Indexing Multi-dimensional Data in a Cloud System

Jinbao Wang #1, Sai Wu §2, Hong Gao #3, Jianzhong Li #4, Beng Chin Ooi §5
#School of Computer Science and Technology, Harbin Institute of Technology,
Harbin, China
§School of Computing, National University of Singapore, Singapore
1,3,4{wangjinbao, honggao, lijzh}@hit.edu.cn, 2,5 {wusai, ooibc}@comp.nus.sg

ABSTRACT

Providing scalable database services is an essential requirement for extending many existing applications of the Cloud platform. Due to the diversity of applications, database services on the Cloud must support large-scale data analytical jobs and high concurrent OLTP queries. Most existing work focuses on some specific type of applications. To provide an integrated framework, we are designing a new system, epiC, as our solution to next-generation database systems. In epiC, indexes play an important role in improving overall performance. Different types of indexes are built to provide efficient query processing for different applications.

In this paper, we propose RT-CAN, a multi-dimensional indexing scheme in epiC. RT-CAN integrates CAN [23]-based routing protocol and the R-tree based indexing scheme to support efficient multi-dimensional query processing in a Cloud system. RT-CAN organizes storage and compute nodes into an overlay structure based on an extended CAN protocol. In our proposal, we make a simple assumption that each compute node uses an R-tree like indexing structure to index the data that are locally stored. We propose a query-conscious cost model that selects beneficial local R-tree nodes for publishing. By keeping the number of persistently connected nodes small and maintaining a global multi-dimensional search index, we can locate the compute nodes that may contain the answer with a few hops, making the scheme scalable in terms of data volume and number of compute nodes. Experiments on Amazon's EC2 show that our proposed routing protocol and indexing scheme are robust, efficient and scalable.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Distributed databases; C.2.4 [**Distributed System**]: Distributed databases

General Terms

Algorithm, Design, Experimentation, Performance

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...$10.00.

Keywords

Cloud, Index, Query Processing

1. INTRODUCTION

Deploying database services on the Cloud poses new challenges to the community. Lacking scalability and reliability, conventional database designs and principles cannot be directly applied to the new platform. Therefore, a new architecture that is specially tailored for the Cloud is essential.

Analytical jobs and online transactions are two basic database workload types. Most recent work on the Cloud focuses on large-scale data analytical jobs [22, 5, 12] while some work such as [10] attempts to support high concurrent OLTP queries. *epiC* (elastic power-aware data-instensive Cloud) [2], is a cloud-based data management system that is being implemented jointly by the National University of Singapore and Harbin Institute of Technology as an integrated solution for both analytical jobs and OLTP transactions. epiC is designed to provide primitive operators based on the principle of *filter* and *refine* to serve as basic building blocks for more complex operators so as to achieve the necessary parallelism and fault tolerance. An SQL query will then be translated into a DAG (Directed Acrylic Graph) of primitive operators, and a master server schedules and monitors the processing of such operators. As in conventional DBMS, these operators may apply index-based processing or scan-based processing. Indexes can indeed improve query performance by orders of magnitude for both large-scale analytical jobs [16] and OLTP queries [10]. In epiC, indexing schemes adopt a two-layer approach, which is light weight and query-pattern conscious, and ride on local indexes.

The first application that we are implementing is a community travelog system designed to help travelers plan their travel itinerary and share their experiences and photos. In such a system, we need to support photo search via keywords/tags and geo-tags. Specifically, each photo object is expressed as $\{g_0, g_1, ..., g_m,\ k_0, k_1, ..., k_n\}$, where g_i represents its geographic information (e.g., latitude, longitude, location name, etc.) and k_i describes the features of the photo (date, topic, photo type, color histogram, textures, etc.). Typical queries include searching photos with specific tags in a given location and finding the geo information about a specific photo. To efficiently support the above queries, a multi-dimensional indexing strategy, called RT-CAN, is designed for epiC.

In this paper, we present our implementation of RT-CAN. We target internet-scale applications, where hundreds of servers are used to support tera-byte data and millions of users; the index must therefore be able to scale up when more servers are added or more data are inserted. One popular solution is to apply a master server to maintain the global index. All queries are sent to the master server to search the global index and then forwarded to corresponding servers. Due to the fact that the size of the global index is proportional to the size of the data and the number of concurrent requests is huge, the master server risks being a bottleneck. Therefore, we propose distributing the global index across servers. Each server only maintains a portion of the global index. The distributed approach improves scalability and fault tolerance.

To distribute the global index, servers are organized into an overlay network. Specifically, CAN (Content Addressable Network) is adopted as it is suitable for multi-dimensional search. Our global index is built on top of the local indexes. To search the local data efficiently, an R-tree like multi-dimensional index should be built for the local database. In the global index, instead of indexing every tuple, we publish local R-tree nodes into the global index. Consequently, the global index in our system plays the role of an "overview" index, composed of R-tree nodes from different servers. Therefore, we name our global index RT-CAN (the R-Tree based index in CAN). Indexing R-tree nodes reduces the size of the global index and hence lowers maintenance cost. Once we have located an R-tree node in the global index, we can continue search in the local R-tree, starting from the node registered in the global index.

The main challenge to the design of an efficient RT-CAN scheme is the distribution of the global index onto compute servers. In RT-CAN, the multi-dimensional search space is partitioned into zones based on CAN's protocols, which is similar in concept to that of the grid-file and the kd-tree [9]. Each server maintains a zone, and the partitions are adjusted adaptively. A "hot" zone can balance its load with its neighbors by shrinking its zone. For the fact that an R-tree node represents a multi-dimensional hyper-cube, we build a mapping function between R-tree nodes and CAN servers. If an R-tree node is inserted into the global index, it will be published to mapped servers. To process a query, we search the global index and route the query to servers whose zones overlap with the query. Upon receiving the query, the servers will process the query in parallel. We apply CAN's routing protocols to forward R-tree nodes and queries from their owners to the mapped servers. In d-dimensional CAN, the average routing cost is $\frac{d}{4}N^{\frac{1}{d}}$, where N is the number of servers. To reduce routing cost, we apply a variant of CAN [11] by maintaining Chord-like routing neighbors along each dimension on the same grid interval, where the routing cost is reduced to $\log \frac{N}{4}$. Moreover, a routing buffer is maintained in each server to further reduce routing cost.

It is not practical to publish every R-tree node to the global index as that incurs high maintenance overhead. Therefore, it suffices to select a portion of R-tree nodes that are useful for routing purposes and publish them in the global index. Since different selections incur different costs (routing cost, maintenance cost and query cost), the challenge lies in choosing an index strategy which incurs the least total cost. We present a scheme to estimate the cost of a possible indexing strategy. Based on the statistics about query distribution and updates, our scheme can dynamically tune the indexing strategy to minimize cost.

This paper makes the following contributions:

1. A distributed global index, the RT-CAN index, is proposed to support efficient multi-dimensional search. The RT-CAN index is built on top of local R-trees and published to cluster servers.

2. We present a method to map a selected R-tree node to a CAN node. Query processing algorithms are designed to support point, range and KNN queries for the RT-CAN index.

3. A cost model is proposed to estimate the cost of different indexing strategies, based on which a dynamic tuning algorithm is proposed to selectively publish local R-tree nodes onto the global index. The tuning algorithm optimizes maintenance cost and query cost.

4. Experiments on Amazon's EC2 verify the effectiveness and efficiency of the RT-CAN index.

The rest of this paper is organized as follows: Section 2 presents related works while Section 3 gives an overview of our indexing scheme. Section 4 presents our query processing algorithms, including the point, range and KNN queries. Section 5 presents index selection and tuning methods, and describes the selection of R-tree nodes and the method to tune them according to the current workload. Section 6 presents the performance evaluation, and we conclude in Section 7.

