1: /* Quad tree */
2: NewName ← φ
3: for each dimension in [1, ..., n] do
4:    NewName ∪ {OldName ⊕ 0, OldName ⊕ 1}
5: /* K-d tree */
6: NewName ← {OldName ⊕ 0, OldName ⊕ 1}
```

layer. Since the index layer is decoupled from the data storage layer, a subspace split 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. The number of new subspaces created depends on the index structure used: a K-d tree splits a subspace only in one dimension, resulting in two new subspaces, while a Quad tree splits a subspace in all dimensions, resulting in 2^n subspaces. For every dimension split, the name of the new subspaces is created by appending the old subspace name with a 0 and 1 at the position corresponding to the dimension being split. Algorithm 5 provides the pseudocode for subspace name generation following a split. Figure 7 provides an illustration of space splitting in the index layer for both Quad and K-d trees in a 2D space.

Even though conceptually simple, there are a number of implementation intricacies when splitting a space. These intricacies arise from the fact that the index layer is

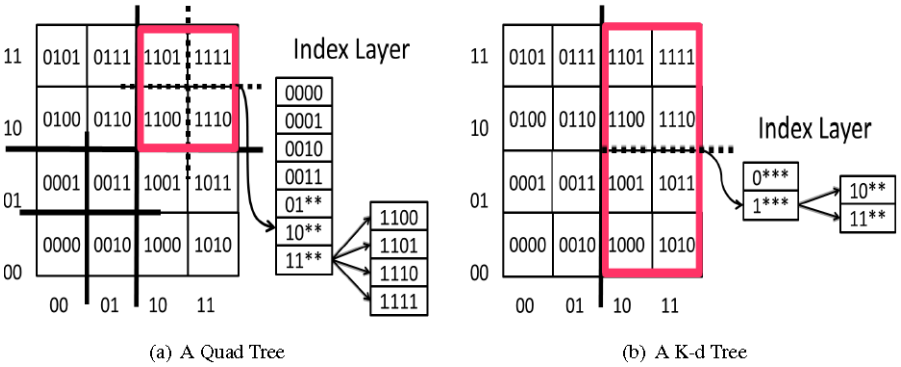


Fig. 7 Space split at the index layer

maintained as a table in the key-value store and most current key-value stores support transactions only at the single key level. Since a split touches at least three rows (one row for the old subspace name and at least two rows for the new subspaces), such a change is not transactional compared to other concurrent operations. For instance, an analysis query concurrent to a split might observe an inconsistent state. Furthermore, additional logic is needed to ensure the atomicity of the split. Techniques such as maintaining the index as a *key group* as suggested in [4] can be used to guarantee multi-key transactional consistency. The transactional operations may introduce additional overhead but we expect data movement at the bucket split seems to cost more expensive. We leave such extensions as future work. In addition, we leave out the delete operation because we think both spatial and temporal dimensions are indexed and it implicitly leads a data append model. However, deletion and the resulting merger of subspaces can be handled as straightforward extension of the Space Split algorithm.

3.3 Implementing the index layer

The index layer is a sorted sequence of subspace names. Since the subspace names encode the boundaries, the index layer essentially imposes an order between the subspaces. In our prior discussion, we represented the index layer as a monolithic layer of sorted subspace names. A single partition index is enough for many application needs. Assume each bucket can hold about 10^6 data points. Each row in the index layer stores very little information: the subspace name, the mapping to the corresponding bucket, and some additional metadata for query processing. Therefore, 100 bytes per index table row is a good estimate. The underlying key-value store partitions its data across multiple partitions. Considering the typical size of a partition in key-value stores is about 100 MB [3], the index partition can maintain a mapping of about 10^6 subspaces. This translates to about 10^{12} data points using a single index partition. This estimate might vary depending on the implementation or the configurations used. But it provides a reasonable estimate of the size.

Our implementation also partitions the index layer for better scalability and load balancing. We leverage the B+ tree style metadata access structure in Bigtable to

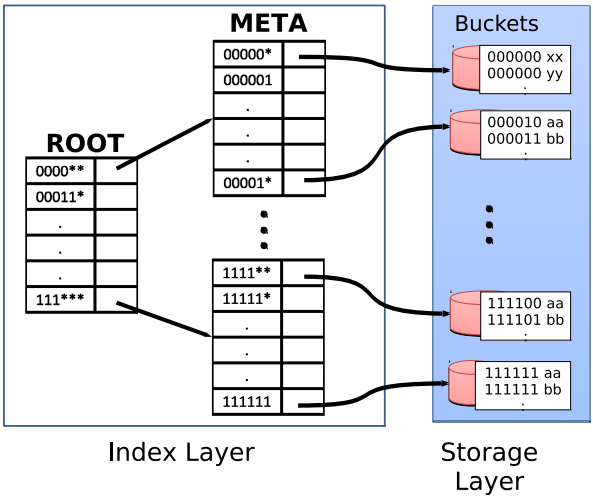


Fig. 8 Index layer implementation on bigtable

partition the index layer without incurring any performance penalty. Bigtable, and its open source variant HBase used in our implementation, uses a two level structure to map keys to their corresponding tablets. The top level structure (called the ROOT) is never partitioned and points to another level (called the META) which points to the actual data. Figure 8 provides an illustration of this index implementation. This seamless integration of the index layer into the ROOT-META structure of Bigtable does not introduce any additional overhead as a result of index accesses. Furthermore, optimizations such as caching the index layer and connection sharing between clients on the same host reduces the number of accesses to the index. Adding this additional level in the index structure, therefore, strikes a good balance between scale and the number of indirection levels to access the data items. Using an analysis similar to that used above, a conservative estimate of the number of points indexed by this two level index structure is 10^{21}.

4 Optimized query processing

Range query processing, presented in Sect. 3.2, performs first-level filtering by eliminating the need to scan the data points that do not intersect with the query regions. However, in determining which data buckets to scan, we must check index entries to determine if the sub-space corresponding to the entry intersects with the query. We determine the first index entry that intersects with the query region and then scan through the index to determine which sub-spaces intersect with the query. However, linearization techniques, such as Z-ordering, only approximately capture data locality; Z-ordering can effectively capture locality when the query region is closer to a hypercube. However, as the query region deviates from a hypercube or when one of the dimensions in unbounded, the scan through the index layer will potentially contain a lot of false positive checks of index buckets. Such false positive scans in the index layer unnecessarily increase the query processing cost. We now present an op-

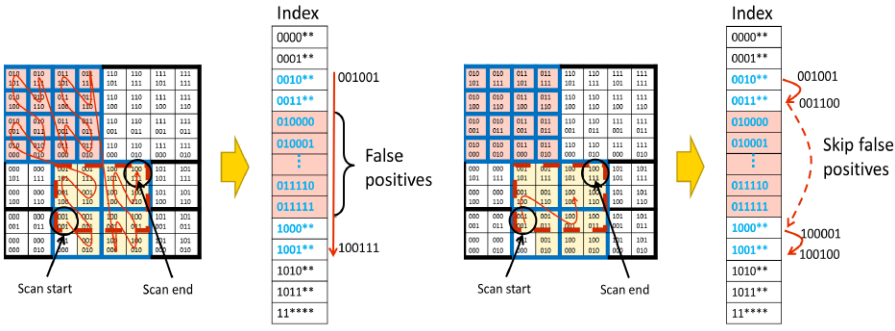


Fig. 9 False positives in scanning the index when processing multi-dimensional range queries. Such false positive must be eliminated to further improve query performance

timized query processing technique that eliminates these false positives, thus further improving query performance.

