2011 Third IEEE International Conference on Coud Computing Technology and Science

A-Tree: Distributed Indexing of Multidimensional Data for Cloud Computing Environments

Andreas Papadopoulos Dimitrios Katsaros
Department of Computer & Communications Engineering
University of Thessaly, Greece
andpapad@inf.uth.gr, dkatsar@inf.uth.gr

***Abstract*—Efficient querying of huge volumes of multidimensional data stored in cloud computing systems has become a necessity, due to the widespread of cloud storage facilities. With clouds getting larger and available data growing larger and larger it is mandatory to develop fast, scalable and efficient indexing schemes. In this paper, we propose the A-tree, a distributed indexing scheme for multidimensional data capable of handling both point and range queries, appropriate for cloud computing environments. A performance evaluation of the A-tree against the state-of-the-art competitor attests its superiority, achieving significantly lower latencies.**

Distributed index, multidimensional data, point query, range query, query processing, cloud computing.

I. INTRODUCTION

Cloud computing and data centers are facing unprecedented challenges due to huge amount of data and number of users that must be handled. Several thousands of computers, terabytes of data and several millions of users comprise a typical cloud computing system, offered as SaaS, PaaS, or IaaS [1]. Every user allocates resources for his needs on demand from the "infinite" cloud, and pays only for what it was really used. Users and companies must as well consider the privacy of their data and cost for the services that will be used and select the most appropriate solution for their needs [2]. The amount of stored data and the rate of querying them, calls for new data structures which can satisfy the needs of a cloud system.

The majority of current cloud storage systems e.g., Google's GFS and BigTable [3], Hadoop's HDFS, and Amazon's DYNAMO are based on key-value pairs, and therefore they can only support point queries. Though this type of query is not rich enough to fulfill the needs of cloud users; more complex queries such as *range queries* are needed. Answering this type of queries becomes more complicated since the queried data are *multidimensional* in nature and spread among several cloud nodes.

Even though the past literature on databases and distributed systems is full of data structures capable of dealing with point and range queries for multidimensional data, the cloud environments poses new challenges that make these solutions inappropriate. First of all, cloud systems are *distributed over wide areas* – even across different countries – with a (usual) two-level hierarchy consisting of *master* and *slave nodes* and therefore the centralized database solutions are not an option. Secondly, proposals that are based on peer-to-peer overlay structures, such as CAN [4], P-Grid [5], BATON [6], are similarly not very efficient since they posses one or more of the following drawbacks: they do not support multidimensional data, or they require time-consuming, communication-hungry and careful balancing operations, or they do not differentiate among nodes. Finally, the recently proposed cloud-aware distributed structures, such as the EEMINC [14], incur high latencies.

In order to design a high performance distributed index for cloud environments, we must use a cost and space efficient indexing scheme capable of answering queries with low latency. Specifically, this article makes the following contributions:

- A new distributed indexing structure for cloud computing environments, the A-tree[1], is described, which is capable of answering both point and range queries. It is based on the combination of R-tree [16] and Bloom filters [17].
- We describe algorithms that distributed the index nodes to the cloud nodes, as well as well as the relevant insertion and deletion algorithms.
- A performance evaluation of the proposed structure against the competing state-of-the-art structure is conducted, which attest the superiority of the new structure.

The rest of the article is organized as follows: Section II describes the relevant work; in Section III, we introduce a request framework to describe the basic concepts of this work. Section IV and V provide the details of the local and global data structure that comprise the A-tree. Specifically, we describe the update strategy, how R-tree nodes are selected for being indexed in the global index, how updates are built and sent, and we also describe the process of update handling and the construction of global index. Section VI provides the experimental evaluation of the A-tree and its comparison against the state-of-the-art EEMINC. Our experimentations have shown that A-tree is a scalable, distributed, fast and space efficient index for multidimensional data supporting querying, insertion and deletion of records. Finally, Section VII concludes the present article.

[1] It is A tree based on Bloom filters for Clouds.

978-0-7695-4622-3/11 $26.00 © 2011 IEEE
DOI 10.1109/CloudCom.2011.61

II. Related Work

Firstly, we need to describe the A-tree's constituent structures, namely Bloom filter and R-tree, and then the relevant cloud indexing structures. Bloom filter is a bit array representing a set of items. For each item in a data set the hash values of independent hash functions are calculated and for every value the corresponding bit is set to 1. Using Bloom filter a point query can be answered in O(1) with a very small probability of false positives. Figure 1 demonstrates a simple Bloom filter with size ten bits and two hash functions are used: modulo three and modulo ten. Initially all values are zero and the diagram shows which bits turn on due to the insertion of elements 57, 83 and 94. Querying whether element 38 belongs to the set, we calculate both hash functions and we notice that the eighth bit is not turned on, which means that 38 does not belong to the set.