2. RELATED WORK

2.1 Distributed Data Processing System

Much effort has been invested in designing distributed storage systems to manage large amounts of data, such as Google File System[18] (GFS), which serves Google's applications with large data volume. BigTable[13] is a distributed storage system for managing structured data of very large scales. There are some open source implementations of GFS and BigTable such as HDFS[1], HBase and HyperTable, which have indeed provided a good platform for research and development. Yahoo proposed PNUTS [10], a hosted, centrally controlled parallel and distributed database system for Yahoo's applications. These systems organize data into chunks, and then randomly disseminate chunks into clusters to improve data access parallelism. Some central servers working as routers are responsible for guiding queries to nodes which hold query results. In [25], *bulk insertion* were proposed to insert data efficiently in these systems. Unlike these work, we build a scalable second-level index which provides data location functionality based on the distributed storage system. Our indexing scheme is designed to route a large amount of queries among a large cluster of storage nodes. Amazon's Dynamo[17] is a readily available key-value store based on geographical replication, and it can provide eventual consistency. Each cluster in Dynamo organizes nodes into a ring structure, which uses consistent hashing to retrieve data items. Consistent hashing is designed to support key-based data retrieval and is not a good candidate to support range and multi-dimensional queries. To support query processing over multi-dimensional data, we use CAN (Content Addressable Network) to build our database storage system. Commercial distributed storage systems such as Amazon's S3 (Simple Storage System) and

Microsoft's CloudDB tend to have little implementation details published. Some other systems such as Ceph[7], Sinfonia[6], etc., are designed to store objects, and they aim to provide high performance in object based retrievals instead of set based database retrieval. MapReduce[22] was proposed to process large datasets disseminated among clusters. MapReduce assigns mappers and reducers to process tasks, where mappers produce intermediate results in parallel and reducers pull the intermediate results from mappers to do aggregations. Recent work, such as [30] and [5], attempt to integrate MapReduce into database systems.

These works form a parallel framework for processing large datasets. Our work is to build a second level overview index, which can also be employed in systems such as GFS and MapReduce, among cluster nodes. Our work follows the framework proposed in [29]. However, to support multi-dimensional data, new routing algorithms and query processing algorithms are proposed. Furthermore, both range queries and KNN queries are studied in this paper.

2.2 Overlay Networks

Besides CAN, other overlay structures, such as tree topology overlays (e.g., BATON[21][20], Pastry[24], P-grid[4] and P-tree[14]) and ring topology overlays (e.g., Chord[26] and P-ring[15]), have also been proposed. Chord and Pastry are very effective for exact match queries. Others can process one-dimensional range queries. However, except CAN, none of the above overlays can support multi-dimensional queries naturally. Much work has been done to enhance the overlay network with the capability of handling multi-dimensional queries [28]. Most of them consider a datum as a point in multi-dimensional search space and index it accordingly. In contrast, we group data into hypercubes and index the hypercubes in CAN. An original overlay structure is defined for dynamic networks such as a peer-to-peer network. In our case, the overlay is only used to build a logical network for partitioning data and routing queries. The overlay can be considered a static network as nodes in the Cloud system will stay online till the hardware fails.

2.3 Content Addressable Network

CAN (Content Addressable Network) is a scalable, self-organized structured peer-to-peer overlay network. A d-dimensional CAN partitions a virtual d-dimensional Cartesian coordinate space among nodes in CAN, and assigns each node a d-dimensional zone. A node in CAN maintains data mapped to its zone.

CAN applies a hash function $hash_{CAN}$ to map a data item via its key to a point in the coordinate space. Data item $(key, value)$ is stored in the CAN node whose zone contains the point $P = hash_{CAN}(key)$. CAN Node N_i maintains a routing table storing the ip addresses of other CAN nodes whose zones are adjacent to N_i's zone. Once N_i gets a query for a certain key, it routes the query to a neighbor whose zone is nearest the point $P = hash_{CAN}(key)$. In a d-dimensional CAN with N nodes, the average number of routing hops for a query is $\frac{d}{4}N^{\frac{1}{d}}$, and each node maintains $2d$ neighbors in its routing table.

A node can freely join or leave a CAN network because CAN is totally self-organized, and the join or departure of a node incurs $2d$ messages to update its neighbors' routing table. In this paper, we use CAN to organize cluster nodes. These cluster nodes are much more stable than peers in P2P networks, and thus, we will not consider the update cost of node join or departure in our cost model.

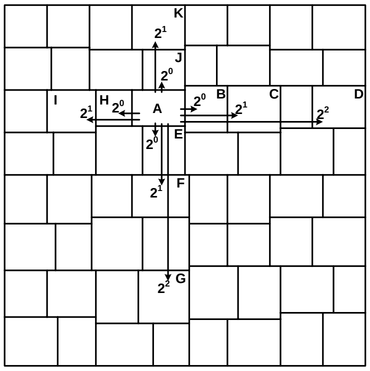


Figure 1: An Example of 2-d C^2 Overlay.

CAN's routing cost is slightly higher than other overlay networks. Therefore, instead of applying CAN, we adopt a CAN's variant, C^2 [11]. C^2 is a hybrid overlay of Chord and CAN. It extends CAN by adding Chord-like neighbor links on each dimension. Specifically, neighbor links are added between nodes with distance of 2^0, 2^1, ... on each dimension. Figure1 shows an example of a 2-d C^2, made up of 64 nodes. Node A keeps a neighbor link to node B, C, D since they are at a distance of 2^0, 2^1 and 2^2 on the horizontal dimension, respectively. Similarly, node A also keeps neighbor links to node E, F, G, H, I, J and K. On average, an C^2 node maintains $O(\log \sqrt[d]{N})$ neighbors on each dimension. Thus, the total number of neighbors is $O(logN)$. Therefore, C^2 actually maintains the same number of routing neighbors as the other overlay networks. C^2 reduces the average routing hop number to $log\frac{N}{4}$.

In the following part, we also use CAN node to refer to a C^2 node, since CAN is the original overlay structure.

3. RT-CAN INDEX

The RT-CAN index is built on top of the local indexes. The key problem of the RT-CAN index lies in mapping a local R-tree node to a CAN server. In this section, we present an overview of the RT-CAN index scheme in our Cloud data management system. Specifically, we focus on the mapping function of the RT-CAN index. We also present a general index publication algorithm for building the RT-CAN index.

3.1 Work Flow and Node Structure

The RT-CAN index is built on a shared-nothing cluster, where application data are partitioned and distributed over different servers. As shown in Figure 2, a cluster server N_i assumes two roles, namely, storage node N_{si} and overlay node N_{oi}. N_{si} joins a distributed storage system and maintains a portion of application data. To facilitate multi-dimensional search, N_{si} builds an R-tree for its local data. N_{si} publishes its index by interacting with N_{oi}. N_{oi} is a node in the structured overlay, CAN. It responds for a partition of CAN. N_{si} adaptively selects a set of R-tree nodes from its local R-tree and publishes them into the overlay network via the interface of N_{oi}. The format of the published R-tree node is (ip, mbr), where ip is N_i's IP address and mbr is the minimal bounding range of the R-tree node. On receiving a publication request from N_{si}, N_{oi} maps the corresponding

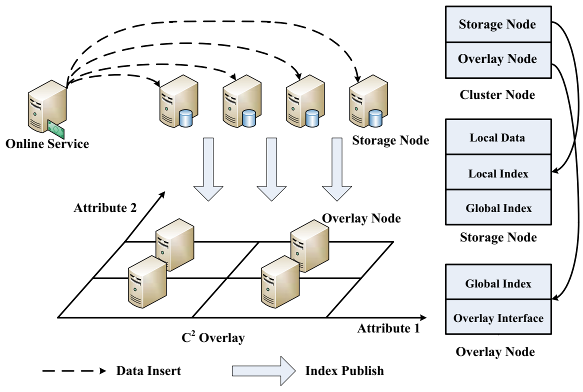


Figure 2: Data Service for Online Applications

R-tree node to a CAN node and forwards the request based on CAN's routing protocol. N_{oi} is responsible for maintaining the global index. After it receives a publication request, it checks whether it is the receiver of the request based on the mapping function. If it is, it keeps a copy of the published R-tree node and buffers the index in memory. In this way, the global index composes of some R-tree nodes from the local indexes and is distributed over the cluster.