4.1 False positives in index scan

To motivate the scenario with large numbers of false positives, let us consider the example shown in Fig. 9(a). We use the Quad-tree as the index structure in our discussion, but the same applies to K-d tree as well. Assume that the index divides the space into sub-spaces as shown by the thick solid lines; the query region $\langle q_l, q_h \rangle$ is shown as a dashed rectangle. It is evident that scanning the index entries in linearization order (shown using arrows and bold-faced in the index layer) starting from q_l up to q_h results in scanning large numbers of index entries that do not overlap the query region; the false positive scans are emphasized using a shadow. Ideally, the query processing technique should be able to skip such false positives as shown in Fig. 9(b).

In the context of location aware services, query range sizes of spatial dimensions and temporal dimensions are often independent. For example, there are high aspect-ratio range queries which specify a narrow spatial region and a very long time interval, semi-infinite interval range queries which specify spatial ranges and only the start time in the temporal range, and whole intervals or missing dimensions in queries (such as queries which specify only spatial ranges but no temporal ranges). The numbers of false positives increase significantly as the dimensionality increases or as the number of index entries increases.

Space filling curves, such as the Z-ordering curves, map a multi-dimensional space to a linearized space. When a query with a regular hyper-rectangle region is projected to the linearized space using Z-ordering, the distance from the minimum value to the maximum value of the linearized space is proportional to the area of the regular hyper-rectangle region in the original space. However, when a range query includes a very long or an infinite interval, the linearized distance tends to be larger due to distortions introduced due to dimensionality reduction.

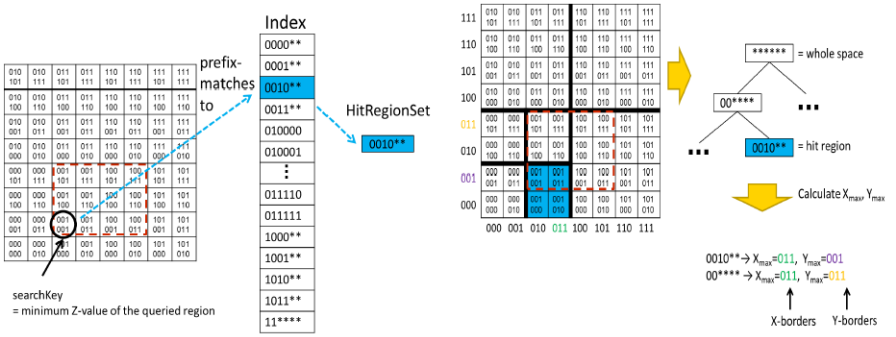
4.2 Eliminating false positives

We now present an optimized query processing technique, *Find & Skip*, that obviates the need for false positives in scanning index entries. The key idea is to recursively sub-divide the input query ($\langle q_l, q_h \rangle$) into sub-queries that emulate the way the index structure partitions the multi-dimensional space. Once the input query has been decomposed into sub-queries, these sub-queries are re-issued on the index layer, thus obviating the need to scan the index starting from q_l. The union of the ranges of the decomposed query is guaranteed to match the input query, thus guaranteeing correctness, i.e., no false negatives. In essence, a sequential scan of the index is replaced by random-accesses to the index and some additional computation to recursively decompose into sub-queries. The intuition is that this advanced query processing technique will be beneficial for highly selective queries encompassing a big range linearized space. Figure 10 illustrates the proposed Find & Skip algorithm using an example and Algorithm 6 outlines the query decomposition algorithm.

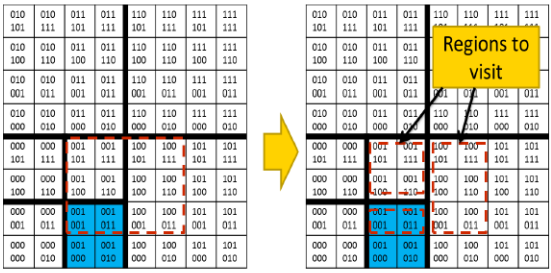
Consider our earlier example of the range query $\langle q_l, q_h \rangle$ which is shown as a dashed line in Fig. 10(a). The Find & Skip algorithm first looks-up the index entry corresponding to the bottom-left corner of the query region (q_l), a process similar to issuing a point query (Algorithm 6 line 8–10). Find & Skip then computes the borders of the sub-space that the index entry points to (Algorithm 6 line 12). The index entry 0010^{**} implies that the multi-dimensional space is partitioned *at least* at two levels as shown in Fig. 10(b); As discussed in Sect. 3, the sub-space naming scheme encodes the boundary information for the sub-space. Once Find & Skip computes the boundaries along which the index splits the query, the query region is partitioned accordingly. Figure 10(c) illustrates this query region partitioning into three sub-regions: the lower-left region is the hit region (i.e., the index bucket that overlaps with the query and is not further sub-divided) and the remaining regions are added to *RegionsToVisit* (Algorithm 6 line 13–14). Finally, the above steps are recursively repeated on the sub-regions in *RegionsToVisit* until no further sub-division is possible; Figs. 10(d) and 10(e) illustrate this recursive sub-division.

The Find & Skip algorithm guarantees that the query region is sub-dividing into non-overlapping sub-regions. Further, any region added to *HitRegionSet* will not overlap with any of the regions in *RegionsToVisit* that will subsequently be added to *HitRegionSet*. Therefore, the index scan and query processing are parallelizable, i.e., while the Find & Skip algorithms determines the next regions to scan, another thread can start scanning the buckets for the entries that have already been added to *HitRegionSet*. This parallelization is beneficial in reducing the overall query processing time and amortizing the added latency introduced by multiple random index accesses, potentially over a network, made by the Find & Skip algorithm.

