SkipIndex: Towards a Scalable Peer-to-Peer Index Service for High Dimensional Data

Chi Zhang[*] *Arvind Krishnamurthy*[†] *Randolph Y. Wang*[*]

Abstract

Indexing of high-dimensional data is essential for building applications such as multimedia retrieval, data mining, and spatial databases. Traditional index structures rely on centralized processing. This approach does not scale with the rapidly increasing amount of application data available on massively distributed systems like the Internet.

In this paper, we propose a distributed high-dimensional index structure based on peer-to-peer overlay routing. A new routing scheme is used to lookup data keys in the distributed index, which guarantees logarithmic lookup and maintenance cost, even in the face of skewed datasets. We propose a novel nearest neighbor (NN) query scheme that can substantially reduce search cost by sacrificing a small amount of precision. We propose a load-balancing mechanism that partitions the high dimensional search space in a balanced manner. We then analyze the performance of our proposed using a variety of metrics with simulation as well as a functional PlanetLab implementation.

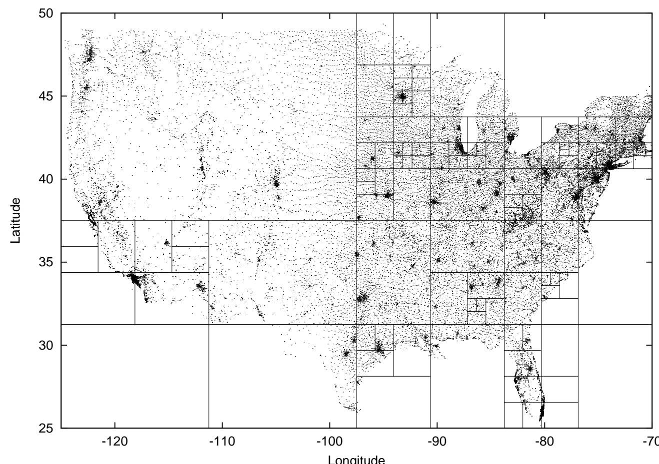


Figure 1: Geographical Distribution of US ZIP Codes.

1 Introduction

The relentless growth of storage density and improvements in broad-band network connectivity has fueled the increasing popularity of massive and distributed data collections. These include multimedia data (music, images, and video), data collected by sensor networks and various types of surveillance devices, various types of scientific data, and medical data. Many of these data sets can be of such massive scale that even their index can easily overwhelm the storage or processing capacity of single nodes. Consequently, distributing the storage and querying of this index data across many nodes becomes necessary. The geographical scale of such distribution can range from tightly coupled cluster systems connected by system-area networks in a machine room to peer-to-peer storage systems encompassing many users across a continent. Regardless the distribution scale, we need efficient ways of managing the distributed index. Our goal is to develop an indexing scheme that can meet the following requirements.

Efficient support for similarity-search and range queries for high-dimensional data. We would like to be able to quickly find a set of objects that are similar to the target object specified by a query within a high-dimensional feature space. We would also like to be able to perform range queries that find objects that are contained within a high-dimensional region specified by the query. These operations are widely used in many applications, including data mining, decision support, pattern recognition, and even text document retrieval. Existing solutions, such as distributed hash tables, that use hashing to perform object location are not suitable for supporting complex queries such as similarity-search and range queries [15, 10]. Furthermore, even in centralized systems, high dimensional similarity search is hard due to the well known phenomenon known as *dimensionality curse* that limits the performance of these operations.

Support for multiple indexes. The distributed infrastructure should be able to support multiple data sets, with different dimensionalities and different distributions. One cannot expect to have each data set be spread across all the nodes, nor is it reasonable to have each data set be stored on a separate node. The system should be able to support a large number of small data sets, each of which is spread over a small number of nodes.

[*] Princeton University, {chizhang,rywang}@cs.princeton.edu.
[†] Yale University, arvind@cs.yale.edu.

Load-balancing. Each node that stores a partial index should be responsible for roughly the same number of objects. The distribution of objects in a high-dimensional feature space can typically be highly non-uniform, as evidenced in the example shown in Figure 1. (In this example, the latitude-longitude coordinates of zip locations constitute the 2D feature space, and the grid patterns superimposed on the map represent an attempt of partitioning this space in a load-balanced fashion. We will discuss later why this attempt is considered not particularly successful.)

Locality. Index data for related or similar objects should be concentrated on a small number of nodes, allowing the execution of range queries or approximate queries to limit the number of affected nodes, while improving the parallelism of supporting multiple independent queries.

Support for scalable overlay networks. The indexing scheme should be deployable over a large-scale peer-to-peer overlay network. Each node should only need to maintain a small amount of state, while still allowing queries to originate from any node and to progress rapidly towards the target nodes that can conclusively answer the queries. Nodes should be able to freely join and leave from the overlay without disrupting the operation of too many remaining nodes.

Existing indexing schemes cannot meet *all* the challenges enumerated above. In this paper, we introduce an indexing system called *SkipIndex*. The system includes the following key mechanisms. (1) The system partitions the search space in a hierarchical tree manner, and organizes the leaf partitions using a Skip Graph-based [3] distributed data structure. Even though the underlying Skip Graph only supports one-dimensional keys, our resulting organization supports high-dimensional range and similarity queries while requiring only a logarithmic number of peer pointers and a logarithmic number of overlay hops. (2) The system provides an approximate query mechanism, where the user gets to specify the desired level of search accuracy and the system intelligently controls the number of nodes interrogated to satisfy the error-bound. (3) Diverse data sets with differing dimensionalities and distributions can be stored in the distributed infrastructure at the same time. (4) The system allows the high-dimensional feature space to be partitioned dynamically among participating index nodes in a load-balanced manner. We believe that the SkipIndex approach satisfies *all* the requirements enumerated earlier in this section.

2 Distributed Index Organization

In a distributed index infrastructure, each node maintains the index for some subset of the keys currently stored in the system. Nodes in the system also maintain a small number of links to other nodes, forming an overlay mesh. Index construction and query processing need to be performed in a completely decentralized manner with any participating node having the capability to insert new data points into the index, perform a query on behalf of a client or a peer node, and propagate queries to its peers in the overlay network. We begin by discussing existing alternatives for partitioning the search space and searching distributed partitions. We show the inadequacy of existing schemes and outline our design of SkipIndex.

2.1 Search Space Partitioning Methods

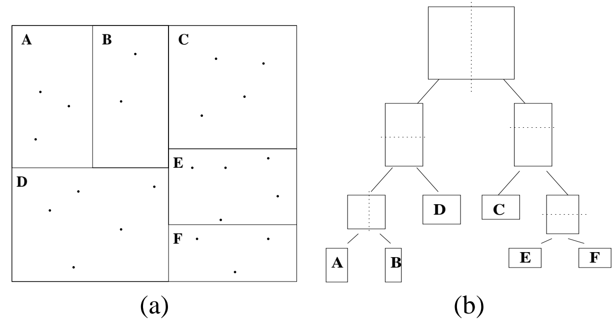


Figure 2: Partition of Search Space

2.1.1 Content Addressable Networks (CAN)

CAN uses a virtual representation of a d-dimensional Cartesian coordinate space to provide distributed hash table (DHT) functionality [20]. Every processor on the network knows the **zone** in space that it "owns." To join the network, a processor may simply pick a random point in space and identify the peer who is responsible for the space that surrounds that point. The zone currently owned by the peer is split into two equal halves, with the splitting dimension chosen using a predetermined order. One of the two resulting halves is then assigned to the new processor. Since the processor joining the system picks a random point in the d-dimensional space, CAN provides probabilistic guarantees on the size of the zones assigned to processors.

Keys are then mapped to zones and are stored by processors that own them. When CAN is used as a traditional DHT, keys are hashed to points in the d-dimensional space (and stored in the processors owning the corresponding zones) to provide probabilistic load-balance guarantees on the number of keys stored at each processor. When presented with an exact query, the system applies the same hash function on the queried key and routes the query to the processor maintaining the zone surrounding the target point. Each processor maintains a routing table that comprises of the processors that own adjoining zones. To route a query, a processor looks in its neighbor table and determines the peer who is closest in Euclidean distance to the destination point, and then forwards the request accordingly. When the d-dimensional space is partitioned evenly across processors, the average number of neighbors for a CAN processor is $O(d)$, and the average routing distance in an N-processor CAN is $O(\sqrt[d]{N})$. Exact queries are thus handled efficiently by the system [20].

The hash function, however, destroys the logical integrity of the keyspace, making it difficult to efficiently support complex similarity searches and range queries [15, 10]. An alternative strategy, in order to support complex queries, would be to refrain from using hash functions for mapping keys to regions of space, but instead simply map a key to a point in the coordinate space based on some simple function that preserves the logical ordering of keys. The resulting ability to

support complex queries however comes at a high price: processors that happen to own densely populated zones would incur high storage costs and be subject to a heavy query load, while some other processors can be virtually idle.

2.1.2 Sampled CAN

To overcome the load-balance problem associated with storing unhashed keys in CAN, pSearch [25] uses the following sampling-based method to assign regions of space to processors. A joining processor, instead of picking a random point in space, chooses the key of some sampled object that currently exists in the system. It then obtains a region as before by locating the peer that stores the chosen key and splitting the peer's region into two halves. By choosing the key of an existing object, densely populated regions are more likely to be partitioned by joining processors, and the resulting data distribution will be more balanced than that obtained by choosing a random point. However, this sampling technique has its limitations. The joining processor uses only one sampled key, which may not accurately reflect the overall key distribution. Also, as the system accumulates more keys, the data distribution could change, resulting in load imbalances. The grids in Figure 1 are generated using a sampled partitioning. The load imbalance is visible in many grids cells.