The global index can be considered as a secondary index on top of the local R-trees. This design splits the processing of a query into two phases. In the first phase, the processor looks up the global index by mapping the query to some CAN nodes. These CAN nodes search their buffered R-tree nodes and return the entries that satisfy the query. In the second phase, based on the received index entries, the query is forwarded to the corresponding storage nodes, which retrieve the results via the local R-tree. The detailed query processing algorithm can be found in Section 4.

Data insertion or deletion may trigger an update to the local R-tree, which violates the global index. To handle this problem, we need to synchronize the global index with the local R-trees. If a local R-tree node has been split or merged and it is marked as being published, we publish it again to replace the outdated global index. The RT-CAN index reduces update cost by publishing only a portion of R-tree nodes. If we index every tuple in the global index (traditional P2P systems adopt this idea), any update to the local database will trigger an update to the global index. However, the RT-CAN index introduces false positive cost. As higher level R-tree nodes cover a large range, a query overlapping the R-tree nodes does not guarantee that valid results will be retrieved. To balance update cost and false positive cost, we propose a tuning approach. The cluster server occasionally re-selects R-tree nodes for publishing based on recent query and update patterns.

3.2 Mapping Scheme in CAN

We use d-dimensional C^2 [11] to maintain index items for d-dimensional data. C^2 is a variant of CAN with enhanced search capability. It reduces routing hops by increasing neighboring links in each dimension, and search cost is bounded by $O(\log N)$. To facilitate index publishing, a mapping function is created to map an R-tree node to a CAN node. Specifically, the center and radius of the R-tree node's bounding box are used as the criteria for mapping. If the radius is smaller than a threshold, say R_{max}, the R-tree node is mapped to one CAN node. Otherwise, it is mapped to multiple CAN nodes. The mapping function works as follows: For an R-tree node n, we first map it to the overlay node N_c whose zone contains n's center. Then N_c compares n's radius and R_{max}. If n's radius is larger than R_{max}, N_c will send n to all the overlay nodes whose range overlaps with n's range. The center is used in the mapping function, as R-tree nodes with close centers may contain similar data with high probability.

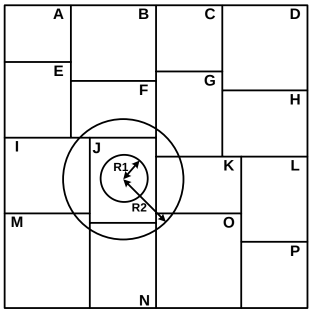


Figure 3: Different Search Space of An Exact Query With Different R_{max} Values (R_1 and R_2)

The above mapping scheme creates multiple replicas of R-tree nodes with larger ranges (radius larger than R_{max}). This strategy works on the basis of update cost and search cost. An R-tree node with a small range refers to a low-level node in the tree with high probability. Low-level node always incurs more updates than the up-level ones. Therefore, keeping multiple replicas introduces high overhead. In contrast, for an R-tree node with a larger range, multiple CAN nodes are covered. If we only maintain one copy of the index for it, all queries need to search that CAN node, which reduces query efficiency.

R_{max} has a significant impact on the search space of a query. A query may need to access more CAN nodes for a larger R_{max}. Figure 3 shows an example of the search space of a query $Q(key)$ with different R_{max} values in a 2-d CAN overlay. The circles with radius R_1 and R_2 are the search spaces of point query when R_{max} is set to R_1 or R_2. The query has to access more compute nodes to obtain all index items which contain the search key if we increase R_{max} from R_1 to R_2. R_{max} is a tunable parameter. In an RT-CAN indexing system, a larger R_{max} degrades query performance as more space must be searched to retrieve complete results. On the other hand, a smaller R_{max} increases maintenance cost by creating too many replicas. In our system, we set R_{max} to a value larger than the radiuses of most leaf nodes of local R-trees.

3.3 Publishing In The RT-CAN Index

Initially, due to lack of information about the query pattern and the update pattern, we cannot select R-tree nodes based on our cost model. Therefore, in an L-level R-tree, we select R-tree nodes in the L-1 level (nodes above the leaf nodes) to publish since these nodes are usually not frequently updated and they do not introduce too many false positives either. After a cluster server N_i selects a set of R-tree nodes, S_i, for indexing, it publishes the index for every R-tree node in S_i. $\forall n \in S_i$, N_i computes its center

c_n and radius r_n, and publishes n based on our mapping function. For example, in a two-dimensional space, suppose node N_i wants to publish an R-tree node n whose range is $[l_1, u_1], [l_2, u_2]$. N_i first computes n's center and radius as $c_n = (\frac{l_1+u_1}{2}, \frac{l_2+u_2}{2})$ and $r_n = \frac{1}{2}\sqrt{(u_1 - l_1)^2 + (u_2 - l_2)^2}$. We use the center as the key to publish the node, namely $key_n = c_n$. If the dimensionality of CAN (d) is larger than the search space d', N_i will generate a $(d - d')$-vector v_n and append it to key_n. Then, n has a d-length combination key $key'_n = (v_n, key_n)$. The index of the R-tree node n is sent to the cluster node N_j whose zone contains n's center, c_n. The CAN dimension d should be chosen properly to ensure that it is not smaller than the largest dimensionality of data. We can choose a large enough d to keep this condition since the dimensionality of a table is usually not very large. *Algorithm* 1 describes how N_i publishes an index item for an R-tree node n. If $n.radius > R_{max}$, all CAN nodes overlapping with n need to maintain n in its global index. In line 1, a subset of R-tree nodes are selected for publishing. In line 4 and 7, we apply the routing protocol of CAN (C^2) to retrieve the overlapping cluster servers.

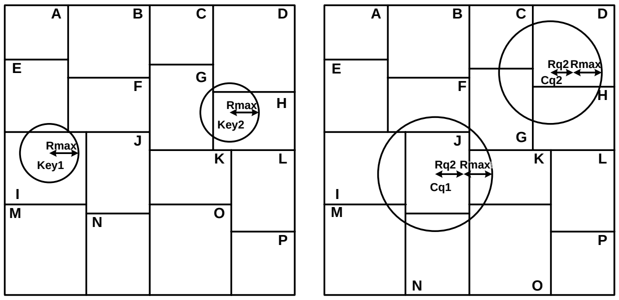
Algorithm 1 IndexPublication(cluster node N_i)

```
 1: $S_i = getSelectedRTreeNode(N_i)$
 2: for each $n \in S_i$ do
 3:     $c_n = n's\ center$; $r_n = n's\ radius$
 4:     $key_n = compositeKey(c_n, r_n)$
 5:     $N_j = CAN.lookup(key_n)$
 6:     $N_j$ inserts $n$ into its global index set
 7:     if $r_n > R_{max}$ then
 8:         $S_n = getOverlappedNode(n)$
 9:         for $\forall N_k \in S_n$ do
10:             $N_k$ inserts $n$ into its global index set
11:         end for
12:     end if
13: end for
```

Given a key k, routing to the RT-CAN node whose zone contains k will cost $log\frac{N}{4}$ messages averagely, where N is the number of servers in the cluster. So, it will cost $log\frac{N}{4}$ messages to publish an R-tree node whose radius is less than R_{max}. If the node's radius is larger than R_{max}, it will cost $log\frac{N}{4} + m$ messages, where m is the number of CAN nodes whose zones overlap with the published R-tree node. The total cost of indexing the set S_i is bounded by $O(|S_i| \log N)$.