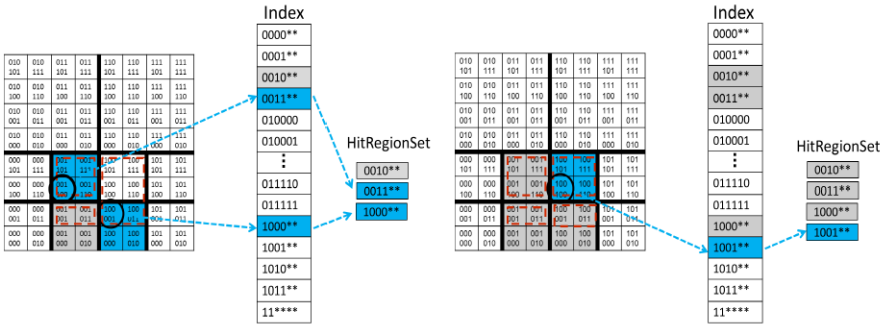
An important property of the Find & Skip algorithm is that upon termination, it ensures that the number of index accesses is exactly equal to the number of index entries that intersect a query region and guarantees that the algorithm minimizes the number of index accesses; a proof of this property is provided in Sect. 4.3. Furthermore, implementing Find & Skip does not require maintaining any additional information in the index layer since it leverages information already encoded in the index entries.



(a) Query region identifies the index entry(Alg. 6 line 8-10) (b) Inferring space split from the index entry(Alg. 6 line 12)



(c) Inferring space split from the index entry(Alg. 6 line 13-14)



(d) False positives in scan (e) Scan avoiding false positives

Fig. 10 An example of the Find & Skip algorithm that eliminates false positives in the index scan

4.3 Correctness guarantees

Guarantee 1 HitRegionSet *includes all regions that intersect the query region*.

Find & Skip sub-divides the given query region into a set of mutually exclusive sub-regions whose union represents the original query. Every single iteration of the find step and the divide step partitions a current region into mutually exclusive and collectively exhaustive sub-regions at least one of which is added to the *HitRegionSet* while the remaining sub-regions are added to *RegionsToVisit*. Regions in *RegionsToVisit* are decomposed into smaller sub-regions and the iterations will terminate when

Algorithm 6 The Find & Skip Algorithm

```
1: /* ⟨q_l, q_h⟩ be the query region. */
2: /* Initialization */
3: HitRegionSet ← φ
4: RegionsToVisit ← ⟨q_l, q_h⟩
5: while RegionsToVisit ≠ φ do
6:     /* Find the region corresponding to the bottom-left corner of the query region */
7:     currentRegion ← extractOne(RegionsToVisit)
8:     searchKey ← minZValueOf(currentRegion)
9:     hitRegion ← lookupIndex(searchKey)
10:    HitRegionSet ← HitRegionSet ∪ hitRegion
11:    /* Divide current region into smaller unvisited regions */
12:    borders ← calcBorders(hitRegion)
13:    subregions ← partition(currentRegion, borders)
14:    RegionsToVisit ← RegionsToVisit ∪ {subregions − hitRegion}
15: return HitRegionSet
```

a region visited is a hit region and is not further sub-divided. Therefore, after the recursive sub-division, the query region is totally covered by the union of the regions in *HitRegionSet*. As stated earlier, any pair of regions generated by the space partitioning method have an empty intersection. Therefore, it is impossible to have a region that is not in *HitRegionSet* and overlaps partially with the query region. Hence, *HitRegionSet* contains *all* regions which intersect the query region.

Guarantee 2 *HitRegionSet does not include* any *region that does not intersect the query region*.

When we lookup the index, we choose a search point which exists within the query region. The index returns hit regions which cover the search point. Therefore, this guarantees that any region in *HitRegionSet* intersects at least at a single point within the query region.

5 Data storage layer

The data storage layer of $\mathcal{MD}$-HBase is a range partitioned key-value store. We use HBase, the open source implementation of Bigtable, as the storage layer for $\mathcal{MD}$-HBase.

5.1 HBase overview

A table in HBase consists of a collection of splits, called *regions*, where each region stores a range partition of the key space. Its architecture comprises a three layered B+ tree structure; the regions storing data constitute the lowest level. The two upper levels are special regions referred to as the `ROOT` and `META` regions, as described in Sect. 3.3. An HBase installation consists of a collection of servers, called *region servers*, responsible for serving a subset of regions. Regions are dynamically assigned

to the region servers; the META table maintains a mapping of the region to region servers. If a region's size exceeds a configurable limit, HBase splits it into two subregions. This allows the system to grow dynamically as data is inserted, while dealing with data skew by creating finer grained partitions for hot regions.

5.2 Implementation of the storage layer

A number of design choices exist to implement the data storage layer, and it is interesting to analyze and evaluate the trade-offs associated with each of these designs. Our implementation uses different approaches as described below.

5.2.1 Table share model

In the table share model, all buckets share a single HBase table where the data points are sorted by their corresponding Z-value which is used as the key. Our space splitting method guarantees that data points in a subspace are contiguous in the table since they share a common prefix. Therefore, buckets are managed by keeping their start and end keys. This model allows efficient space splitting that amounts to only updating the corresponding rows in the index table. On the other hand, this model restricts a subspace to be mapped to only a single bucket.

5.2.2 Table per bucket model

The table per bucket model represents another extreme of the spectrum where we allocate a table per bucket. The data storage layer therefore consists of multiple HBase tables. This model provides flexibility in mapping subspaces to buckets and allows greater parallelism by allowing operations on different tables to be dispatched in parallel. However, a subspace split in this technique is expensive since this involves moving data points from the subspaces to the newly created buckets.

5.2.3 Hybrid model

The hybrid model strikes a balance between the table share model and the table per bucket model. First, we partition the space and allocate a table to each resulting subspace. After this initial static partitioning, when a subspace is split as a result of an overflow, the newly created subspaces share the same table as that of the parent subspace. Of course, it requires domain knowledge and an understanding of data distribution to decide the initial partitioning. However, it is only a one-time process when the data store is initialized, and is not needed in steady state operation.

5.2.4 Region per bucket model

A HBase table comprises of many regions. Therefore, another approach is to use a single table for the entire space and use each region as a bucket. A region split in HBase is quite efficient and is executed asynchronously. This region per bucket design therefore has a low space split cost while being able to efficiently parallelize

operations across the different buckets. However, contrary to the other three models discussed, this model is intrusive and requires changes to HBase; a hook is added to the split code to determine the appropriate split point based on the index structure being used.

5.3 Optimizations

Several optimizations are possible that further improve performance of the storage layer.