2.1.3 Balanced Region Allocation

pSearch's sampling technique improves load-balance, but its effectiveness is hobbled by the fact that the zones are partitioned at join time using just one sample. We next consider a technique that strives to achieve better load balance.

Initially, there is only one processor in the system, which controls the entire keyspace. As we add more keys into the system and once the storage exceeds the threshold of the number of objects to be allocated to each processor in the system, the initial processor splits the keyspace along one of the dimensions into two non-overlapping regions containing an equal number of keys and hands over one of the regions to an idle processor. The dimension with the maximum span is chosen as the splitting dimension. The partitioning process continues to form a binary tree. Figure 2(a) shows an example where a 2D space is partitioned across six processors.

The above scheme is clearly over-simplified for a dynamic peer-to-peer setting, as it relies on static notion of load threshold and an idealized pool of idle nodes. Appropriate run-time refinements to the space partitions are required for a dynamic setting. We address such issues in Section 4.2, where we build on this basic scheme. We can however use the simplified version proposed here to understand the implications of load-balanced CAN systems.

2.1.4 Comparisons and Discussions

Figure 3 (a) graphs the load distribution of a sample dataset under the partitioning strategies discussed above. The dataset used for this experiment is a 20-dimension color histogram data described in Section 5. Both CAN and sampled-CAN strategies exhibit worst-case loads that are one to two orders of magnitude higher than the balanced region allocation scheme.

We also evaluate the different techniques in terms of the amount of routing state maintained by each processor as the means to measure scalability. A scalable peer-to-peer system must be able to operate without global knowledge regarding many other nodes and their data partitions. The number of neighbors maintained by each processor reflects the cost of periodical peer probing and routing repairs when processors join and leave the system. As node joins, departures, and failures are routine events in a large peer-to-peer system, limiting the number of neighbors becomes an essential requirement. The amount of routing and query processing load is also proportional to the number of neighbors. The "hub" processors with high neighbor counts not only incur high maintenance cost, but could also exhibit high routing and query load and limit the throughput of the system. Thus we also use the maximum number of neighbors maintained by some processor as a scalability metric for the systems.

CAN, with its random choice of a point for joining processors and even splitting of node space, achieves an almost uniform distribution of space across processors, thereby limiting the maximum neighbor count. As we try to improve its load-balancing with sampling and balanced partitioning, the variants show significant problems in the number of peers. The reason is that the load-balanced schemes partition the space according to application key distribution, which can be highly skewed. The hyper-rectangles that result from such uneven partitions are highly likely to interleave with each other, thus breaking CAN's promise of $O(d)$ number of neighbors.

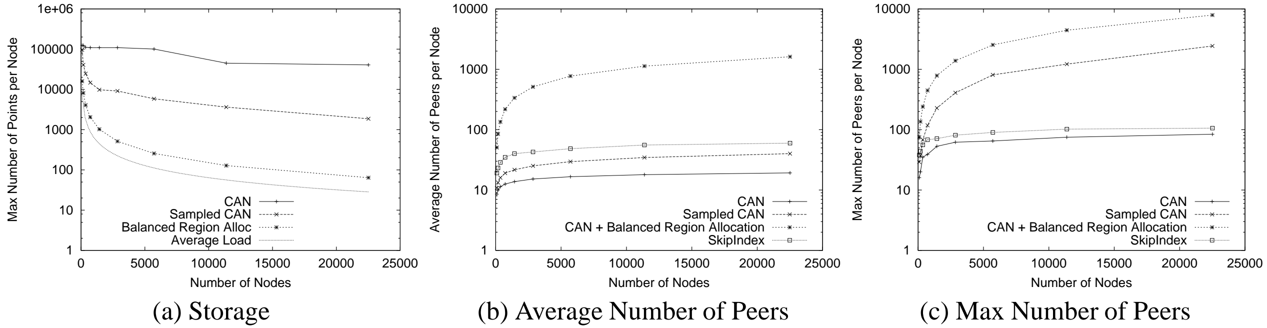
We have now run into a fundamental conflict between load-balance and neighborhood state for CAN-based schemes. We will show in the experimental results section that both load-imbalance and high neighborhood state translate into high query processing loads on some of the processors in the system. We need an indexing mechanism that addresses both issues simultaneously.

2.2 Hierarchical Partition Organization

Hierarchical multi-dimensional tree structures have received extensive research in the database community, such as K-D tree [6], R-tree [9], R*-tree [5], X-tree [8] and many other structures. These approaches hierarchically partition the search space and data set into smaller and smaller regions. Insertion and search operations navigate the tree from the root down to appropriate leaf nodes. We examine whether these hierarchical techniques can be adapted to build efficient distributed indexing systems.

2.2.1 Distributing Search Trees

One could build a distributed index infrastructure by distributing portions of a hierarchical search tree data structure across different processors. Tree nodes, representing regions of space, would be distributed objects containing pointers to other tree nodes. Non-local pointers of the tree structure would have to be represented as generalized global addresses that specify both the processor ID and the local address of the tree node within the processor. Navigating the tree structure in order to perform insertions or queries would require inter-processor messages.



(a) Storage (b) Average Number of Peers (c) Max Number of Peers

Figure 3: Storage and Routing State Balancing in Different Partition Schemes

The fundamental manner in which distributed search trees differ from CAN-like system is how routing is performed to insert or query objects. In a distributed search tree, all operations follow the hierarchical tree pointers to reach the target tree node. CAN-like systems instead perform cartesian coordinate routing to find the target region. This means that the routing state (or the neighborhood information) maintained by a distributed search tree is strictly a function of the outdegree of tree nodes, which is bounded in all of the hierarchical schemes, and is not dependent on the data distribution.

However, binding tree nodes to global addresses significantly limits the ability to perform load balancing and provide fault-tolerant execution. Assigning tree nodes to processors and modifying the assignments when data items are inserted dynamically are not easy tasks any more. Virtualizing the global addresses using a DHT can partially alleviate such problems.

2.2.2 Distributing Search Trees using a DHT

A distributed hash table (DHT) disperses objects in a load-balanced fashion and allows efficient lookup by their IDs. Using a DHT to access the nodes of a search tree would require the system to refer to tree objects by some unique ID rather than a global address. Tree objects can be transparently moved from one processor to another, in order to provide load-balance and handle processor churn.

Ratnasamy *et.al.* [21] outline an approach to support range queries on a one-dimensional keyspace by organizing the keys into a *Prefix Hash Tree*, and distributing the tree nodes using a DHT. Awerbuch and Scheideler [4] propose the general mechanism of combining two orthogonal data structures to provide a load-balanced, range-queriable structure. One data structure supports range query operation over the data keys in the system. The other data structure, which can be any DHT, maps the objects of the first data structure to some processor in the distributed system.

2.2.3 Discussions

Hierarchical search trees result in structures with bounded degree and, in some cases, bounded depth. There are, however, many challenges in deploying them in distributed settings. The tree has to be searched top-down from the root (except for [4]), with each step resulting in a network communication or even a DHT lookup. The processor maintaining the root tends to become a performance bottleneck and is also a single point of failure. In an unbalanced tree, like K-D tree, the search path can be very long, further degrading the search efficiency. Also, care should be taken so that concurrent updates to the tree data structure are performed safely and correctly. The maintenance of a balanced tree, such as an R-tree, could further complicate the concurrency issues associated with distributed updates.

2.3 SkipIndex: Organizing Regions into a Skip Graph

We now outline our approach for storing a multi-dimensional index. We maintain a hierarchical partitioning of space that is constantly refined based on the current set of objects maintained by the system. The partitioning of space can be described by what we refer to as the **region tree**, as depicted in Figure 2 (b). We then associate a one-dimensional key with each region in the system in order to obtain a total order on the regions. This key captures the hierarchical manner in which the region was created. The keys are then used to store the leaf regions in a searchable *Skip Graph*, or equivalently *Skip Nets*, which supports insertion and lookup based on a one-dimensional key. These structures were chosen because they do not use hashing and therefore preserve the logical integrity of the keyspace. Furthermore, given a data point and a leaf region, we can decide whether the region containing the point appears before or after the leaf region in the total order. In order to locate a region containing a data point, the query is routed through the Skip Graph in a manner such that the distance to the target region, measured in terms of the total order, is probabilistically halved in every routing step. The region tree is therefore not used for navigation purposes; instead each processor maintains a partial view of the region tree to aid query processing and to determine the ordering between a region and the destination point. Furthermore, we leverage a number of useful properties of the Skip Graph, such as its ability to perform routing while requiring only a logarithmic number of neighbors even for skewed data distributions. We formalize our approach, which we will refer to as *SkipIndex*, in the following subsections.