4. QUERY PROCESSING

In this section, we show how the RT-CAN index can be applied to process various types of queries. We use point, rang and KNN queries as examples. As discussed before, query processing can be split into two phases. In the first phase, we look up the RT-CAN index to retrieve the cluster servers that may provide possible results. In the second phase, the query is routed to the servers and processed locally. We focus on the first phase, as the second phase is similar to any centralized system. Given a query Q, to guarantee the completeness of the result, we need to retrieve all possible cluster servers that overlap with Q. To achieve this goal, efficient algorithms are proposed to prune the search space, using center and radius of both indexed R-tree nodes and queries. In the rest of this paper, we assume the dimensionality of RT-CAN to be d, and we discuss the processing algorithms for different queries.



(a) Search Space of Exact Query Q(Key1) and Q(Key2)

(b) Search Space of Range Query q1 and q2

Figure 4: Search Space of Queries

4.1 Point Query Processing

The point query is denoted as $Q(key)$, where $key = (v_1, ..., v_d)$, indicating a d-dimensional point. Given an indexed R-tree node n and its bounding box $[l_1, u_1], ..., [l_d, u_d]$, if $l_i \leq v_i \leq u_i$ for $1 \leq i \leq d$, n should be searched to retrieve possible results. Suppose key is mapped to a CAN node N, based on our publishing strategy, if n's $radius$ is larger than R_{max}, n may be maintained by a CAN node other than N. Thus, to retrieve all possible results, the search space for a point query $Q(key)$ is a circle C centered at point key with radius R_{max}. All the CAN nodes whose zones overlap with C should be accessed to retrieve the corresponding R-tree nodes. Theorem 1 guarantees the completeness of the results. Figure 4(a) illustrates the search space when d=2.

THEOREM 1. *For a point query $Q(key)$, if we search the circle centered at key with radius R_{max}, we can guarantee the completeness of the results.*

PROOF. Suppose an indexed R-tree node n has center c and radius r, and it contains the search key, e.g., $distance(key, c) \leq r$. Based on our mapping function, if $r \leq R_{max}$, n must overlap with the search circle, as $distance(c, key) < R_{max}$. Otherwise, if $r > R_{max}$, we have created the replicas for n in every overlapped CAN node. Therefore, the search circle must contain a replica of the R-tree node. □

To process a point query $Q(key)$, we first forward the query to the CAN node N_{init}, whose zone contains the key. N_{init} searches its buffered global index and returns matched results to the user. Then, it generates a search circle C for the query and forwards the query to other CAN nodes that overlap with the circle. The query message is forwarded via CAN's routing protocol. To reduce overhead, we slightly modify the routing algorithm. Suppose N_{init}'s zone is $[l_1, u_1], ..., [l_d, u_d]$. We partition the search range (e.g., C) based on N_{init}'s range iteratively. Specifically, C is first partitioned by the first dimension. Three sub-ranges are generated, R_0, R_1 and R_2. R_0 is the sub-range between l_1 and u_1. R_1 is the sub-range below l_1 and R_2 is the sub-range above u_1. R_1 and R_2 are routed to the neighbors of N_{init} in the first dimension. R_0 is continuously partitioned by the second dimension. This process completes when the search range has been partitioned by all dimensions. Besides the sub-range handled by N_{init} itself, the remaining sub-ranges are forwarded to the neighbors. The head of the routing message follows the format of $\{R, N_i\}$, where R is the sub-range and N_i is the destination. N_i is responsible for forwarding the query to all the other CAN nodes that overlap with R.

Algorithm 2 shows the idea of point query processing. We first route the query to the CAN node responsible for the key (line 2). Then, the search circle is generated based on the key and the maximal radius R_{max}. From line 4 to 10, we continuously partition the search circle into sub-ranges and forward the query to the neighbor nodes. Finally, the node searches its buffered global index and returns the corresponding indexed R-tree nodes to the user(line 11-15). In line 9 and 10, when the neighbor nodes receive the query message, they invoke a similar algorithm to partition and process the query. The only difference is that they do not need to find the initial node and generate the search circle.

Algorithm 2 Point Query Processing

```
    Input: $Q(key)$
    Output: $S_i$ (the result of indexed R-tree nodes)
 1: $S_i = \phi$
 2: $N_{init} = CAN.lookup(key)$
 3: $C = generateSearchCircle(key, R_{max})$
 4: for i=1 to $d$ do
 5:     partition $C$ into $R_0$, $R_1$ and $R_2$ based on $l_i$ and $u_i$
 6:     $C = R_0$
 7:     $N_1 = getNeighbor(N_{init}, R_1)$
 8:     $N_2 = getNeighbor(N_{init}, R_2)$
 9:     Forward query message ($N_1$, $R_1$) to $N_1$
10:     Forward query message ($N_2$, $R_2$) to $N_2$
11: end for
12: $S_i = N_{init}.globalIndex$
13: for $\forall I \in S_i$ do
14:     if $I$'s bounding box does not contain $key$ then
15:         $S_i = S_i - \{I\}$
16:     end if
17: end for
18: return $S_i$
```

The cost of processing a point query $Q(key)$ is bounded by RT-CAN's routing cost. Averagely, it costs $\log \frac{N}{4}$ messages to forward a message to the initial node N_{init}. Note that by applying the variant of CAN, we reduce the routing cost from $O(\sqrt[d]{N})$ to $O(\log N)$. At the initial node, the message is continuously forwarded to another M nodes that overlap with the search circle. Thus the total cost is about $log\frac{N}{4}+M$.

4.2 Range Query Processing

A range query is denoted as $Q(range)$, where $range = [l_1, u_1], ..., [l_d, u_d]$ is a multi-dimensional hypercube. The range's center is defined as $(\frac{l_1+u_1}{2}, ..., \frac{l_d+u_d}{2})$ and its radius is computed as $\frac{1}{2}\sqrt{(u_1-l_1)^2 + ... + (u_d-l_d)^2}$. If an R-tree node n's bounding box $[l'_1, u'_1], ...[l'_d, u'_d]$ overlaps with $range$, n should be searched to retrieve possible query results. Figure 4(b) shows the search space of two range queries $Q1$ and $Q2$. An R-tree node n, contains query results for $Q(range)$ only if $distance(n.center, range.center) \leq R_{max}+range.radius$. Thus, the search space of a range query $Q(range)$ is a circle C centered at $range.center$, and its radius is $R_{max} + range.radius$. Any server whose zone overlaps with C should be accessed to retrieve possible query results.

Theorem 2. *For a range query $Q(range)$, if we search the circle centered at key with radius $R_{max} + range.radius$, we can guarantee the completeness of the results.*

Proof. Suppose an indexed R-tree node n has center c and radius r and it overlaps with $range$. We have $distance(c, range.center) < range.radius + r$. If $r \leq R_{max}$, c must be contained by the search circle. If $r > R_{max}$, we have created replicas of n in every overlapped CAN node. Therefore, the search circle must contain a replica of the R-tree node. □

Given a d-dimensional range query $Q(range)$, our algorithm will return all indexed R-tree nodes whose bounding boxes overlap with $range$. The processing progresses in a manner similar to point query processing. First, the query is routed to the CAN node N_{init} whose zone contains the center of $Q(range)$. Then, the query is recursively forwarded to all neighbors that overlap with the search circle. The query is broadcast in a way similar to point query processing. The search space is partitioned for each dimension and sent to the corresponding neighbors. On receiving the query, the CAN node searches its global index and returns candidate R-tree nodes. The detail of the range query algorithm is omitted here as it is similar to Algorithm 2.

4.3 KNN Query Processing

Given a KNN query $Query(key, k)$, our algorithm returns top-k data items which are closest to key. The intuition of the KNN algorithm is to start with a small range and enlarge the search range accordingly. Although our scenario is similar to the KNN problem in the R-tree index, conventional algorithms, such as MINDIST-based pruning, cannot be applied directly. The RT-CAN index is disseminated to different cluster servers. Given a point key, which is mapped to the CAN node N_{init}, it is challenging to find the closest indexed R-tree node of key. N_{init} needs to communicate with its neighbors in all dimensions to find all candidates. This may incur significant network overhead. To retrieve top-k results, we need several iterations. Each iteration involves communication among a set of CAN nodes. Exploiting the parallelism in Cloud systems, we adopt a simple but effective algorithm.