5.3.1 Space splitting pattern learning

Space splitting introduces overhead that affects system performance. Therefore, advanced prediction of a split can be used to perform an asynchronous split that will reduce the impact on the inserts and queries executing during a split. One approach is to learn the split patterns to reduce occurrences of space splitting. Location data is inherently skewed, however, as noted earlier, such skew is often predictable by maintaining and analyzing historical information. Since $\mathcal{MD}$-HBase stores historical data, the index structure inherently maintains statistics of the data distribution. To estimate the number of times a new bucket will be split in the future, we lookup how many buckets were allocated to the same spatial region in the past. For example, when we allocate a new bucket for region $([t_0, t_1], [x_0, x_1], [y_0, y_1])$, we lookup buckets for region $([t_0 - t_s, t_1 - t_s], [x_0, x_1], [y_0, y_1])$ in the index table. The intuition is that the bucket splitting pattern in the past is a good approximate predictor for the future.

We think the effect of the optimization depends on the underlying storage model. Table per bucket model may receive huge bonus because the model costs expensive at the bucket split due to a synchronous table copy. On the other hand, we think the effect is limited in case of Table Share Model and Region per Bucket Model because table split cost is much lower than that of Table per bucket model.

We have two cases that future accesses do not follow past behavior: a hot region turns to a cold region and vise versa. The former may increase buckets with less occupancy. It does not affect insertion throughput but may affect query performance because the store may touch more buckets at query processing. The later is the same situation that this optimization is not applied.

5.3.2 Pre-partitioning and lazy bucket allocation

To further reduce the overhead of space splitting and the splits needed when large amounts of data is inserted, our implementation pre-partitions space as a grid once the split patterns are learned from historical data. Even though pre-partitioning the space eliminates the number of space splits, it may unnecessarily increase number of index entries, thus affecting query performance. Therefore, our implementation uses lazy bucket allocation where a new bucket in the storage layer is allocated only when the first data item is inserted to the corresponding bucket.

5.3.3 Random projection

Data skew makes certain subspaces hot, both for insertion and querying. An optimization is to map a subspace to multiple buckets with points in the subspace distributed amongst the multiple buckets. When the points are distributed randomly, this technique is called *random projection*. When inserting a data point in a subspace, we randomly select any one of the buckets corresponding to the subspace. A range query that includes the subspace must scan all the buckets mapped to the subspace. Thus, the random projection technique naturally extends our original algorithm; it, however, presents a trade-off between load balancing and a potential increase in query cost.

5.3.4 Bucket scan order

In case of the Table share model, the order in which buckets are scanned affects its performance. Scanning buckets in the order of the sub-space names results in conflicting accesses to the disk, thus limiting the disk bandwidth available to the queries. This is because the sub-space names tend to be similar due to locality. In addition, if the bucket size is smaller than the region size, some buckets might be allocated to the same region in HBase. To mitigate such conflicts, our implementation randomizes the bucket scan order and our experiments have shown that such randomization can improve query performance by about 30 %.

6 Experimental evaluation

We now present a detailed evaluation of our prototype implementation of $\mathcal{MD}$-HBase. We implemented our prototype using HBase 0.20.6 and Hadoop 0.20.2 as the underlying system. We evaluate the trade-offs associated with the different implementations for the storage layer and compare our technique with a MapReduce style analysis and query processing using only linearization over HBase. Our experiments were performed on an Amazon EC2 cluster whose size was varied from 4 to 16 nodes. Each node consists of 4 virtual cores, 15.7 GB memory, 1,690 GB HDD, and 64 bit Linux (v2.6.32). We evaluate the different implementations of the storage layer as described in Sect. 5: table per bucket design simulating the K-d and Quad trees (*TPB/Kd* and *TPB/Quad*), table sharing design simulating the K-d and Quad trees (*TS/Kd* and *TS/Quad*), and a region per bucket design for K-d trees (*RPB/Kd*).[3] The baseline is an implementation using z-ordering for linearization (*ZOrder*) without any specialized index. We also implemented the queries in Map Reduce to evaluate the performance of a MapReduce system (*MR*) performing a full parallel scan of the data; our evaluation used the Hadoop runtime system. Our evaluation uses synthetically generated data sets primarily due to the need for huge data sets (hundreds of gigabytes) and the need to control different aspects, such as skew and selectivity, to better understand the behavior of the system.

[3] Since in HBase, a region can only be split into two sub-regions, we could not implement RPB for Quad trees as our experiments are for a 3D space.

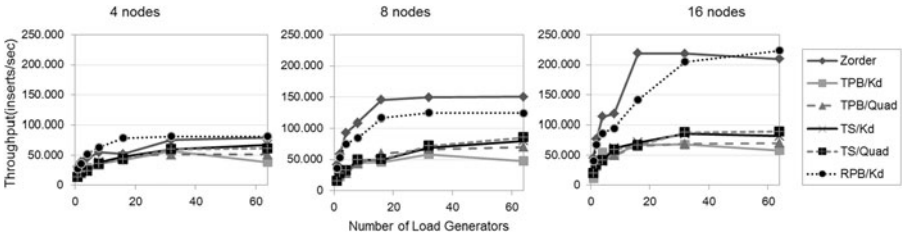


Fig. 11 Insert throughput (location updates per second) as a function of load on the system. Each load generator creates 10,000 inserts per second

6.1 Data model

In this section, we explain the data model of location information used in our experimental evaluation of $\mathcal{MD}$-HBase. A user (or any mobile object) is identified by a user (or object) ID. The user's location is stored as the latitude and longitude. While the current location of a user is of interest to all location based services, many location enabled applications mine a user's mobility patterns to further improve the personalization and user experience. Therefore, $\mathcal{MD}$-HBase stores the entire (or a large subset of) location history of all the users. That is, when a user moves, it results in an insert into the data store with a newer timestamp and does not result in a delete of the old location. The location with the largest timestamp value is treated as the most recent location of the user.

Each record has four key attributes that are of interest to us: object ID, timestamp, latitude, and longitude. Depending on the application's requirements, additional attributes might also be stored, and $\mathcal{MD}$-HBase is flexible enough to handle such additional application-specified attributes. Object ID is an unsigned long integer (64-bit) and the other attributes are unsigned integers (32-bit). $\mathcal{MD}$-HBase indexes the three dimensions: timestamp, latitude, and longitude. Therefore, all experiments treat a data set as a three dimensional data set.

Different objects may exist at the same location and at the same time. Our underlying key-value store implementation, HBase, permits each key-value entry to have any number of columns. We define a column per object with a given object ID. Therefore, different objects can be co-located within the same key-value entry.