2.3.1 Background: Skip Graphs

Our routing scheme is based on Skip Graphs [3], which is a generalization of Skip Lists [19, 17, 16] for one-dimensional range queries. A Skip List offers a randomized alternative to

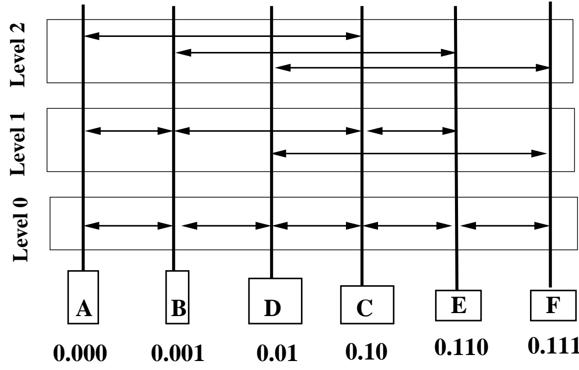


Figure 4: Skip Graph

the more complex balanced-tree data structures, such as red-black trees or b-trees. Each element in a Skip List participates in several levels of linked lists (or "rings"). The lowest level list consists of all elements ordered by their keys. Each key that appears in the list at Level i, would also appear in the list at Level $i+1$ with some probability p. At each level, a key stores pointers to its left and right neighbors. To locate a key, one searches the highest level (which might have just a few keys), dropping down to the more densely-populated lower levels if needed. There are, on average, $O(\log n)$ levels in the system, meaning that a search will traverse $O(\log n)$ keys until it reaches its destination.

Skip Graphs extend Skip Lists for distributed environments by adding redundant connectivity and multiple handles into the data structure. It is equivalent to a collection of up to n Skip Lists (where n is the number of elements in the lowest level) with each element participating in exactly one list at each level and some of the lower levels shared across many Skip Lists. The increased connectivity provides greater fault-tolerance and avoids hot-spots as any element could be located using any one of the top-level elements of the different Skip Lists.

2.3.2 Formal Description of SkipIndex Routing

Before we describe the details, we define some terms that we use in the description.

- **Regions:** A region stands for a node in the **region tree** tree (Figure 2 b). Each region is a hyper-rectangle in the search space. **Intermediate regions** generate two child regions when split along one of the dimensions. **Leaf regions** are associated with physical machines that are responsible for managing that portion of the index.

- **Split history** of a region represents the path in the **region tree** from the root to the region. It is a list of tuples $\langle dim_{split}, pos_{split} \rangle$ with each element specifying the dimension and position of a split.

- **Region code** is a string of 0s and 1s that represents how a region is generated by the splitting process. When we split a region into two pieces, the region code for the left child (which is the region with smaller coordinates in the splitting dimension) is generated by appending "0" to the end of the region code of the parent region. The right child's region code is obtained by appending "1" to the parent's region code. We represent the code in a binary fraction format, with a decimal point to the left of the most significant bit. The ordering imposed by the region codes corresponds to the in-order (or "left-to-right") traversal of the region tree. When combined with the **split history**, the region code completely describes the coordinates of a region.

- **Ordering of a region R and a point p:** Let R_p be the leaf region covering the point p. If $Code(R_p)$ is known, then one can use the following simple checks to determine the ordering between R and p. If R encloses R_p, *i.e.,* $Code(R)$ is a prefix of $Code(R_p)$, we have $p \in R$. Otherwise, if $Code(R) < Code(R_p)$, then we can say that $R < p$, else $p < R$. However, if $Code(R_p)$ is not currently known, we need to use the split history of R to determine where the point p appears in relationship to the various region splits that had occurred to generate R.

  for ith tuple $[dim_{split}, pos_{split}]$ **in** $History(R)$ **do**
  c $\leftarrow$ ith bit in $Code(R)$
  if ($c = 0$ & $p[dim_{split}] \geq pos_{split}$) **then**
  return ($p > R$)
  else if ($c = 1$ & $p[dim_{split}] < pos_{split}$) **then**
  return ($p < R$)
  return ($p \in R$)

We now describe the details of our routing algorithm and the neighborhood state maintained by each processor in the system.

We organize the leaf regions into a Skip Graph using their region codes as the keys. Figure 4 illustrates the region codes assigned to the regions depicted in Figure 2 and the resulting Skip Graph organization of the regions. A node responsible for a leaf region maintains the following neighbor state that is used during routing queries: a) For each level of the Skip Graph, a node owning a region maintains the identity of nodes that own adjacent regions. b) For each level of the Skip Graph, a node owning a region also maintains the split histories and region codes of adjacent regions.

When a node owning region A is posed with a query to locate the leaf region enclosing a target point p, it performs the following routing actions:

Algorithm 1 SkipIndex Routing

```
if ( p ∈ A ) then
  A is the destination region
else if ( p > A ) then
  /* move right */
  for i = max-level down to 0 do
    if ( ! right_region[i] > p ) then
      forward to right_neighbor[i]
else /* move left, omitted */
```

In other words, the node owning region A examines its neighboring regions, starting with the highest level of the Skip

Graph, and routes the query to its farthest neighbor without overshooting the destination point. As the query nears the destination point, the routing algorithm uses progressively lower levels of the Skip Graph. The routing distance and the number of peer links are both bounded as $O(\log N)$ in an N-node Skip Graph. These bounds are independent of the data distribution and can be achieved even when the region sizes are highly skewed (as supported by the experimental results shown in Figure 3 (b), (c)).

In addition to being able to achieve efficient routing and low neighborhood state, the SkipIndex approach has other useful properties that makes it suitable for performing nearest neighbor queries. Note that all the **leaf regions** obtained from recursively splitting a single **intermediate region** are contiguous in the lowest level of the Skip Graph. This proximity enable us to broadcast a query to all **leaf regions** obtained from a single **intermediate region**. Later we will use this operation for performing range queries and nearest neighbor queries.

We use a dynamic version of balanced region allocation in SkipIndex. A node brings an idle node to offload its region when overload is detected. The details of how to decide overload and how to generate an idle node is discussed in Section 4.2. The new node obtains from the splitting node both the split history and the portion of the data keys that are to be assigned to the new node. After the split, both nodes modify their split histories and region codes. Specifically, the splitting node retains the lower half of the original region and adds a "0" to its region code, while the new node obtains the upper half and appends "1" to its region code. Since we organize the Skip Graph based on the binary fraction value of region codes, the splitting node does not change its key value during this process, and the new node becomes its right neighbor in the lowest level of the Skip Graph. The new node joins the Skip Graph with its region code. We detail the join process in Section 4.4. After insertion, the splitting node also notifies its peers about changes to its split history and region code.

2.4 Discussion

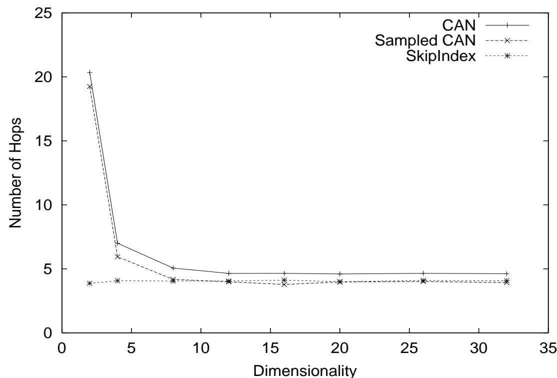


Figure 5: Routing distance changes as the dimensionality increases

Figure 5 shows the routing distance for varying dimensionality. CAN and Sampled CAN only achieves performance comparable to SkipIndex when dimension is above 8. Figure 3 (b), (c) compare the number of peers used by these systems. We compare these systems again in the experimental results section regarding measures such as the distribution of query processing loads.

In summary, the issues of balancing storage and query load, minimizing the size of the routing table, and lowering the cost of maintaining the overlay, are important in a massively distributed system. These properties are not easy to achieve simultaneously in CAN-style systems. SkipIndex exhibits logarithmic bounds on routing state and routing distance. These bounds are independent of data distribution and dimensionality, making it superior to CAN routing and its variants.

3 Distributed Query Processing

SkipIndex supports several types of queries: point query, range query, and k-nearest neighbor query. Point query is equivalent to routing the query to the node owning the enclosing region (which was discussed in the previous section) and performing a local search. In this section, we therefore focus on the other two types of queries.

Range query can be defined as returning all of the data points falling within a given hyper-rectangular or hyper-spherical region. The query range can also be specified by other shapes, such as hyperellipsoids that result from using a distance metric with different weights for different dimensions. These queries can be supported efficiently with a range-limited "multicast" operation, which dispatches the query to all the nodes whose regions intersect the query range. In SkipIndex, the multicast operation can be implemented in a logarithmic number of steps even when a large number of nodes maintain regions that intersect with the specified range.

3.1 Range Query by Multicasting

In a centralized index tree, like K-D-Tree or R-Tree, a range query is implemented using a top-down tree traversal. Starting from the root of the index tree, the algorithm recursively visits every sub-tree whose bounding region intersects the query range. SkipIndex performs tree traversal in a similar but distributed manner, without requiring any node to maintain the complete view of the index structure.

Each node maintains a **partial view** of the **region tree**. The **partial tree view** or **local tree view** of a node comprises of the split histories of its local region and that of the regions maintained by its Skip Graph peers, where each split history provides information about the path from the tree root to a leaf region (as discussed in the previous section). Figures 6 (b)-(d) illustrate the partial views of three index nodes.[1]

There are three types of "leaves" in the partial tree view of a node: the local region of the node itself, the regions of its peers, and the "obscured" regions corresponding to unknown parts of the region tree. We call the latter two types **remote regions**. Note that an obscured region can be either a single leaf region or a group of leaf regions; the local node does not

[1] In order to simplify the illustrations, we draw only those peer nodes that are adjacent in the bottom-level Skip Graph (shown in Figure 6 (a)). Partial views actually contain the split histories of *all* neighboring regions that are adjacent to the node in any one of the levels of the Skip Graph.