Algorithm 3 illustrates our KNN algorithm. In lines 3 and 4, we set an initial search radius based on the data distribution. $\delta = \frac{D_k}{k}$, where D_k is the estimated distance between key and its k-nearest neighbor. D_k can be estimated by the following equations [27]:

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

where d is the dimensionality of RT-CAN, N is the estimated number of data in the whole space, $\Gamma(x+1) = x\Gamma(x)$, $\Gamma(1) = 1$ and $\Gamma(\frac{1}{2}) = \frac{\pi}{2}$. The KNN computation comprises a series of range queries (line 6). $Search(key, r)$ returns the data, whose distances to key are no larger than r. Suppose $key = (x_1, ..., x_d)$, $Search(key, r)$ is similar to range query processing, and the only difference is that $Search(key, r)$ uses r as the radius of the query and uses key as the center of the query. On receiving the search message, the CAN node will check its global index and forward the request to the corresponding cluster server, where a local search algorithm is invoked to return its local data in the search range (key, r). After each iteration, if we have not obtained enough results yet, we increase the search radius by δ (line 10).

Algorithm 3 KNN Query Processing

```
Input: $Q(key, k)$
Output: $S_i$ (the result of indexed R-tree nodes)
$S_i = \phi$
$N_{init} = CAN.lookup(key)$
$\delta = estimateRadius(k)$
$r = \delta$
while true do
    $S_i = Search(key, r)$
    if $|S_i| \geq k$ then
        return top $k$ results of $S_i$
    else
        $r = r + \delta$
    end if
end while
```

The cost of KNN algorithm depends on the value of K. For a large K, Algorithm 3 has to search a larger range to retrieve enough data, which means more CAN nodes will be contacted and more local search will be invoked. If the data follows uniform distribution, the search space of a KNN query $Q(key, k)$ is in proportion to k. If the data is skewed, and the estimated radius is not large enough to retrieve enough data, a KNN query will cost more iterations, reducing its efficiency.

5. INDEX MAINTENANCE

We adopt an adaptive indexing strategy in this paper to minimize the maintenance cost of the RT-CAN index. Based on the current query and update patterns, the cluster server adaptively selects a portion of local R-tree nodes for publishing. A low-level R-tree node (node close to the leaf) generates fewer false positives in query processing. However, it may incur more update costs. It is necessary to balance the cost of false positive and updates. When selecting R-tree nodes for indexing, we must guarantee that the index is complete and unique.

Definition 1. ***Index Completeness***
Suppose cluster server N_i selects a set of local R-tree nodes S_r to build its global index. The index is said to be complete, if and only if for any tuple t in N_i's local database, t is contained by one R-tree node in S_r.

Definition 2. ***Unique Index***
Suppose cluster server N_i selects a set of local R-tree nodes S_r to build its global index. The index is said to be unique, if for an R-tree node n_i and its ancestor node n_j, $n_i \in S_r \rightarrow n_j \notin S_r \wedge n_j \in S_r \rightarrow n_i \notin S_r$.

Index completeness guarantees the correctness of query processing while the unique index strategy reduces maintenance overhead by removing redundancy. Our index selection algorithm satisfies the above two requirements and is a variant of dynamic programming. Its cost is $O(V)$, where V is the number of local R-tree nodes. Our index selection algorithm aims to reduce index maintenance cost and false positive cost at the same time. It is invoked occasionally to adapt to the change of query or update patterns. Before delving into the details of the selection algorithm, we first propose a cost model to estimate the cost of a specific indexing strategy.

5.1 Cost Model

We only consider network cost in our cost model, as: 1) disk I/O is much faster than network I/O, and 2) we have built an R-tree index for the local data, which facilitates data retrieval. In the RT-CAN indexing system, two types of cost dominate the system's performance: index maintenance cost and query processing cost. To estimate the cost, we record queries and updates in a time window T. The cost $C(n)$, incurred by an indexed R-tree node n, consists of query processing cost $C_P(n)$ and index maintaining cost $C_M(n)$.

$$C(n) = C_P(n) + C_M(n)$$

As we have discussed, applying the RT-CAN index to query processing generates false positives. Some cluster servers that cannot provide any result are searched as well. This is because we index some high-level R-tree nodes, which cover a wide range. Therefore, the routing cost of query processing can be classified into essential cost $C_E(n)$ and false positive cost $C_{FP}(n)$. As $C_E(n)$ cannot be further reduced, we focus on $C_{FP}(n)$ in this paper. For an indexed R-tree node n, we record the query incurring false positives in a set $Q_{FP}(n)$. $Q_{FP}(n)$ is defined as:

$$Q_{FP}(n) = \{q | q.range \cap n.range \neq \phi \wedge |f(n, q)| = 0\}$$

where $f(n, q)$ returns the result set for q. Suppose there are N cluster servers, $C_{FP}(n)$ can be computed as follows:

$$C_{FP}(n) = \log \frac{N}{4} |Q_{FP}(n)|$$

The index maintenance operation is triggered by the local R-tree's update. A cluster server N_i is responsible for updating its published index in the overlay. Once an R-tree node n_j in N_i's published set is split or merged, N_i needs to notify the cluster server that maintains the index for n_j. If n_j is split into two R-tree nodes, N_i removes the existing index for n_j and publish the index of the new nodes. Algorithm 4 summarizes the processing of index update. The splitting of an indexed R-tree node incurs $log\frac{N}{4}$ messages to delete the published index item. It also takes another $2log\frac{N}{4}$ messages to publish newly generated R-tree nodes. Thus, the total cost is about $3log\frac{N}{4}$. If n_j is merged with another R-tree node n_k, we need to replace n_j's old index with the new one. Based on our index selection algorithm, which we will discuss later, n_k is indexed as well. Therefore, the total cost of merging is $3log\frac{N}{4}$.

Algorithm 4 UpdateIndex

```
Input: $n$
if $n$ is split into $n_1$ and $n_2$ then
    delete($n$)
    publish($n_1$), publish($n_2$)
end if
if $n$ is merged with $n'$ into $n_{new}$ then
    delete($n$), delete($n'$)
    publish($n_{new}$)
end if
```

To capture the update pattern, a histogram is built in each cluster server. The histogram partitions the space into equal-size grids. In each grid, we maintain the number of insertions and deletions in the last T time. Given an R-tree node n, we can use the histogram to estimate the number of insertions and deletions for n. Applying a two-state markov chain model [19], we can estimate the probabilities of splitting and merging in the next time T. Let $p_{split}(n)$ and $p_{merge}(n)$ denote the probabilities of splitting and merging R-tree node n, respectively. We estimate the index maintenance cost as:

$$C_M(n) = 3 \log \frac{N}{4}(p_{split}(n) + p_{merge}(n))$$

Suppose n has selected a set of R-tree nodes S for indexing. The total cost of the current index is estimated as:

$$\begin{aligned} C(S) &= \sum_{n \in S} C(n) \\ &= \sum_{n \in S} (C_{FP}(n, Q) + C_M(n, L)) \qquad (2) \\ &= \sum_{n \in S} \log \frac{N}{4}(|Q_{FP}(n)| + 3(p_{split}(n) + p_{merge}(n))) \end{aligned}$$

The aim of the index selection algorithm is to minimize this cost.