6.2 Insert throughput

Supporting high insert throughput for location updates is critical to sustain the large numbers of location devices. We evaluated insert performance using five different implementations of the storage layer on a cluster with 4, 8, and 16 commodity nodes. Figure 11 plots the insert throughput as a function of the load on the system. We varied the number of load generators from 2 to 64; each generator created a load of 10, 000 inserts per second. We used a synthetic spatially skewed data set using a Zipfian distribution with a Zipfian factor of 1.0 representing moderately skewed data. Using a Zipfian distribution allows us to control the skew while allowing quick generation of large data sets. Both the RPB/Kd system and the ZOrder systems showed

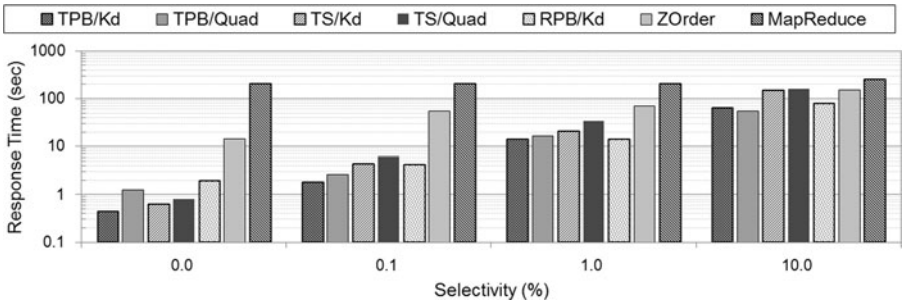


Fig. 12 Response times (in log scale) for range query as function of selectivity

good scalability; on the 16 node cluster, the RPB/Kd and ZOrder implementation sustained a peak throughput of about 220 K location updates per second. In a system where location devices register a location update every minute, this deployment can handle 10–15 million devices. The main reason for the low scalability of the table per bucket and table sharing designs is the cost associated with the splitting a bucket. On the other hand, the region per bucket design splits buckets asynchronously using the HBase region split mechanism which is relatively inexpensive. The TPB/TS systems block other operations until the bucket split completes. In our experiments, the TPB design required about 30 to 40 seconds to split a bucket and the TS design required about 10 seconds. Even though these designs result in more parallelism in distributing the insert load on the cluster, the bucket split overhead limits the peak throughput sustained by these designs.

6.3 Range query

We now evaluate range query performance using the different implementations of the index structures and the storage layer and compare performance with ZOrder and MR. The MR system filters out points matching the query range and reports aggregate statistics on the matched data points.

We generated about four hundred million points using a network-based model by Brinkhoff et al. [2]. Our data set simulates 40,000 objects moving 10,000 steps along the streets of the San Francisco bay area. Since the motion paths are based on a real map, this data set is representative of real world data distributions. Evaluation using other models to simulate motion is proposed future work. We executed the range queries on a four-node cluster in Amazon EC2.

Figure 12 plots the range query response times for the different designs as a function of the varying selectivity. As is evident from Fig. 12, all the $\mathcal{MD}$-HBase design choices outperform the baseline systems. In particular, for highly selective queries where our designs show a one to two orders of magnitude improvement over the baseline implementations using simple Z-ordering or using MapReduce. Moreover, the query response time of our proposed designs is proportional to the selectivity, which asserts the gains from the index layer when compared to brute force parallel scan of the entire data set as in the MR implementation whose response times are the worst amongst the alternatives considered and is independent of the query selectivity.

Table 1 False positive scans on the ZOrder system

	Selectivity (%)			
	0.01	0.1	1.0	10.0
No. of buckets scanned	7	28	34	45
False positives	3	22	16	16
Percentage false positive	42.9 %	78.5 %	47.1 %	35.5 %

In the design using just Z-ordering (ZOrder), the system scans between the minimum and the maximum Z-value of the query region and filters out points that do not intersect the query ranges. If the scanned range spans several buckets, the system parallelizes the scans per bucket. Even though the ZOrder design shows better performance compared to MR, response time is almost ten times worse when compared to our proposed approaches, especially for queries with high selectivity. The main reason for the inferior performance of the ZOrder is the false positive scans resulting from the distortion in the mapping of the points introduced by linearization. Table 1 reports the number of false positives (buckets that matched the query range but did not contain a matching data point) of the ZOrder design. For example, in case of 0.1 percent selectivity query, 22 out of 28 buckets scanned are false positive. The number of false positive scans depends on the alignment of the query range with the bucket boundaries. In the ideal situation of a perfect match, query performance is expected to be close to that of our proposed designs; however, such ideal queries might not be frequently observed in practice. Our proposed designs can further optimize the scan range within each region. Since we partition the target space into the regular shaped subspaces, when the query region partially intersects a subspace, $\mathcal{MD}$-HBase can compute the minimum scan range in the bucket from the boundaries of the query region and the subspace. In contrast, the ZOrder design partitions the target space into irregular shaped subspaces and hence must scan all points in the selected region.

A deeper analysis of the different alternative designs for $\mathcal{MD}$-HBase shows that the TPB designs result in better performance. The TS designs often result in high disk contention since some buckets are co-located in the same data partition of the underlying key-value store. Even though we randomize the bucket scan order in the TS design, significant reduction in disk access contention is not observed. The TPB designs therefore outperform the TS designs. For queries with high selectivity, the RPB design has a longer response time compared to other designs; this however is an artifact of our implementation choice for the RPB design. In the RPB design, we override the pivot calculation for a bucket split and delegate the bucket maintenance task to HBase which minimizes the insert overhead. However, since HBase stores bucket boundary information in the serialized form, the index lookup for query processing must de-serialize the region boundaries. As a result, for queries with very high selectivity, the index lookup cost dominates the total response time.

Comparing the K-d and the Quad trees, we observed better performance of K-d trees for queries with high selectivity. However, as the selectivity decreases, the response times for Quad trees are lower. The K-d tree creates a smaller number of buckets while the Quad tree provides better pruning compared to the K-d tree. In the case of high selectivity queries, sub-query creation cost dominates the total response

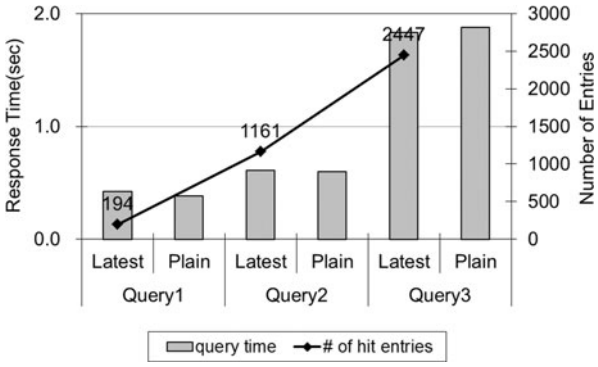


Fig. 13 Response times for latest position range query as function of selectivity. The vanilla range query, where the current position requirement is not specified, is shown as a comparison to compare the overheads. As is evident, the current position query does not introduce any additional overhead over the vanilla range query

time; each sub-query typically scans a short range and the time to scan a bucket comprises a smaller portion of the total response time. As a result, the K-d tree has better performance compared to the Quad tree. On the other hand, in case of low selectivity queries, the number of points to be scanned dominates the total response time since each sub-query tends to scan a long range. The query performance therefore depends on the pruning power of the index structure.