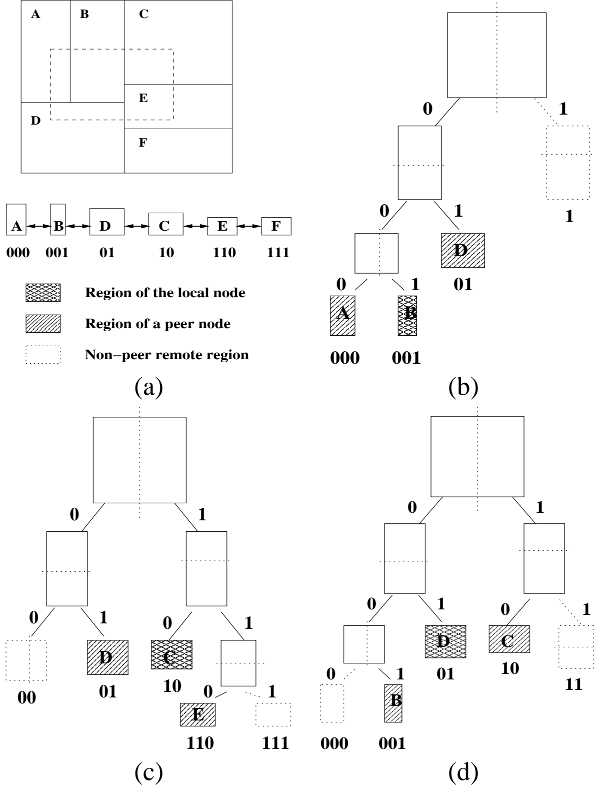


Figure 6: Partial tree views of different nodes

have sufficiently detailed information regarding these regions. For example, the structure of regions C, E, and F are not visible to the node maintaining region B; instead they appear collectively as a single obscured region.

The range query starts by routing to the center of the query range, *qrange*. The node enclosing the center does a local range query, then traverses its partial tree view to find out which other regions intersect *qrange*. This is equivalent to the tree traversal performed on a centralized index tree, except that it returns a set of regions to be searched on remote machines. Then the query is multicasted to each region in the set to continue the search.

For an obscured destination region, *dregion*, we route a message to some node that indexes at least a portion of *dregion*. This node can, if necessary, further decompose *dregion* into finer regions and forward a message to the appropriate nodes. The messages includes both *qrange*, the original query range, and *dregion*, the region which the receiving node should further decompose to forward the message. When a node forwards the message to other nodes, it specifies non-overlapping *dregion* values in order to ensure that no duplicate messages are generated for a given query.

A more formal description of the multicast process is described below. We use $r \cap qrange$ to denote the check as to whether region r intersects the query range, and $r_1 \subseteq r_2$ to denote the check for region enclosure.

Although we use the analogy of region tree traversal in our informal descriptions, the multicast process detailed above

Algorithm 2 Range-Limited Multicast

Upon $node$ receiving query$\langle qrange, dregion\rangle$:
if $local_region(node) \subseteq dregion$ **then**
  perform local query (qrange)
for all remote regions R in partial tree view **do**
  if $R \subseteq dregion$ **and** $R \cap qrange$ **then**
    forward query$\langle qrange, R\rangle$ **to** R

does not descend the tree sequentially, but rather "jumps" into subtrees of the region tree. The depth of multicast, *i.e.*, the maximum number of hops to reach a leaf region that belongs to the target area, does not depend on the height of the region tree, which can be $O(N)$ in the worst cast. Instead, in order to multicast a query to a given region, the multicast process forwards the message using links in the Skip Graph, and each step of this forwarding process achieves one of the following two goals: a) The distance to the set of nodes representing the region is halved, or b) the destination region is decomposed into subregions, each of which is at most half the size of the destination region. This intuition is formalized in Theorem 1 in Appendix. When the query range is small, *i.e.*, only a few neighboring regions are involved, multicast depth is typically O(1), as those regions are already included in the partial tree view of the initial node.

3.2 Nearest Neighbor Query Algorithm

k-nearest neighbor search is similar to a spherical range query, except that the radius of the query ball is not prespecified but is instead determined dynamically during the search. Our k-nearest neighbor algorithm is therefore based on the range query algorithm, which is enhanced with a demand-driven process for determining which regions to query.

As the first step, we route the k-nearest neighbor search to the node A owning the region that contains the query point q. A then executes a local query to determine an initial candidate set of k-nearest neighbors, denoted by S_{knn}. The maximum distance from q to points in S_{knn} is the initial value for the KNN query radius, R_{knn}. If the size of S_{knn} is less than k, then R_{knn} is set to a value that covers the entire d-dimensional space. A also maintains a priority queue, Q_{search}, of regions to be searched, ordered by their minimum distances to the query point. The initial contents of Q_{search} is determined by traversing A's partial tree view and finding all regions that intersect with the current query ball $\langle q, R_{knn}\rangle$.

The algorithm queries the regions in Q_{search} in increasing order of distance. In each step, A extracts the minimum distance region, R, from the queue and sends a query message $\langle q, R_{knn}, R\rangle$ towards region R. When this message is received by a node, which could have more detailed information about R, the search region is refined based on the node's partial tree view and forwarded towards the sub-region of R that is closest to q. When the search region is eventually refined to a leaf region and received by the corresponding node, a local query is performed to determine points contained in the region that are closer than the current KNN distance estimate

R_{knn}. The results are reported back to A along with the newly discovered sub-regions that intersect with the query ball. A then updates S_{knn} and R_{knn}, and inserts sub-regions found during the step into Q_{search}. A repeats the query step until there are no regions left in Q_{search} within distance R_{knn}. Algorithm 3 provides the formal description.

Algorithm 3 Basic KNN query algorithm

A does a local KNN search and initializes S_{knn} and R_{knn} based on the results
A traverses its partial tree view and initializes Q_{search} with all regions R, such that $mindist(R, q) \leq R_{knn}$
while (Q_{search} not empty) **do**
  $R \leftarrow$ extract_min(Q_{search})
  forward query$\langle q, R_{knn}, R\rangle$ **to** R
  receive result$\langle Set_p, Set_R\rangle$
  insert regions in Set_R into Q_{search}
  update S_{knn} and R_{knn} with Set_p
  prune Q_{search} with new R_{knn}
return S_{knn}

Upon node B receiving the query$\langle q, R_{knn}, R\rangle$:
if ($local_region(B) == R$) **then**
  $Set_p \leftarrow K$ nearest points within R_{knn}
  $Set_R \leftarrow$ leaf regions inside R from local tree view
  reply to A the results $\langle Set_p, Set_R\rangle$
else
  find the leaf R_l in local tree view with minimum $mindist(q, R)$
  forward query$\langle q, R_{knn}, R\rangle$ **to** R_l

This algorithm sequentially queries leaf regions in ascending order of their minimum distances to the query point. Thus, it is optimal in the number of machines searched. In fact, our distributed algorithm is inspired by the BBKK local query algorithm proposed by Berchtold *et al.* [7], which minimizes the number of index pages visited during a nearest neighbor query on a centralized index. However, the sequential nature of this query may result in higher latencies. This drawback could be addressed by initiating searches on the M closest regions at a time, instead of searching just one region in each step, thereby increasing the amount of parallelism. This improves the search latency at the expense of making some unnecessary queries.

3.3 Approximate Search

It is difficult to limit the scope of nearest neighbor searches for high-dimensional datasets. Many previous studies [26, 7] have shown that as the dimensionality is increased, more and more of the bounding regions in index structures intersect with the query sphere, eventually degrading the search into scanning the whole data set. This is referred to as the *dimensionality curse*.

For many applications, the dimensionality curse can often be alleviated by using approximate search due to the following reasons.

First, many applications, such as content-based retrieval of multimedia datatypes, can tolerate small inaccuracies in query results. Furthermore, most distance metrics used in these applications are of heuristic nature in the first place, and providing slightly inaccurate results might not have serious ill-effects.

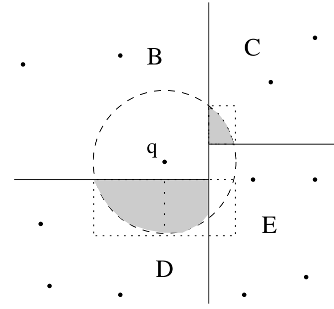


Figure 7: Intersection of Query Sphere and Regions

Second, though the distance to the k nearest neighbor, and hence the query radius, increases steadily with dimensionality, the volume of the query sphere typically does not increase. A majority of the bounding regions intersect the query sphere with only one of their corners. These intersections are, therefore, typically negligible in volume. If the data points are uniformly distributed within the local range of the query, these tiny regions of intersection could be omitted as they hold little chance of yielding data points.

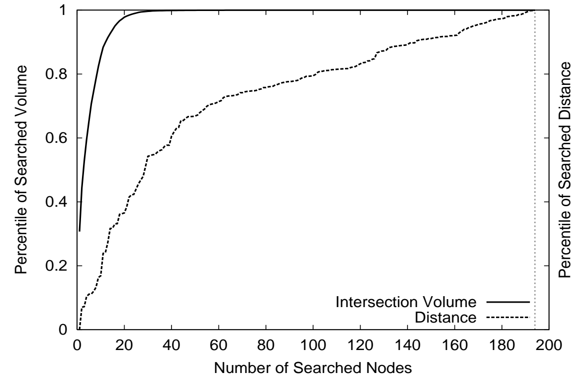


Figure 8: CDF of intersection volumes of searched regions and their minimal distances to the query point.