R-tree is a widely used data structure for multidimensional data. R-tree is the extension of B-tree to multidimensional data. Each node covers an area in the multidimensional space, usually represented as a hyper bounding box. For non-leaf nodes, this bounding box covers all of its children's bounding boxes. Leaf nodes contain pointers to the data. A lot of varieties have been studied including R^+-tree and R*-tree with most of them trying to minimize overlap between nodes' coverage. Figure 2 demonstrates a simple example of an R-tree in two dimensional space using rectangles to handle space partitions. R-tree is extensively being used on many indexing systems including geographical information systems to provide spatial index operations in the cloud [7].

A solution to provide query services over peer-to-peer networks was the Distributed Hash Table (DHT) and some variances [8]. Another relevant data structure is a scalable distributed B-tree [9]. With these data structures though, we can only answer point queries and not range queries over multidimensional data. As the Map-Reduce framework is widely used on cloud systems, a decent index structure was proposed [10]. Although this solution is claimed to be suitable for large datasets, the Map-Reduce framework itself adds a significant overhead during processing, which makes it inappropriate for extremely huge datasets. In addition efforts to support standard SQL statements with high performance global index have been made [11]. Other relevant indices include the RT-CAN [12] and the BR-tree [13]. Although these structures can answer both point and range queries, their drawback is that they organize the cloud nodes as a structured peer-to-peer network, which requires communication overhead and moreover they ignore the inherent partitioning of the cloud nodes as master and slaves. Therefore, they are not appropriate (native) for cloud computing systems.

The most relevant work to ours is the EEMINC index structure [14]. It exploits the separation of the cloud nodes as masters and slaves. All nodes with data, i.e., the slaves, are indexed using a KD-tree as a local index. Every node is described by a node cube, i.e., a set of ranges for every attributes of the data. E.g., {(10,20),(10,40),(30,80)} is a node cube covering two dimensional data where the first, second and third attributes range from ten to twenty, from ten to forty and from thirty to eighty, respectively. As a node cube usually is very large to avoid forwarding queries to irrelevant nodes, each node splits its own node cube to smaller ones and all divisions which cover some data are sent over the network to master nodes. Node cubes that are sent to master nodes are organized in an R-tree structure which comprises the global index. Upon a request arrival at a master node, the local R-tree is searched to find the nodes where potential results are stored. After accumulating a set of 'candidate' nodes, the request is forwarded to them for further processing.

Although EEMINC took a first step towards addressing the problem of indexing multidimensional data in clouds in a native way, it is inefficient since it results in a high number of false positives, especially for point queries, which in turn implies larger latencies during query processing.

III. Request-Response Framework

In this section we describe the workflow during query processing. Nodes are categorized as *master* nodes and *slave* nodes. The difference between the two categories is that if a node is a master node, it maintains some metadata about the system. This is a typical approach in the clouds, e.g., in Map-Reduce frameworks, since it makes a lot of operations easier and more efficient, despite the fact that the whole system becomes 'less' distributed.

In a cloud system, users will communicate and request results from the master nodes. These nodes will process the request and forward it to the appropriate nodes on the platform. After this step, any communication with the master node terminates, and the users will communicate only with the corresponding slave nodes. If a query is forwarded to a node,

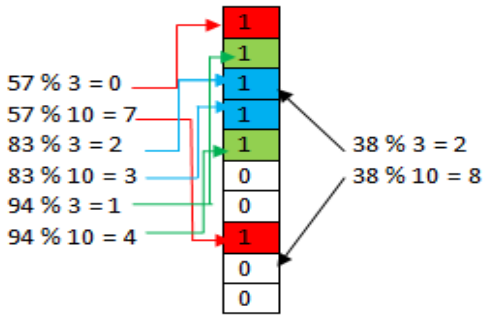


Figure 1. Example of a Bloom filter.

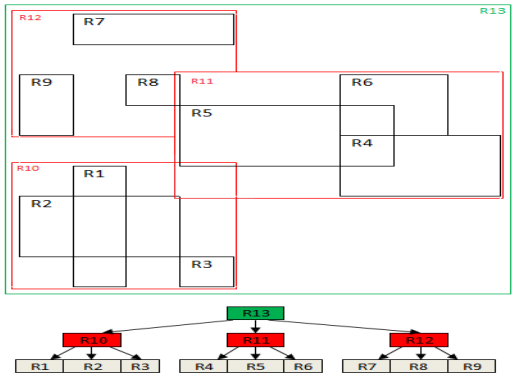


Figure 2. A sample R-tree in 2-D space.