5.2 Index Selection

The index selection algorithm must guarantee the completeness and uniqueness of the index. Figure 5 shows an example of R-tree nodes selection, where dark colored nodes are selected. Based on the recent query and update patterns, the index selection algorithm should select a set of nodes satisfying the two properties and minimizing the total cost of the published index. The formal definition of index selection is:

Definition 3. *Given an R-tree T_R and its node set V, the weight of a node $n \in V$ is defined as $C(n)$, namely, its indexing cost. The index selection problem is to select a set of R-tree nodes S, where*

1. *$S \subseteq V$*
2. *$\forall n_i, n_j \in S, n_i$ is not the ancestor of n_j and vice versa*
3. *For a leaf node n_i of T_R, $n_i \in S$ or there is an ancestor of n_i in S*
4. *The weight of S, defined as Equation 2, is minimized.*

In Equation 2, $\log \frac{N}{4}$ is a constant for a specific overlay network. Therefore, we discard it and simplify the cost as:

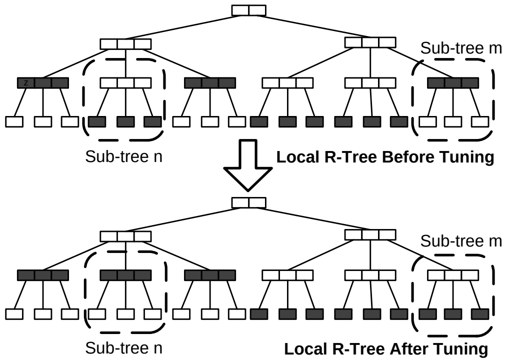
$$C(S) = \sum_{n \in S} (|Q_{FP}(n)| + 3(p_{split}(n) + p_{merge}(n)))$$

and the cost of n becomes

$$C(n) = |Q_{FP}(n)| + 3(p_{split}(n) + p_{merge}(n)).$$

Algorithm 5 R-Tree Node Selection Algorithm

```
Input: local R-Tree $T_R$
Output: A set of R-Tree nodes $S$
for each R-tree node $n_i$ in $T_R$ do
    update cost of $n_i$ using query and update histograms
end for
$\{S, C\}$=$IndexSelect(T_R.root)$ /* invoke Algorithm 6 */
return $S$
```



Index tuning while Sub-tree n is frequently updated, rarely queried and Sub-tree m is frequently queried, rarely updated

Figure 5: Selective Indexing

Algorithm 6 IndexSelect

```
    Input: R-Tree node $n$
    Output: A set of R-Tree nodes $S$, Total Cost $C$
 1: $S = \{n\}$
 2: $C = C(n)$
 3: if $n$ is a leaf then
 4:     return $S$ and $C$
 5: else
 6:     $S_{temp} = \emptyset$, $C_{temp} = 0$
 7:     for $\forall n_i \in n.child$ do
 8:         $\{S', C'\}$=$IndexSelect(n_i)$
 9:         $S_{temp} = S_{temp} \cup S'$
10:         $C_{temp} = C_{temp} + C'$
11:     end for
12:     if $C_{temp} < C(n)$ then
13:         return $S_{temp}$ and $C_{temp}$
14:     else
15:         return $S$ and $C$
16:     end if
17: end if
```

We use dynamic programming to achieve the minimal total cost by selecting an optimal node set S. For each $n \in V$, our algorithm computes the minimal total cost of the subtree rooted at n, denoted as $C(T_R, n)$. For each internal node n,

$$C(T_R, n) = min\{\Sigma_{n_i \in n.child} C(T_R, n_i), C(n)\}$$

$C(T_R, n)$ equals to $C(n)$, if n is a leaf node. Algorithm 5 describes the selection of R-tree nodes to minimize the cost. We first update the cost of all R-tree nodes, based on the most recent query and update patterns. Then, we invoke Algorithm 6 to recursively compute the optimal cost and selection. In Algorithm 6, if the node is a leaf node, it simply returns its cost and selects itself (lines 3 and 4). Otherwise, we compare the costs of two strategies, combining the optimal solutions of the child nodes and use the current node as the solution (lines 6-14). We select the solution with the least cost. The cost of Algorithm 6 is bounded by $O(|V|)$, where $|V|$ denotes the number of local R-tree nodes.

Theorem 3. *Algorithm 5 returns the optimal indexing set with regard to the query pattern and the update pattern.*

Proof. Because we apply the dynamic programming technique, for an R-tree node n, we only need to prove that the

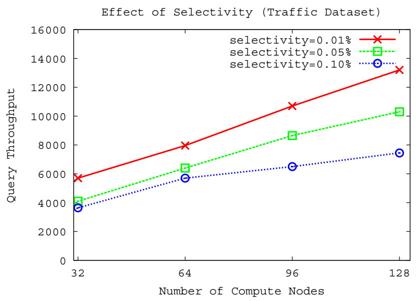


Figure 6: Effect of Selectivity (Traffic Dataset)

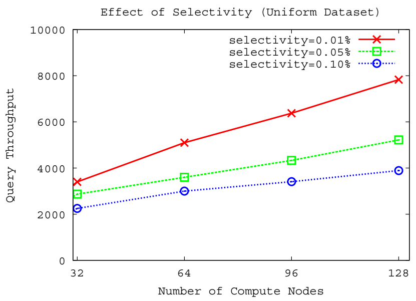


Figure 7: Effect of Selectivity (Uniform Dataset)

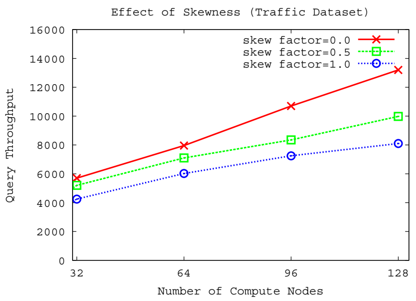


Figure 8: Effect of Skewness (Traffic Dataset)

optimal solution of n is either to index n or combine the sub-solutions of n's children. If indexing n is the best solution, the problem is solved. Otherwise, as the solution of one child does not affect the solution of another, the sub-solution is actually independent. Moreover, the total cost is the sum of all costs of the sub-solutions. Therefore, by combining all sub-optimal solutions, we can get the optimal solution. □

6. PERFORMANCE EVALUATION

We evaluate RT-CAN indexing scheme on Amazon's EC2 platform. We implement our system in Java 1.6.0.13 and run it on a set of EC2 computing units. Each computing unit is a small instance of EC2 with 1.7GHz Xeon processor, 1.7GB memory and 160GB hard disk. The computing units are connected via 250Mbps network links. The number of computing units in our system varies from 32 to 128. We use two different datasets to evaluate our indexing scheme. In the uniform dataset, we generate 500,000N objects, where N is the number of computing nodes. The object has 2 to 5 attributes and values of the attributes are uniformly distributed in the range $[0, 10^9]$. We partition the data into N_g equal-size grids, where N_g is several times of N. The grids and their corresponding data are randomly disseminated to the computing nodes. In this way, one computing node handles multiple grids and all computing nodes roughly maintain the same number of objects. In the traffic dataset, we apply [3] to generate 100,000N 2-d moving objects based on the city's map, which represents the real-time traffic. Objects in the traffic dataset follow skewed distribution. And we apply the same approach as in the uniform dataset to partition the data among computing nodes.

In each computing node, we build an R-tree index for its local data. The page size of the R-tree node is 4K and each internal node contains at most 127 entries. We test the performance by generating queries that follow both uniform and skewed distribution. We use zipfian distributions to generate skewed queries that focus on some hotspots. Table 1 lists the experiment parameters and their default values.

Query throughput (processed queries per-second) is used as the metric in the experiments. We use a master node to control the behaviors of all the other compute nodes. The master node partitions the data among the computing nodes and generates queries continuously. The computing node fetches queries from the master node. After finishing its current query, the computing node will pull a new query from the master. All experiments are running for 200 seconds and we collect statistics (number of processed queries) every 10 seconds. And the average results are used.