Based on these observations, we propose a method to meet a precision bound for queries that is based on the volume of the searched region. Once the volume of the explored region within the query sphere exceeds the desired precision, the search is to be terminated. Figure 7 shows a query sphere centered at point q intersecting four rectangular regions. Since the intersection of the sphere with C is of negligible volume, we can skip querying C without sacrificing search accuracy.

To characterize the importance of this optimization, we examine the distribution of the intersection volumes while performing a 10-NN query on the PHOTO dataset (which is described in the experimental results section). Figure 8 shows

the cumulative intersection volume from regions intersecting a query ball. Although the total number of intersecting regions is large, a majority of these regions share negligible intersection volumes with the query sphere.

Another approach to approximation is provided by the ϵ-approximate nearest neighbor search [2, 13], which returns points within $(1 + \epsilon)$ distance of the exact nearest neighbors. We do not use this scheme in our system because of the following property of high-dimensional rectangles: A corner of a high-dimensional rectangle could encroach significantly into the query sphere, but the corresponding intersection could still be negligible. The ϵ-approximate nearest neighbor search methods would have to query these low-overlap regions. This effect is also illustrated in Figure 8, where a gradual increase in distance actually corresponds to a dramatic decrease in intersection volume. Distance-based approximations are therefore likely to be too conservative to significantly reduce the number of queried regions.

3.3.1 Estimating Intersection Volume

It is, however, quite challenging to calculate the intersection volume of a hypercube and a hypersphere in high dimensional space. We take a conservative approach that calculates the upper bound on the intersection volume. Consider the intersection of C with the query sphere in Figure 7 and a rectangular bounding box (BB) that encloses the intersection. The ratio of the volume of the intersection volume to the volume of its bounding box is upper-bounded by the ratio of the volume of the entire sphere to the volume of the sphere's bounding box. Since the bounding box of a sphere with radius R has volume $(2R)^D$, we get the following volume bound:

$$V_{intersect} \leq \frac{V_{sphere}}{(2R_{knn})^D} * V_{BB}$$

V_{sphere} is simply a function of R and D. V_{BB} can be computed by determining the points of intersection of the hypercube and the hypersphere. It is therefore possible to use the bound to calculate the upper-bound on $V_{intersect}$. The proof can be found in Appendix, Corollary 1.

The intersection of the sphere with regions D and E are a bit more complex to bound, as they do not meet the premise of our upper bound. These regions overlap with more than one quadrant of the query sphere. In such cases, we extend the region so that its corners do not lie inside the sphere, break up the extended region into different pieces so that each piece intersects with the query sphere in exactly one quadrant, and then use the upper bound given above to calculate the intersection volume generated by each one of the pieces. (In Figure 7, region D is extended to form the dashed box that encroaches into region E and then broken down into two pieces.)

3.3.2 Early Termination of Approximate Search

Assume that each nearest neighbor query specifies an error bound ϵ. The query is then said to tolerate results where up to ϵ fraction of the reported nearest neighbors are non-optimal. Assuming uniform data distribution around the query point, we are allowed to disregard ϵ fraction of the query sphere. We modify the nearest neighbor algorithm to keep track, for each region in Q_{search}, the extended bounding box of the intersection of the region with the query sphere. The search termination condition is then modified to:

$$\sum_{\forall R \in Q_{search}} (V_{EBB(R)}) < \epsilon * (2R_{knn})^D$$

3.4 Optimizations

Many of the traditional optimization techniques used by centralized indexes could also be applied to SkipIndex. For example, we could use the minimum bounding rectangle of all points stored in a node as its region, instead of the whole subspace generated by splitting its parent region. This can improve the estimation of both the minimum distance to a query point and the intersection volume, thus reducing unnecessary queries. To fully exploit this optimization, there needs to be some background exchange of bounding box information amongst peers.

An even more aggressive background preprocessing is proposed in pSearch [25], to allow a node to sample some points stored in its neighbors that are closest to its region and store them locally. Such a scheme could be integrated with data replication strategies that are designed to improve availability. The data on a node can be replicated on its *close buddies* (discussed in Section 4.5). The replicas may answer queries on behalf of the original nodes, in order to reduce communication costs.

4 Building an Index Service

In this section, we discuss how to use SkipIndex to build a scalable and robust index service for peer-to-peer platforms. We address the problems of index diversity, dynamic load balancing, network proximity and maintenance. We also briefly discuss our prototype implementation developed on PlanetLab.

4.1 Accommodating Diversified Indexes

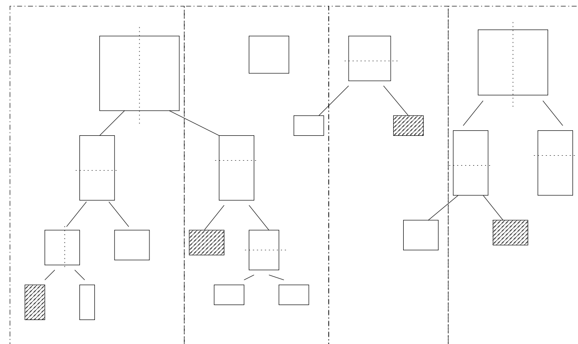


Figure 9: Partitioning a Forest of Indexes

Till now, we have been focusing on the organization of a single index dataset. In practice, it is hardly the case that one monolithic data set is indexed across a widely distributed system. The underlying peer-to-peer structure implies the accommodation of numerous indexes highly diversified in their

scale, dimension, distribution and query patterns. Distributing each index to all the nodes with separated routing structure is certainly out of the question, as the amount of routing state will be linear to the number of indexes. Partitioning the nodes for each index is possible. But then one needs to develop algorithms for partitioning, and might also introduce a level of indirection to locate the node set for each index.

We extend the SkipIndex described above to accommodate multiple indexes. Figure 9 illustrates the basic idea. The indexes are ordered by their ID. So their region trees form a forest. Now we can use SkipIndex to organize the leaf regions in the forest order, with the keys generalized to a two-tuple comprising of $\langle indexID, treecode \rangle$. With large number of indexes, we combine neighboring tree or branches into one node, as depicted by the dashed rectangles in Figure 9. The key of each node is given by the key of the left-most leaf maintained by the node, which is shown as a shaded region.

Several properties of this scheme are worth noting here:

- The routing distance and the amount of routing state are independent of the number of indexes. They only depend on the number of nodes.
- Dimensionality of an individual index is irrelevant to the routing structure.
- Moving a part of index on one node across boundary to its neighbor changes the key of this node, but does not change the Skip Graph structure.
- Removing a node will shift its responsibility to the left neighbor.

The first property ensures the scalability of SkipIndex. The second provides the diversity of data sets. The last two properties aid the load balancing and failure recovery mechanisms of SkipIndex. In the next section, we present a dynamic load balancing scheme to complement the balanced region allocation scheme for SkipIndex.

4.2 Dynamic Load Balancing

In Section 2.1, we introduced the notion of overload-triggered splits as a means of balanced region allocation among SkipIndex nodes. In a real index system, there are two dynamic issues to be addressed in the partitioning: how to decide the overload threshold in a distributed fashion and how to find idle nodes.

We do not consider the approaches using centralized management or global consensus protocols for the obvious reasons of scalability and availability. Our solution exploits pairwise gossiping to spread information about overloaded and underloaded nodes. Then each node makes their own decision on how to split. The idle nodes are created by a combination of node virtualization and compaction.

SkipIndex nodes keep coarse load information about other nodes. Periodically, a node talks to its neighbors about its current load and the k highest and lowest loaded nodes it knows. The load gossips comes with a timestamp from the original host to control consistency. Upon receiving a gossip message, a node remembers the entries with timestamps newer than local versions. If an entry bears an obsolete timestamp, the current value will be fed back to the sender in the next gossip exchange. Our gossip protocol relies on the fact that Skip Graphs have a high expansion ratio [3] and information regarding load can be quickly propagated through the Skip Graph links.

The overload threshold is locally decided by each node as: MAX { k-th highest load, k-th lowest load*C, T }, where T is a fixed threshold as a lower bound for overload, and C is a stabilizing constant greater than 1.

When a node detects that its load is above the threshold for a sustained time period, it invokes the partitioning procedure. The node to which it will offload data is chosen as follows:

- If one of the closest neighbors is underloaded, choose it.
- Else if there is an idle node in local view, choose it.
- Else if there is an underloaded physical node with less than V virtual nodes, create a virtual node there.
- Otherwise, pick an underloaded node, make it idle by moving its data to its neighbor (compaction).

The choice of creating a virtual node or compacting an existing node depends on the amount of routing state allowed per physical node. A machine with many virtual nodes can harbor a large number of peer connections, increasing the cost of peer maintenance and danger of disconnection during node failures. Compaction, on the other hand, incurs a higher latency to transfer objects. In practice, compaction can be done proactively in background to maintain a pool of idle nodes at any time.

4.3 Physical Proximity

A major concern for peer-to-peer systems is the performance of multi-hop routing. Several DHT schemes [22, 20, 11] address this problem by taking advantage of physical network proximity. We use a similar heuristic to improve SkipIndex routing locality. When a node splits, it picks the physically closest one from the top k lightly-loaded nodes. The physical network distance can be measured using direct ICMP pings or landmarks [20].

The implication of this heuristics is that lower levels of Skip Graph peers tend to be close to each other as a result of proximity-aware splitting. Higher level peers are more randomly distributed as they are decided by random membership codes.[2] Query operations benefit from this proximity as they crawl the neighboring regions around a query point, which are often found to be contiguous in level-0 SkipIndex routing table. Insertion operations benefit less as most of the hops are through higher level links. Simulation results on Internet topology (Figure 18) confirms this observation.