We also expect the bucket size to potentially affect overall performance. Selecting a bucket size is a trade-off between the insertion throughput and query performance. Larger buckets reduce the frequency of bucket splits thus improving insertion throughput but limits sub-space pruning power during query processing.

6.4 Range query for mobility

We now evaluate a case where an application (similar to application examples in Sect. 2.1) is interested in only the current object locations. Since $\mathcal{MD}$-HBase records the entire history of object locations, a plain range query may return duplicate objects. Therefore we need to post-process the result to extract objects with the latest position. To design the post-process, we assume that object positions are measured periodically with a certain interval, and object moving speed is finite.

We now define the latest position range query to achieve the objective. The latest position range query takes these parameters: *query region*, *margin size* and *measurement interval*. *margin size* guarantees that the final result does not contains objects that moved out from the query region in the measurement interval. The query processing proceeds as follows. First, we run a multidimensional range query. The query region is enlarged by *margin size* and the range of the temporal dimension is from the current time going backwards to the *measurement interval*. The results are aggregated per object ID and the system finally returns the objects with the latest timestamp within *query region*.

To evaluate the performance of such a query, we use the same data set as described in Sect. 6.3. We use three queries and each query returns a result set of different size, i.e., we vary the selectivity of the queries. We execute both the latest position range queries and the simple range queries to compare the overheads, if any.

Figure 13 summarizes the results of the experiment. As expected, the response time of both queries depends on the result set size. As is also evident, the latest posi-

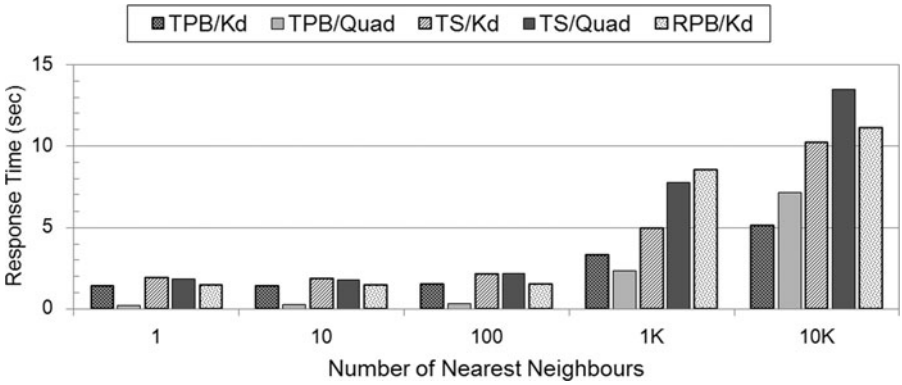


Fig. 14 Response time for kNN query processing as a function of the number of neighbors returned

tion range queries are competitive against the vanilla range queries where the current position requirement is not specified. Therefore, the results show that $\mathcal{MD}$-HBase can effectively retrieve latest positions of objects even though $\mathcal{MD}$-HBase keep the entire location history as the object moves. Therefore, $\mathcal{MD}$-HBase can efficiently process both types of queries.

6.5 kNN query

We now evaluate the performance of $\mathcal{MD}$-HBase for k Nearest Neighbor (kNN) queries using the same dataset as the previous experiment. Figure 14 plots the response time (in seconds) for kNN queries as a function of the number of neighbors to be returned; we varied k as 1, 10, 100, 1,000, and 10,000. In this experiment, expansion of the search region did not occur for $k \leq 100$, resulting in almost similar performance for all the designs. The TPB/Quad design has the best performance where the response time was around 250 ms while that of other designs is around 1500 ms to 2000 ms. As we increase k to 1 K and 10 K, the response times also increase; however, the increase is not exponential, thanks to the best-first search algorithm used. For example, when we increased k from 1 K to 10 K, the response time increased only by three times in the worst case.

The TPB/Quad design outperformed the other designs for $k \leq 100$. Due to the larger fan-out, the bucket size of a Quad tree tends to be smaller than that of a K-d tree. As a result, TPB/Quad has smaller bucket scan times compared to that of the TPB/Kd design. However, both TS designs have almost similar response times for small k. We attribute this to other factors like the bucket seek time. Since the TPB designs use different tables for every bucket, a scan for all points in a bucket does not incur a seek to the first point in the bucket. On the other hand, the TS designs must seek to the first entry corresponding to a bucket when scanning all points of the bucket. As mentioned earlier, the RPB design has a high index lookup overhead which increases the response time, especially when k is large. This however is an implementation artifact; in the future, we plan to explore other implementation choices for RPB that can potentially improve its performance.

Comparing between the K-d tree and the Quad tree, Quad trees have smaller response times for smaller k, while K-d trees have smaller response times for larger k. For small values of k, the search region does not expand. Smaller sized buckets in a Quad tree result in better performance compared to larger buckets in a K-d tree. On the other hand, for larger values of k, the probability of a search region expansion is higher. Since a Quad tree results in smaller buckets, it results in more search region expansions compared to that of K-d tree, resulting in better performance of K-d tree.

Since the ZOrder and the MR systems do not have any index structure, kNN query processing is inefficient. One possible approach is to iteratively expand the query region until the desired k neighbors are retrieved. This technique is however sensitive to a number of parameters such as the initial query range and the range expansion ratio. Due to the absence of an index, there is no criterion to appropriately determine the initial query region. We therefore do not include these baseline systems in our comparison; however, the performance of ZOrder and MR is expected to be significantly worse than the proposed approaches.