Table 1: Default Experiment Setting

Parameters	Default Values
Dimension	3
Query Type	Range Query
Query Selectivity	0.01%
Skew Factor	0
Index Tuning	Yes
Usage of Routing Cache	No
Number of Computing Nodes	128

6.1 Performance of Range Queries

Range queries are most popular queries in web 2.0 applications. Point query can be considered as a special type of range queries by setting the search range to zero. KNN queries are processed via a set of range queries. In this experiment, we evaluate the performance of range queries. We define the selectivity as the percent of searched space. For example, if we set $selectivity = 0.05\%$ in a 3-d space, the query will search approximately 8% range in each dimension. Figure 6 and Figure 7 show the performances of range queries with different selectivities and network sizes for traffic dataset and uniform dataset, respectively. When we increase the number of computing nodes, the query throughput increases almost linearly, which verifies the scalability of the RT-CAN index. A larger selectivity value causes more objects to be retrieved and more cluster servers to be involved in the query processing. Therefore, the performance degrades significantly. The results of traffic dataset are much better than those of the uniform dataset. This is because the traffic dataset has 2 dimensions, and the uniform dataset has 3 dimensions. Processing a query on the uniform dataset costs more routing hops and local disk searches than processing on the traffic dataset.

In real systems, queries follow skewed distributions (e.g. users could be more interested in photos about Liberty Statue, Eiffel Tower and Great Wall). In Figure 8 and Figure 9, we generate queries following zipfian distribution and test the performance of the RT-CAN index. When skewed factor is set to 0, the query follows a uniform distribution. When skewed factor is set to 1, about 80% queries focus on 20%

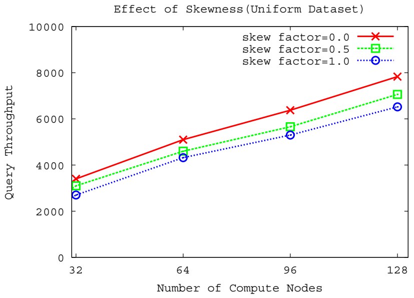


Figure 9: Effect of Skewness (Uniform Dataset)

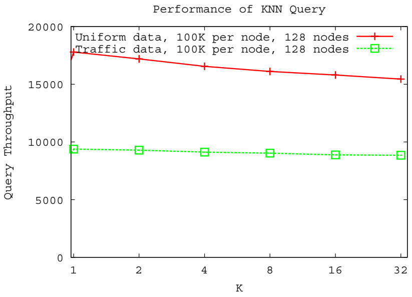


Figure 10: Performance of KNN Query (2-d Dataset)

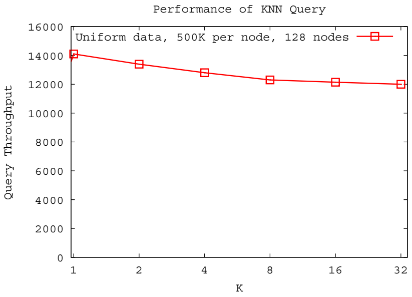


Figure 11: Performance of KNN Query (3-d Dataset)

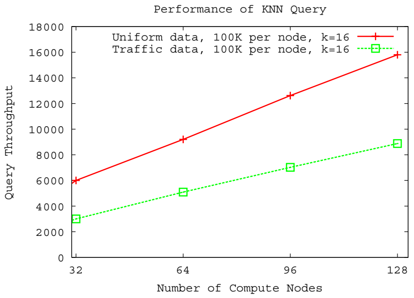


Figure 12: Performance of KNN Query (2-d Data Dateset)

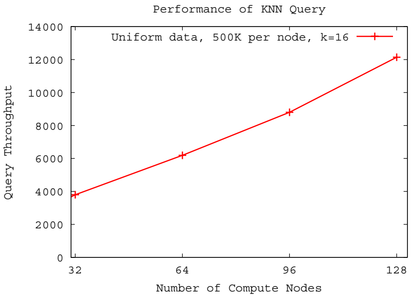


Figure 13: Performance of KNN Query (3-d Dataset)

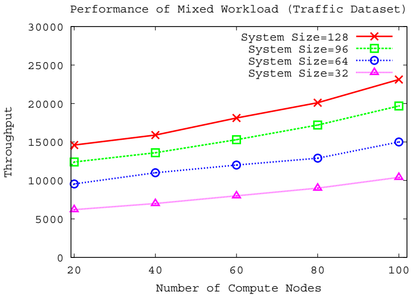


Figure 14: Performance of Update (Traffic Dataset)

objects. The experiments show that the RT-CAN index performs best for queries of uniform distribution, because the query skewness causes unbalanced workload among the servers. However, it can still provide a reasonable performance when both queries and objects follow skewed distributions (skewed factor=1 in the traffic dataset).

6.2 Performance of KNN Queries

The KNN query is more complex than the range query. There have been many proposals on pruning the search branches of an R-tree progressively. They are, however, not directly applicable to the RT-CAN structure; therefore, in this paper, a simple but efficient strategy is adopted. We first estimate the distance between the query point and its K-nearest neighbor based on the data distribution. The distance is used as the initial search radius. Then, we iteratively enlarge the search range until satisfied results are retrieved. In Figure 10 and Figure 11, we vary the value of K from 1 to 32 and show the performances of traffic dataset and uniform dataset, respectively. Figure 10 shows performance on traffic dataset (we also show the diagram of uniform dataset for comparison), where each node holds 100K 2-d data. Figure 11 shows the performance on uniform dataset, where each node holds 500K 3-d data. The throughput decreases when the parameter k increases, because the search space increases with k. More index entries will be involved and more local disk search will be incurred when k increases. The performance of traffic dataset is worse than the uniform dataset and it is also less effected by k, because of the skewness of data distribution.

Figure 12 and Figure 13 show the performance of KNN query processing at different system sizes, ranging from 32 nodes to 128 nodes. The KNN parameter is set to 16 in the test. As we increase the number of computing nodes, the performance improves dramatically. Comparing the results of KNN queries with the results of range queries, we find that range query performs better than KNN queries on traffic dataset, and KNN queries performs better than range queries on uniform dataset. This is because in the traffic dataset, KNN query needs to incur several iterations of range query to retrieve enough data. But for the uniform dataset, as the initial estimation of the search range is quite accurate, in most cases, only one range query is required to retrieve the top-k results. And for KNN query, the search range is much smaller than that of the range query.

6.3 Performance of Updates

As in real systems, queries and updates are processed concurrently, we generate a mixed workload by combining the queries and updates. In this experiment, we generate uniform insertions for the local R-trees. We vary the percentage of updates from 20% to 100%. Figure 14 and Figure 15 show the update performance of RT-CAN index for traffic dataset and uniform dataset, respectively. As a matter of fact, updates incur less overheads than queries. First, as we select a portion of R-tree nodes for indexing, we need to update the global index only when the indexed R-tree nodes have been modified. Second, to update an indexed R-tree nodes, we only need $3\log\frac{N}{4}$ routing messages, whereas to process a range query, besides $\log\frac{N}{4}$ routing messages, we need to search the local R-trees and return the corresponding results. Increasing the number of compute nodes leads to a better throughput. RT-CAN index can handle both queries and updates efficiently.