4.4 Node Join

A new node joins a SkipIndex in two steps. First it "attaches" to some existing active nodes as an idle node. The active nodes can publish such attached idle nodes as having zero load in their peer-wise gossips. Later, an overloaded active

[2] Pastry and Tapestry display similar characteristics with the first few hops corresponding to highly local links, while the latter hops degenerate into higher latency connections.

node draws the idle node into active service through an atomic join protocol.

The first attaching step does not involve any consistency issues. The new node randomly picks some random Skip Graph ring membership codes, and routes *Attach* requests to nodes having longest prefix match with these codes (as in [11]). We use membership codes instead of Skip Graph key because the latter is not uniformly randomly distributed. The nodes accepting the idle node are responsible for publishing gossips about this node, and periodically checking its liveness.

Inserting an idle node into active skip graph can be tricky when we have to deal with concurrent joins as well as other routing activities. Our insertion algorithm is similar to that in [3]. Insertion starts at level 0 by routing a $Join$ message to the node with region code closest to the new node. In case of SkipIndex node split, the splitting node is the starting point because the newly spawned region is to the right of the splitting node in the region tree. This node forwards the join request to other nodes in the level that should be peer with the newcomer. After level 0 peers are established, the joining node traverses the level 0 ring to find out level 1 peers. This process continues until it reaches the highest level where there is no other nodes in the ring. It has been shown in [3, 11] that the joining process takes expected $O(\log N)$ steps and $O(\log N)$ messages.

When nodes join concurrently, their messages may compete to set the peer links of an existing node. The one arriving first sets the peer in the routing table, which may later be replaced by another closer node. In this case, the joining node will send a message to the earlier requester as it should peer with it on the other side.

We augment the above insertion algorithm with an atomic protocol similar to two-phase commit. The purpose is to avoid interference to normal routing in case of race conditions and failures during the insertion process. The active nodes record the joining party, but do not use it for normal routing until receiving signal of successful commit from the joining node. If the joining node fails before finishing its join, the record expires eventually. The node can send the success signal as soon as the two closest level 0 peers are established successfully, since Skip Graph routing is fully functional with just first level pointers.

4.5 Failure Recovery

We consider several cases of routing repairs in our system: a) a node quits voluntarily, b) it fails without warning due to node crash or network disconnection, or c) the network link connecting two peers fails.

In order to repair the index structure after a node disappears, we maintain a set of "backup" nodes for each machine. The backups are chosen from the closest nodes in the Skip Graph and are referred to as **close buddies**. In the prototype, we use a dense routing table [11], so the first level peers are sufficient for choosing close buddies. A background operation replicates data among nearby buddies. The replicas not only improve availability in case of failure, but also help to reduce query delay as an idle node can serve queries pertaining to its busy buddy.

When a node leaves voluntarily, it transfers its region to the closest buddy, and then informs all of its peers about its departure. Failure or disconnection of a node is detected by periodic gossip messages. When a peer continuously misses several gossips, the node diagnoses whether it is a link fault or a node failure by routing several probes through other peers. If any of the redirected messages reaches the destination, the link is labeled as path failure and any later communication will go through the detour.

If none of the probes are successful, the node is declared dead and removed from the routing table. Any later messages toward the region of the failed node will fall to its closest buddy on its left side. So this node naturally takes over the responsibility. This is simpler than the recovery scheme in CAN, which creates a virtual node to take over the fault zone and establishes new peer links as it uses zone neighbors in routing.

4.6 SkipIndex Prototype

We implemented SkipIndex and deployed the prototype on Planet-Lab [18]. We use the dense routing table design proposed by Skip Nets [11]. Instead of just storing two (left and right) peers per level of routing table, this design stores $2(k-1)$ closest peers continuous in the ring. Overall, the routing distance is improved to $O(\log_k N)$. Current implementation includes features of similarity search, range multicast, peer-wise heart-beat and gossiping of load information, dynamic partitioning and support for concurrent node joins. Continued work includes failure recovery and network proximity-aware partitioning.

5 Experimental Results

We have performed extensive experiments to evaluate the query performance of SkipIndex and other schemes. We now present results obtained using a data set called **PHOTO**, which contains 32-dimension color histogram vectors extracted from one million color images downloaded from online photography sites. We perform a dimension reduction transform (PCA) on the raw feature vectors to generate keys whose dimensionality is varied from two to twenty in our experiments. We also tested the system with another large data set (**SOUND**) containing two million short-duration sound frames obtained from a library of sound-effects used by moviemakers. This dataset is indexed using keys comprising of 29 features. The results for this dataset showed trends similar to that of the PHOTO data. Without specific note, all results reported here are generated with the PHOTO dataset.

We implemented the various algorithms described in this paper within a simulator and a real implementation. We simulated the routing and query algorithms with up to 25,000 nodes using our simulator. We also report on the results from the SkipIndex prototype deployed on Planet-Lab, utilizing up to 105 nodes distributed in North America. We also simulated a 314 nodes ISP topology generated by RocketFuel [23], to make measurements on query latency for larger realistic topologies. We measured various performance metrics, such as the query/insertion latency, number of nodes visited during a query, the number of queries handled by each node, and the

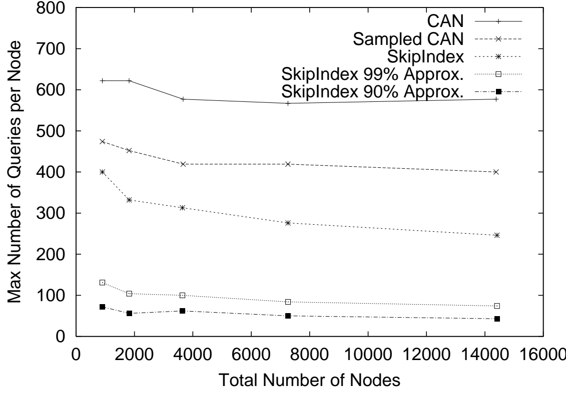


Figure 10: Maximal query load per node.

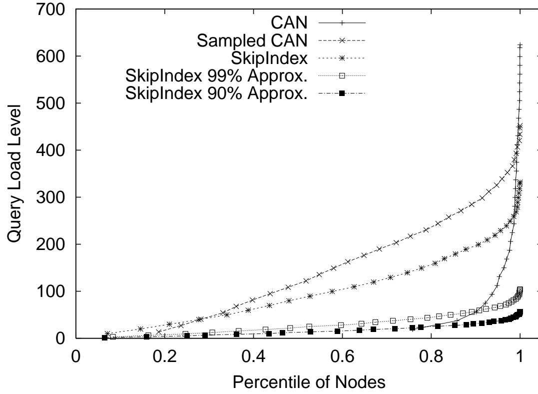


Figure 11: Distribution of query load.

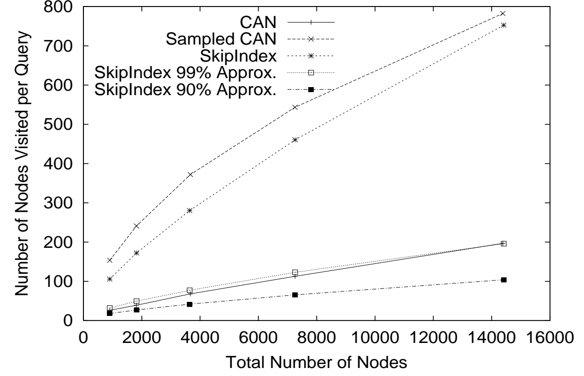


Figure 12: Increase in the number of searched nodes as the system is scaled.

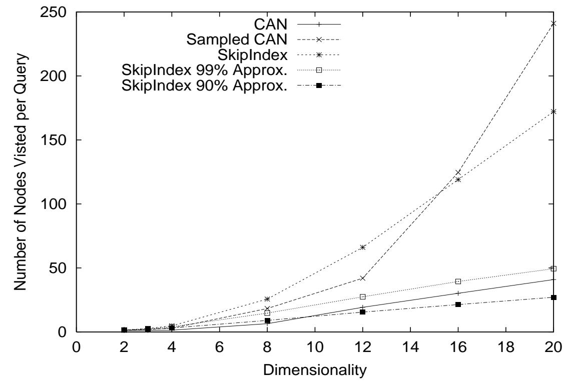


Figure 13: Change in query cost as dimensionality increases.

accuracy of results returned by approximate queries.

We evaluate the performance of the following four systems: **CAN**, **Sampled CAN**, **SkipIndex** and **SkipIndex Approx.**. The query algorithm used for **CAN**, and **Sampled CAN** is a modified version of the neighbor walking algorithm used in the pSearch system [25]. We modify the algorithm so that it returns the exact nearest neighbors and is also optimal in the number of machines searched. **SkipIndex Approx.** uses the approximate search algorithm described in 3.3. We measure the accuracy of approximate results by comparing them to the exact search results. The accuracy is defined as $Accuracy = \frac{|S_{exact} \cap S_{appox}|}{K}$, where S_{exact} and S_{appox} are the sets of results returned by the exact and approximate queries.

Figure 10 presents the maximum load experienced by an index node for various schemes in order to identify whether the scheme suffers from throughput bottlenecks. This metric reflects a system's ability to balance objects across different nodes as well as its ability to avoid routing hot-spots. The SkipIndex-based schemes perform best. Figure 11 depicts the query load distribution on the nodes. CAN shows extremely unbalanced load: 80% of the nodes get negligible workload, while most of the queries are handled by 10% of the nodes. Sampled-CAN performs marginally better.