6.6 Evaluating Find & Skip

We use the network-based model by Brinkhoff et al. [2] to generate a synthetic data set to evaluate the effectiveness of the Find & Skip query processing technique when compared to the simple technique that scans the index layer. The synthesized data set comprises 800 million data points corresponding to five cities: a big city with 400 million points and four small cities with 100 million points each. Each city has approximately 1500 K $\times$ 1500 K spatial steps (or units) and each person moves 10 K steps for 10 minutes, thus resulting in 6,000,000 time steps (600 secs $\times$ 10,000 steps). Figures 15–17 compare the performance of the Find & Skip technique (*F&S*) to the vanilla scan based technique (*Scan*). Each experiment is repeated with different bucket sizes in the storage layer; the bucket sizes are varied from 8 MB to 64 MB. Smaller bucket sizes imply more index entries representing scenarios where Find & Skip will be beneficial. In each figure, the primary y-axis (left) reports the end-to-end response time (in seconds) for processing the query (represented using the bars) and the secondary y-axis (right) represents the number of index entries accessed (in logscale) to process the query (represented using the lines). The end-to-end query response time is further sub-divided into the time spent in scanning the index layer and the storage layer respectively. The simple scan technique sequentially scans the index layer and hence is efficient but prone to false positives. The F&S technique performs random short scans on the index and eliminates false positives. The goal of this evaluation is to understand this trade-off between the two techniques.

Figure 15 reports the response times for queries with 10 K $\times$ 10 K spatial region and whole time duration. This is a typical case where the total time depends on the time to scan the index. Even though the narrow spatial region may potentially limit the number of index entries that intersect with the query, the whole time duration maximizes the distortion of the linearized query region, thus resulting in large numbers of false positives. Figure 15 shows that the simple scan strategy takes longer time where the bucket size is smaller. As expected, the improvements are more significant when the bucket size is smaller, since smaller buckets correspond to more

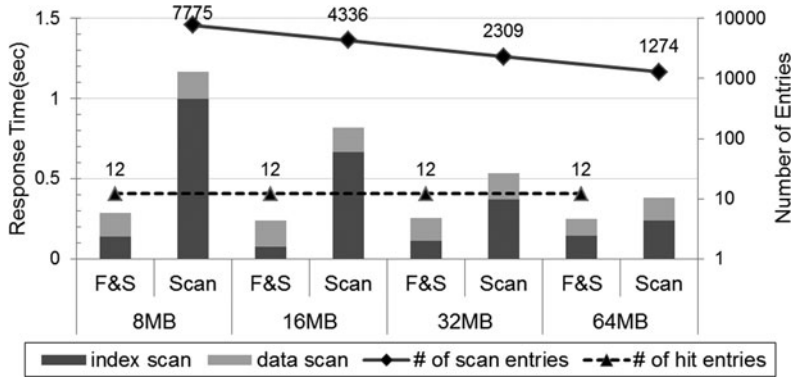


Fig. 15 10 K × 10 K spatial range with no time bounds

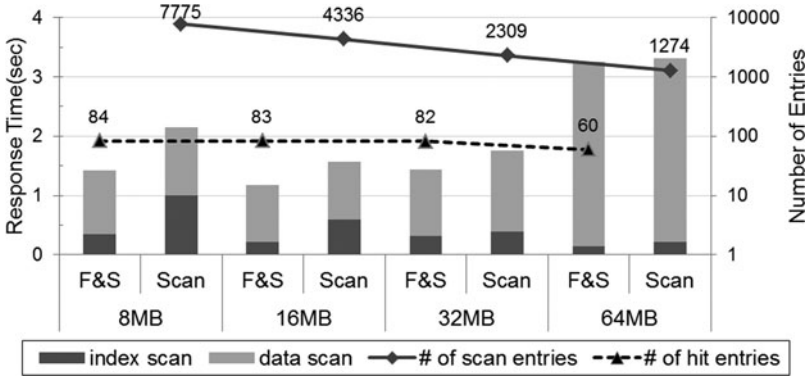


Fig. 16 100 K × 100 K spatial range with no time bounds

index entries and potentially more false positives. The index entries scanned further supports our argument. The vanilla index scan technique ends up scanning the entire index layer while F&S scans only a few index entries that overlap with the query. For instance, when the bucket size is 8 MB, the simple scan strategy accesses 7753 index entries out of which only 12 entries (as found by F&S) intersect with the query, thus demonstrating the high percentage of false positives. Note that the response times and the number of entries scanned by the F&S algorithm is independent of the bucket size and total number of index entries since it efficiently skips the false positive entries.

Figure 16 shows the response times for queries with 100 K × 100 K spatial region and the whole time duration. In comparison with the previous query, the number of hit entries increases due to the larger spatial region. This increase in index hits affects the index scan time of the Find & Skip strategy which issues multiple reads on the index layer. The simple strategy scans the entire index layer even for this query and the scan cost is amortized by the higher percentage of hits. This query also results in an increase in the time spent in scanning the data buckets which also increases as the bucket size increases. Therefore, in the case where a 64 MB storage bucket is used, the total response time is dominated by the time to scan the data buckets.

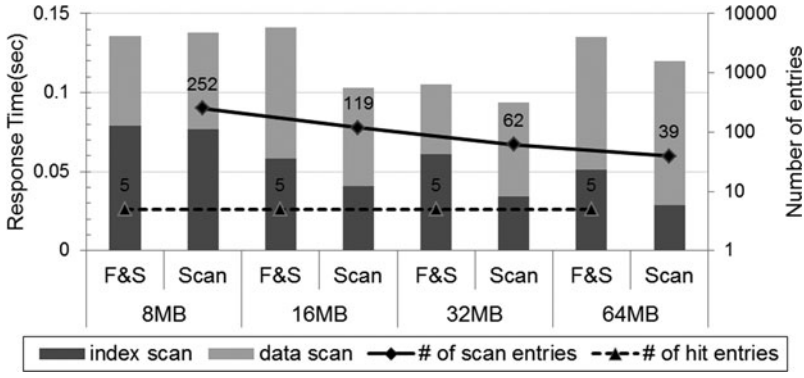


Fig. 17 100 K × 100 K spatial bound with 100 K time steps

Figure 17 shows the response times for queries with 100 K × 100 K spatial region and a limited temporal range of 100 K steps. This query represents a region with similar span in all dimensions where the number of false positives is low and number of hit entries is relatively high. Therefore, the performance gains in using the Find & Skip strategy is expected to be marginal. In fact, for such queries, the simple scan strategy results in a single sequential scan of the index layer and might outperform F&S which issues multiple random accesses to the index layer. As is evident from the figure, the performance of F&S is comparable to Scan even for such queries.

In conclusion, our evaluation shows that the Find & Skip algorithm considerably reduces the query processing cost in scenarios where one or more query dimensions are unbounded, have long intervals, or in scenarios with large numbers of index entries and the queries have high selectivity. In scenarios where the query region is closer to a hypercube or when the selectivity is low, Find & Skip exhibits performance similar to the simple scan strategy.