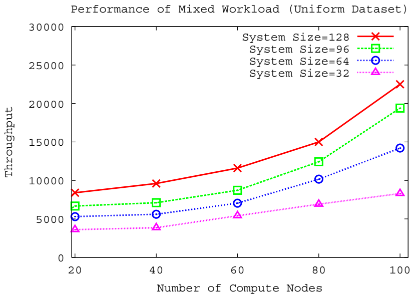


Figure 15: Performance of Update (Uniform Dataset)

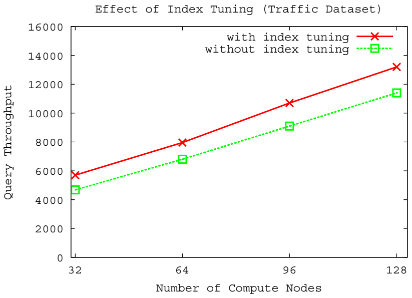


Figure 16: Effect of Tuning (Traffic Dataset)

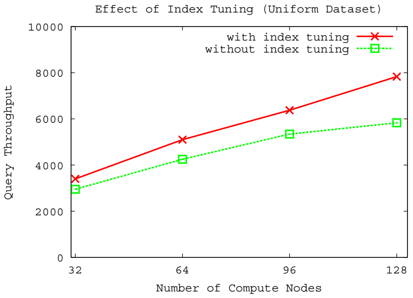


Figure 17: Effect of Tuning (Uniform Dataset)

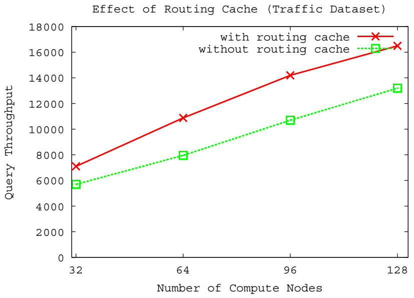


Figure 18: Effect of Cache (Traffic Dataset)

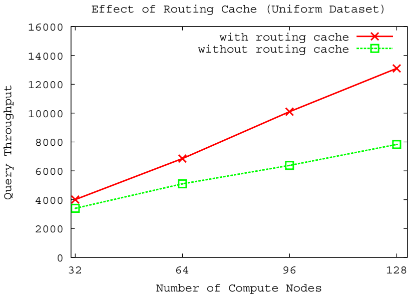


Figure 19: Effect of Cache (Uniform Dataset)

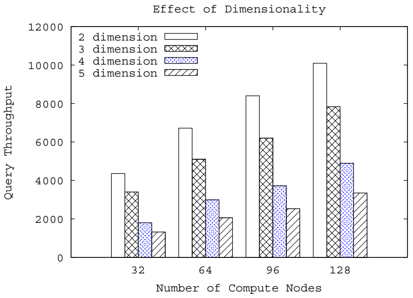


Figure 20: Effect of Dimensionality (Uniform Dataset)

6.4 Effect of Index Tuning

To balance the query processing cost (false positive cost) and index maintenance cost, a tuning approach is adopted in the RT-CAN index to dynamically change the indexing strategy. In our experiment, local R-trees have three levels. And initially, we publish the second-level nodes (root node's children) into the global index. After running for a while, RT-CAN index collects the query pattern and update pattern and apply the cost model to compute a new indexing strategy. Figure 16 and Figure 17 illustrate the tuning effect. When applying the tuning algorithm, we record the throughput, after the index becomes stable. We deliberately generate some skewed update operations to trigger the index tuning. The results show that index tuning can effectively reduce the cost by catching the query and update patterns.

6.5 Effect of Routing Cache

Message routing cost is bounded by $O(\log N)$ in the RT-CAN index. This is not a big issue in Peer-to-Peer network, but in the Cloud systems, due to its high concurrency and high throughput requirements, the routing cost must be reduced to $O(1)$. In Cloud systems, the overlay is stable. Nodes always stay online until hardware fails. Therefore, we apply routing cache to buffer the addresses of cluster servers and their key spaces in CAN. When routing a message, we first look up the routing cache. By checking the key spaces of each buffered server, we know whether it is the receiver of the message. If a server in the cache is exactly the message's destination, the message is sent out via the buffered information. Otherwise, the original RT-CAN's routing algorithm is applied. The cache is maintained in a lazy-update way. If a cache entry is stale and provides wrong routing information, we will invoke RT-CAN's routing algorithm to forward the message and update the entry based on the correct routing information.

Figure 18 and Figure 19 show the effect of routing cache. A significant improvement is observed for the cache-based schemes. As a matter of fact, some other Cloud systems, such as Dynamo [8], also adopt routing cache to improve the performance.

6.6 Effect of Dimensionality

The RT-CAN index is designed to support multi-dimensional relational data. In this experiment, we show its performance by varying the number of dimensions. As the traffic dataset is limited to 2 dimensions, we only present the results of the uniform dataset. Figure 20 shows the performances of the RT-CAN index in different dimensions. The best throughput is achieved for 2 dimensional dataset. The throughput decreases as the number of dimensionality increases. This is because for d-dimensional space, the range query will overlap with a large number of index items and hence incurs high communication costs. For example, in a 5-dimensional space, if query Q searches 0.01% of the whole space, we need to search about 16% of range in each dimension. A large search circle will be generated based on Algorithm 2.

7. CONCLUSION

This paper presents RT-CAN, a multi-dimensional indexing scheme, for our epiC[2] cloud system. RT-CAN is built on top of local R-tree indexes and provides efficient data retrieval service for large-scale shared-nothing clusters. Instead of indexing objects directly, RT-CAN dynamically selects a portion of local R-tree nodes to publish onto the global index. The global index is disseminated to different cluster servers, which are organized as a logical CAN-based overlay network. We propose algorithms for processing point, range and KNN queries. To reduce query cost and index maintenance cost, a dynamic tuning algorithm is proposed to adaptively select R-tree nodes for indexing. A cost model is used to collect the statistics about query and update patterns and estimate the cost of the current indexing strategy. Extensive experiments were conducted on Amazon's EC2. The results verify the effectiveness and efficiency of RT-CAN.

8. ACKNOWLEDGMENTS

The project epiC is in part funded by Singapore Ministry of Education Grant No. R-252-000-394-112 under the project name of Utab, China National Grand Fundamental Research 973 Grant 2006CB303000, Key Program of National Natural Science Foundation of China Grant 60773063, National High Technology Research and Development Program (863 Program) of China Grant 2009AA01Z149 and NSF C-RGC grant 60831160525.

9. REFERENCES

[1] http://hadoop.apache.org/.

[2] http://www.comp.nus.edu.sg/∼epic.

[3] http://www.fh-oow.de/institute/iapg/personen/brinkhoff/generator/.

[4] K. Aberer, P. Cudré-Mauroux, A. D. Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-grid: A self-organizing structured p2p system. In *SIGMOD 2003*.

[5] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[6] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP 2007*.

[7] S. A.Weil, S. A.Brandt, E. L.Miller, and D. D.E.Long. Ceph: A scalable, high-performance distributed file system. In *SODI 2006*.

[8] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, 2000.

[9] E. Bertino, B. C. Ooi, R. Sacks-Davis, K. Tan, J. Zobel, B. Shidlovsky, and B. Cantania. *Indexing Techniques for Advanced Database Applications*. Monograph series, Kluwer Academic, 1997.

[10] B.F.Cooper, R.Ramakrishnan, U.Srivastava, A.Silberstein, P.Bohannon, H.A.Jacobsen, N.Puz, D.Weaver, and R.Yerneni. Pnuts: Yahoo!'s hosted data serving platform. In *VLDB 2008*.

[11] W. Cai, S. Zhou, W. Qian, L. Xu, K. Tan, and A. Zhou. C2: a new overlay network based on can and chord. *Int. J. High Perform. Comput. Netw.*, 3(4):248–261, 2005.

[12] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[13] F. Chang, J. Dean, S.Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *OSDI 2003*.

[14] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *WebDB 2004*.

[15] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-ring: an efficient and robust p2p range index structure. In *SIGMOD 2007*.

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

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SIGOPS 2007*.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP 2003*.

[19] W. Gilks, S. Richardson, and D. Spiegelhalter. Markov chain monte carlo in practice. 1996.

[20] H. V. Jagadish, B. C. Ooi, K.-L. Tan, Q. H. Vu, and R. Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In *SIGMOD*, 2006.

[21] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *VLDB 2005*.

[22] J.Dean and S.Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI 2004*.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM 2001*.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms 2001*.

[25] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *SIGMOD 2008*.

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

[27] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE Trans. on Knowl. and Data Eng.*, 16(10):1169–1184, 2004.

[28] Q. H. Vu, M. Lupu, and B. C. Ooi. *Peer-To-Peer Computing: Principles And Applications*. Springer, November 2009.

[29] S. Wu and K.-L. Wu. An indexing framework for efficient retrieval on the cloud. *IEEE Data Engineering Bulletin*, 32(1):77–84, 2009.

[30] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, pages 1029–1040, 2007.