Figure 12 depicts the number of nodes visited by a query (also referred to as the query span) as the system scales. CAN exhibits very low query span, because its load distribution is highly skewed. A small fraction of CAN nodes host most of the index data and serve most of the queries, as documented by Figure 10 and 11. Amongst the exact search algorithms, Sampled CAN shows a slightly higher span than SkipIndex.

Figure 12 also demonstrates the importance of approximate search for a scalable system. As the system size increases, exact query schemes exhibit much steeper increase in query span than approximate ones. With 14,400 nodes, a 90% approximation search visits about 100 nodes, almost an order of magnitude better than the query span for an exact SkipIndex search.

Figure 13 depicts the query span as we increase the dimensionality of the dataset from 2 to 20. Both SkipIndex and sampled CAN exhibit high query spans for high dimensionality data. However, the query span of sampled CAN increases faster than that of SkipIndex, suggesting that the SkipIndex scheme is more suitable as dimensionality increases.

We next evaluate the approximate query mechanism and study its accuracy. Figure 14 illustrates the improvement in accuracy as we search more nodes in the decreasing order of their intersection volumes with the query sphere. The error ratio decreases sharply after the search has covered a small number of nodes. Figure 15 depicts the relationship between

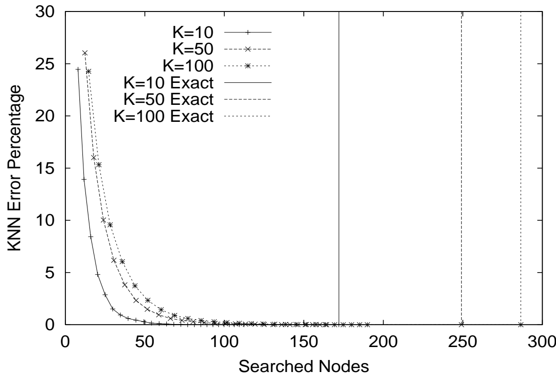


Figure 14: Accuracy vs. Search Span

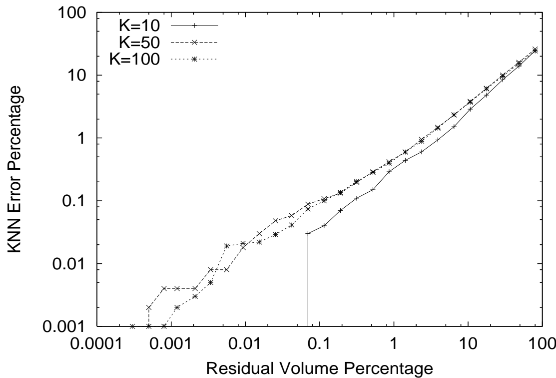


Figure 15: Accuracy vs. Volume Bound

error ratio and the unsearched volume. The volume metric closely bounds the search error rate.

Figure 16 reports the query costs for the SOUND data set. This data set has higher dimensionality and a less clustered distribution. Thus the exact queries incur significantly higher costs and flood 16% to 27% of index nodes. Approximate search substantially reduces this cost.

Figure 17 reports the prototype performance running on 105 Planet-Lab machines. Though the nodes are widely distributed, we achieved 200-300 milli-second query response time with 99% approximation. This latency can be further reduced with the proximity heuristics described in Section 4.3. This is shown by simulations with topologies generated by RocketFuel (see Figure 18).

6 Related Work

The basic problem that this paper addresses – indexing and querying high dimensional data – has received extensive attention from the database research community [6, 9, 5, 8]. Most of the existing work focuses on centralized indices. But the basic approach of partitioning the search space to support efficient similarity search is applicable to distributed implementations. Specifically, we draw inspiration from K-D-Tree [6] with regards to partitioning the search space.

Distributed Hash Tables (DHTs), like CAN [20], Chord [24], Pastry [22] and Tapestry [28], achieve scalability and resilience by building self-organizing overlays to locate resources in peer-to-peer systems. Such systems exhibit a slew of desirable features, such as scalability, fault-tolerance and self-organization. But since these systems use hashing to achieve load-balance, they are not suitable for performing range and nearest neighbor queries.

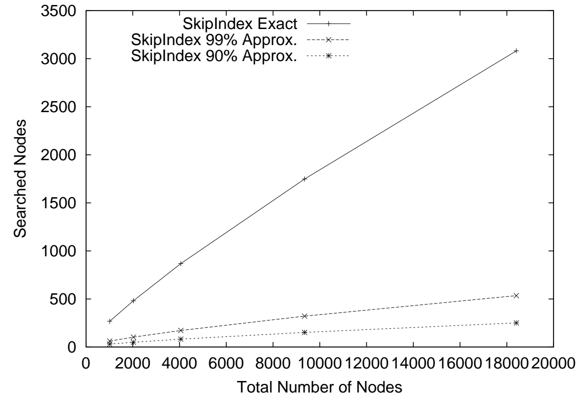


Figure 16: 10 NN query results of SOUND data set

There has been work on supporting range queries by utilizing distributed replicated B-trees [14], space-filling curves [1], and so on. Huebsch *et. al.* have proposed schemes for providing database query functionality, such as the `join` operation, in a peer-to-peer system [12].

Our work is most closely related to pSearch [25], which is a peer-to-peer document retrieval system based on a distributed index of high dimensional semantic vectors. Our work differs from pSearch in several ways. We leverage a new high-dimensional routing scheme that exhibits logarithmic routing state and routing distance regardless of data distribution and dimensionality. We use an overload-triggered space partitioning scheme that can achieve better load balance. pSearch only provides best effort search without a bound on accuracy. Our system provides both exact nearest neighbor query and bounded-error approximate search.

Our distributed index utilizes Skip Graphs [3] as the routing facility. Skip Graphs provide the ability to perform one dimensional range queries using a highly resilient and load-balanced structure. We combine it with our space partitioning scheme and extend it to perform routing over high-dimensional space.

Weber *et al.* [26] analyzed the query performance of index structures and showed quantitatively the difficulty of reducing search cost in high dimensional space. We alleviate the dimensionality curse with approximate search. Our approach differs from existing ϵ-Nearest Neighbor search algorithms [2, 13] in that we bound the error with search volume, which directly relates to the accuracy of search results.

7 Conclusions

In this paper, we present the design and implementation of a peer-to-peer index service for high dimensional data that is capable of handling complex queries. We discussed the various alternatives for distributed index organization and consid-

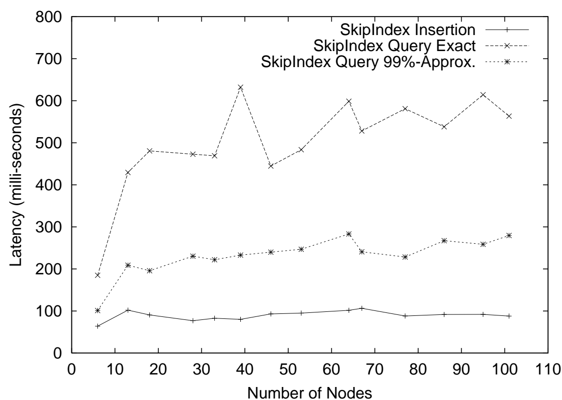


Figure 17: Prototype performance on Planet-Lab

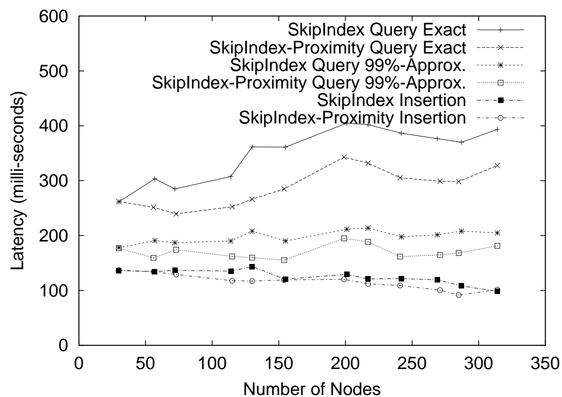


Figure 18: Performance with RocketFuel Topology

ered the challenges associated with efficient query processing, dynamic load balancing, and other key issues. In particular, we make the following contributions in this paper:

- We designed a index service for high dimensional data that achieves logarithmic routing distance and maintains only a logarithmic number of peer pointers regardless of data distribution and dimensionality. Our approach exploits the one-dimensional search capability of Skip Graphs and enhances it for searching high dimensional regions. The same idea can be used to efficiently distribute other hierarchical tree structures.
- We developed algorithms for performing complex queries, such as range queries and nearest neighbor queries, that execute in a completely decentralized manner based on partial local knowledge.
- We introduced the concept of using the intersection volume to bound the error of approximate search. Using this approximation, we can significantly reduce the search range based on a user-specified parameter regarding the desired level of accuracy.
- We compared existing schemes that partition high dimensional search space amongst the index machines, and proposed a balanced partitioning scheme with dynamic load balancing.
- We built a prototype and evaluated it in a real Internet environment using real world data sets. We also performed extensive simulations to compare our proposed scheme with other alternatives.

Our planned future work includes further implementation and evaluation of the failure recovery mechanisms, and evaluation with more dynamic, large-scale platforms and more diverse application data sets.

References

[1] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *Second IEEE International Conference on Peer-to-Peer Computing*, 2002.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, 1994.

[3] J. Aspnes and G. Shah. Skip Graphs. In *Proceedings of Symposium on Discrete Algorithms*, 2003.