7 Related work

Scalable data processing—both for large update volumes and data analysis—has been an active area of interest in the last few years. When dealing with a large volume of updates, key-value stores have been extremely successful in scaling to large data volumes; examples include Bigtable [3], HBase [10] etc. These data stores have a similar design philosophy where they have made scalability and high availability as the primary requirement, rich functionality being secondary. Even though these systems are known to scale to terabytes of data, the lack of efficient multi-attribute based access limits their application for location based applications. On the other hand, when considering scalable data processing, MapReduce [5, 9] has been the most dominant technology, in addition to parallel databases [6]. Such systems have been proven to scale to petabytes of data while being fault-tolerant and highly available. However, such systems are suitable primarily in the context of batch processing. Furthermore, in the absence of appropriate multi-dimensional indices, MapReduce style processing has to scan through the entire dataset to answer queries. $\mathcal{MD}$-HBase complements

both key-value stores and MapReduce style processing by providing an index structure for multi-dimensional data. As demonstrated in our prototype, our proposed indexing layer can be used directly with key-value stores. Along similar lines, the index layer can also be integrated with MapReduce to limit the retrieval of false positives.

Our use of linearization for transforming multi-dimensional data points to a single dimensional space has been used recently in a number of other techniques. For instance, Jensen et al. [11] use Z-ordering and Hilbert curves as space filling curves and construct a B+ tree index using the linearized values. Tao et al. [17] proposed an efficient indexing technique for high dimensional nearest neighbor search using a collection of B-tree indices. The authors first use locality sensitive hashing to reduce the dimensionality of the data points and then apply Z-ordering to linearize the dataset, which is then indexed using a B-tree index. Our approach is similar to these approaches, the difference being that we build a K-d tree and a Quad tree based index using the linearized data points. The combination of the index layer and the data storage layer in $\mathcal{MD}$-HBase however resembles a B+ tree, reminiscent of the Bigtable design. Subspace pruning in the index layer is key to speeding up the range query performance which becomes harder for data points with high dimensionality. In such cases, dimensionality reduction techniques, as used by Tao et al. [17], can be used to improve the pruning power.

Another class of approaches make the traditional multidimensional indices more scalable. Wang et al. [18] and Zhang et al. [20] proposed similar techniques where the systems have two index layers: a global index and a local index. The space is partitioned into several subspaces and each sub-space is assigned a local storage. The global index organizes subspaces and the local index organizes data points in the subspace. Wang et al. [18] construct a content addressable network (CAN) over a cluster of R-tree indexed databases while Zhang et al. [20] use an R-tree as the global index and a K-d tree as the local index. Along these lines, $\mathcal{MD}$-HBase only has a global index; it can however be extended to add local indices within the data storage layer.

Lawder [12] proposed a technique to skip index entries for a point-base partitioned space linearized using the Hilbert curve. The proposed technique can reduce number of buckets to be scanned since point-base partitioning retains higher occupancy of buckets compared to trie-base partitioning, and the Hilbert curve provides better locality than the Z-order curve. Introducing these concepts into the design of $\mathcal{MD}$-HBase are worthwhile directions of future work.

8 Conclusion

Scalable location data management is critical to enable the next generation of location based services. We proposed $\mathcal{MD}$-HBase, a scalable multi-dimensional data store supporting efficient multi-dimensional range and nearest neighbor queries. $\mathcal{MD}$-HBase layers a multi-dimensional index structure over a range partitioned key-value store. Using a design based on linearization, our implementation layers standard index structures like K-d trees and Quad trees. We demonstrated how the proposed

index layer and obviate scanning data buckets that do not overlap with the query region. We also presented an optimized query processing techniques that ensures that there are no false positive scans even in the index layer.

We implemented our design on HBase, a standard open-source key-value store, with minimal changes to the underlying system. The scalability and efficiency of the proposed design is demonstrated through a thorough experimental evaluation. Our evaluation using a cluster of nodes demonstrates the scalability of $\mathcal{MD}$-HBase by sustaining insert throughput of over hundreds of thousands of location updates per second while serving multi-dimensional range queries and nearest neighbor queries in real time with response times less than a second. In the future, we plan to extend our design by adding more complex analysis operators such as skyline or cube, and exploring other alternative designs for the index and data storage layers.

Acknowledgements This work is partly funded by NSF grants III 1018637 and CNS 1053594.

References

1. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
2. Brinkhoff, T., Str, O.: A framework for generating network-based moving objects. GeoInformatica **6**, 2002 (2002)
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: OSDI, pp. 205–218 (2006)
4. Das, S., Agrawal, D., El Abbadi, A.: G-Store: a scalable data store for transactional multi key access in the cloud. In: SOCC, pp. 163–174 (2010)
5. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
6. DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Commun. ACM **35**(6), 85–98 (1992)
7. Finkel, R.A., Bentley, J.L.: Quad trees: a data structure for retrieval on composite keys. Acta Inform. **4**, 1–9 (1974)
8. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: SIGMOD, pp. 47–57 (1984)
9. The Apache Hadoop Project. http://hadoop.apache.org/core/ (2010)
10. HBase: Bigtable-like structured storage for Hadoop HDFS. http://hadoop.apache.org/hbase/ (2010)
11. Jensen, C.S., Lin, D., Ooi, B.C.: Query and update efficient b+-tree based indexing of moving objects. In: VLDB, pp. 768–779 (2004). VLDB Endowment
12. Lawder, J.K.: Querying multi-dimensional data indexed using the Hilbert space-filling curve. SIGMOD Rec. **30**, 2001 (2001)
13. Morton, G.M.: A computer oriented geodetic data base and a new technique in file sequencing. Tech. rep., IBM Ottawa, Canada (1966)
14. Nishimura, S., Das, S., Agrawal, D., El Abbadi, A.: MD-HBase: a scalable multi-dimensional data infrastructure for location aware services. In: MDM, pp. 7–16 (2011)
15. Ramabhadran, S., Ratnasamy, S., Hellerstein, J.M., Shenker, S.: Prefix hash tree: an indexing data structure over distributed hash tables. Tech. rep., Intel Research, Berkeley (2004)
16. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, San Francisco (2005)
17. Tao, Y., Yi, K., Sheng, C., Kalnis, P.: Quality and efficiency in high dimensional nearest neighbor search. In: SIGMOD, pp. 563–576 (2009)
18. Wang, J., Wu, S., Gao, H., Li, J., Ooi, B.C.: Indexing multi-dimensional data in a cloud system. In: SIGMOD, pp. 591–602 (2010)
19. http://en.wikipedia.org/wiki/List_of_mobile_network_operators (2010)
20. Zhang, X., Ai, J., Wang, Z., Lu, J., Meng, X.: An efficient multi-dimensional index for cloud data management. In: CloudDB, pp. 17–24 (2009)