[4] B. Awerbuch and C. Scheideler. Peer-to-peer systems for Prefix Search. In *Proceedings of the Symposium on Principles of Distributed Computing*, 2003.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of SIGMOD*, 1990.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9), 1975.

[7] S. Berchtold, C. Bohm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. of the 16th ACM Symposium on Principles of Database Systems*, 1997.

[8] S. Berchtold and H.-P. Kriegel. The X-tree: an index structure for high-dimensional data. In *Proc. of 22nd VLDB*, 1996.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD*, 1984.

[10] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In *Proceedings of IPTPS02*, 2002.

[11] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. of Fourth USENIX Symposium on Internet Technologies and Systems*, 2003.

[12] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *VLDB*, 2003.

[13] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. of the 30th annual ACM symposium on Theory of computing*, 1998.

[14] T. Johnson and P. Krishna. Lazy updates for distributed search structures. In *Proceedings of ACM SIGMOD*, 1993.

[15] P. Keleher, B. Bhattacharjee, and B. Silaghi. Are virtualized overlay networks too much of a good thing, 2002.

[16] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Inf.*, 31(9), 1994.

[17] T. Papadakis, J. I. Munro, and P. V. Poblete. Analysis of the expected search cost in skip lists. In *Proceedings of the second Scandinavian workshop on Algorithm theory*, 1990.

[18] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. First Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[19] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Workshop on Algorithms and Data Structures*, 1989.

[20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM*, 2001.

[21] S. Ratnasamy, J. Hellerstein, and S. Shenker. Range Queries over DHTs. Technical Report IRB-TR-03-009, Intel Research, 2003.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms*, 2002.

[23] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with rocketfuel. In *SIGCOMM*, August 2002.

[24] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[25] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of SIGCOMM*, 2003.

[26] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th VLDB*, 1998.

[27] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 2004.

Appendix

A Proof of Logarithmic Multicast Depth

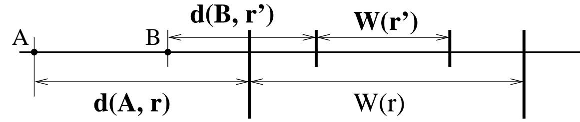


Figure 19: Shrinking of destination region during multicast.

Theorem 1 *(Logarithmic Multicast Depth)*
In a SkipIndex with N nodes, the maximum number of hops in a range-limited multicast is $O(\log N)$.

Proof. Consider forwarding a message from node A to a remote region r. As the message is forwarded to a neighboring node B, B decomposes r into smaller sub-regions and forwards the message to those sub-regions that intersect the query range. So it suffices to prove that the original destination region r can be refined into a leaf region in $O(\log N)$ steps.

As we had noted in Section 2.3.2, all leaf regions obtained from recursively splitting an intermediate region are contiguous in the Skip Graph order. Figure 19 depicts the starting node A and the destination region r in the ordered list. We define $W(r)$ as the number of leaves in region r, or the *width* of r. Also define $d(A, r)$ as the number of leaf nodes between A and r in the list, or the *distance* from A to r. If $d(A, r) < W(r)$, there should be a peer of A falling in the region r, because Skip Lists/Graphs maintain peer pointers at roughly $2^d, d = 1, 2, \dots \log N$, distances. There is no such peer of A because A doesn't know anything inside r in its partial tree view. So we have $d(A, r) \geq W(r)$.

Suppose A routes the query to B, which refines the region to r'. From the Skip Graph property, we have $d(A, r) \geq 2d(B, r')$. So

$$\frac{d(A, r) + W(r)}{d(B, r') + W(r')} \geq \frac{3}{2}$$

Initially for a query initiated at node A, $d(A, r) + W(r) \leq N$. Therefore, within $O(\log N)$ steps, the message would be forwarded to a node X, such that $d(X, r') + W(r')$ becomes 1, which means that r' is a leaf region owned by node X. □

B Intersection of a Query Range with a Region

We consider the intersection of a cube with a quadrant of sphere in D-dimensional space. A D-dimensional sphere has 2^D quadrants. The cube has edge lengths greater than the radius of the ball, so there is exactly one vertex V of the cube inside the quadrant. The sphere intersects the edges of the cube at D points, forming D vertices of the bounding box around the sphere-cube intersection.

Now consider an ellipsoid centered at V, with the edge lengths of the bounding box as the lengths of its axes. A quandrant of the ellipsoid is enclosed inside the bounding box, while its surface is tangential to D surfaces of the bounding

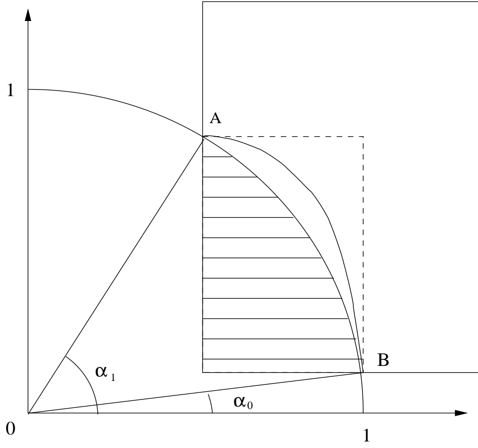


Figure 20: Enclosure of the intersection in 2D space

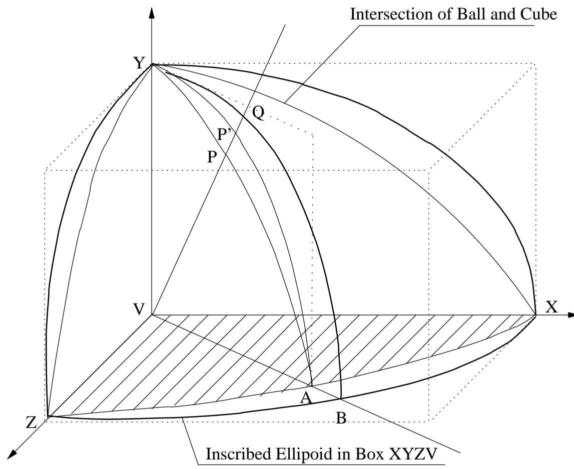


Figure 21: Extending the enclosure relationship from D-dimension to (D+1)-dimension.

box. We call this quadrant of the ellipsoid as the *bounding ellipsoid* of the sphere-cube intersection, because of the following theorem. Figure 20 and Figure 21 illustrate the cases of 2D and 3D intersections with their bounding ellipse/ellipsoid.

Theorem 2 *(Bounding a Sphere-Cube Intersection)*
D-dimensional Sphere-Cube intersections of the type described above are enclosed by a quadrant of an ellipsoid that is constructed to be tangential to the bounding box.

Proof. We prove the D-dimensional case with induction on the dimensionality D.

Figure 20 depicts the base case of 2D intersection. Without losing generality, we assume that the intersection is within the first quadrant. At the vertex A of the bounding box, the ellipse has gradient 0 while the circle has gradient $\cot \alpha_1 \leq 0$. At vertex B, the ellipse has gradient $-\infty$ while the circle has gradient $\cot \alpha_0 \leq 0$. Since the circle and ellipse can have only two intersection points in the first quadrant, the ellipse encloses the circle in the region between A and B.

Now assuming that the statement is true in D-dimensional space, we look at the case at dimension $D+1$. Figure 21 depicts the cases. V is the vertex of the cube inside the sphere. The thin curves represent the intersection sphere, which is bounded by the box $V-X-Y-Z$. The thick line represents the ellipsoid tangential to the bounding box.

For any point P on the sphere surface inside the bounding box, we draw a line VP that intersects the ellipsoid at point Q. The point P and the axis VY forms a 2D plane that is vertical to the D-dimensional hyperplane of $V-X-Z$. Plane $V-P-Y$ intersects $V-X-Z$ with a line. Suppose that this line intersects the sphere surface at point A, and the ellipsoid at point B. From induction hypothesis, the D-dimensional intersection $V-Z-A-X$ is enclosed in ellipsoid $V-Z-B-X$. So point A is between point V and B.

Now consider the intersection of the sphere with 2D plane $V-P-Y$. The intersection part is an arc $A-P-Y$, which is enclosed in its bounding ellipse $V-A-P'-Y$. $V-A-P'-Y$ is in turn enclosed in ellipse $V-B-Q-Y$, since A is between B and V. So we have $\|VP\| \leq \|VP'\| \leq \|VQ\|$, *i.e.*, the ellipsoid surface is outside of the intersection in dimension $D+1$. □

Corollary 1 *Upper Bound of Intersection Volume*
The volume of a sphere-cube intersection I inside one quadrant is bounded as below, where $BB(I)$ is the bounding box of the intersection, and $BB(sphere)$ is the cube circumscribing the entire sphere.

$$\frac{V_I}{V_{BB(I)}} \leq \frac{V_{sphere}}{V_{BB(sphere)}}$$

Proof. From Theorem 2, the intersection is enclosed in the quadrant of an ellipoid, which is also enclosed in the bounding box of the intersection. Thus we get

$$\frac{V_I}{V_{BB(I)}} \leq \frac{V_{ellipsoid}}{V_{BB(ellipsoid)}}$$

From the formula for the volume of an ellipsoid and a sphere in high dimensional space, we have

$$\frac{V_{ellipsoid}}{V_{BB(ellipsoid)}} = \frac{V_{sphere}}{V_{BB(sphere)}} = \frac{\pi^{\frac{n}{2}}}{Gamma(\frac{n}{2}+1)}$$

where $Gamma$ function is a generalization of the factorial function (*i.e.* $Gamma(z+1) = z!$). □