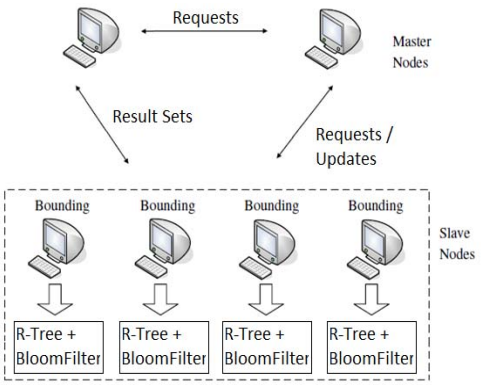


Figure 3. The framework for query processing.

and after executing it, the result set is empty, then it is said that a *false positive* occurred. For a cloud index to be successful, *false positives should be as low as possible*, because this also implies small latencies.

In this framework (Figure 3), the task of query processing on a master node can be divided into two phases: a) locate relevant nodes, and b) forward the query to these nodes. Relevant nodes will process the received query locally, and will return the results directly to the user, avoiding thus unnecessary resource consumption on master nodes.

Every slave node is responsible to share any data updates with all master nodes which fully cover the data located at this particular slave node. An update is sent after a node starts up and – according to the changes that will be done – partial or full updates will be exchanged. In case of deletions and insertions, it is mandatory that master nodes must be informed about them. Another important aspect, which is beyond the scope of this paper, is load balancing. Load balancing must be given attention too, and there are already some proposals such as PASSION [15], which can be easily adopted to raise the performance of the described framework.

Our indexing scheme, A-tree, is composed of an R-tree and a Bloom filter on each slave node and an array of updates on each master node (which is different than the approach of EEMINC), as shown in figure 3. In order to introduce how master nodes handle requests we must first introduce how slave nodes handle data and how updates are sent and maintained.

IV. Operations on Slave Nodes

A. Local Operations on Slave Node

Slave nodes are the place where actually data are and query processing is performed. The data structure that will be used on the slave nodes must be capable to handle point and range query, as well as insertion and deletion of records. For this purpose we use the R-tree data structure. Because an R-tree uses bounding boxes to describe a node's coverage, we will have *high false positives for point queries*. To avoid this we also use a Bloom filter, a very space efficient structure which allows with O(1) time complexity to determine if a point belongs to a dataset or not with a small probability of false positive.

The operations and algorithms for searching, inserting and deleting data are those that involve the R-tree, and moreover for every insertion on the data set, we add the record to the Bloom filter of the node. For point and range queries a traditional search over the R-tree costs O(log n). This is the cost for range queries, but for point queries Bloom filters are used. If a point does not belong in the dataset, then the search using the R-tree structure is avoided, resulting in faster point queries processing. For points that according to Bloom filter belong to the dataset, then the R-tree is searched to avoid false positives. If the system allows a small probability of errors, then only the Bloom filter is used resulting in O(1) cost for point query.

Record deletion is more complicated due to the fact that simple Bloom filter does not support deletion of data. Deletions are handled as well by our indexing system, even though deletion is not very common in cloud computing platforms. The process is described in the next section.

B. Distributing R-Tree Nodes

In this section we describe our algorithm for selecting and distributing R-tree nodes to master nodes.

Every slave node must structure an update and send it over the network to all master nodes. As every R-tree node is described by a hyper bounding box, an update is a combination of some hyper bounding boxes. To avoid high false positives rate for point queries, the Bloom filter of the node is also included in the update that will be sent.

With the assumption that the number of queries that will be forwarded to a node for processing are proportional to the volume of the hyper bounding boxes included in the update, we developed an algorithm (Algorithm 1) for calculating the benefit of indexing the children of an R-tree node.

Now that we are able to determine the benefit of distributing a node, Algorithm 2 is proposed for selecting the R-tree nodes that will be distributed. In real cloud computing systems, an R-tree may be configured with a high fan out, e.g., 100. We have a tradeoff between how many nodes to index, as more nodes will result to higher resource consumption on master nodes for finding relevant nodes to a query. As this is mostly depended on the system, we use as parameters the maximum number of hyper bounding boxes and the minimum

ALGORITHM 1. BENEFIT FOR INDEXING CHILDREN OF NODE N

```
1.  procedure double BenefitForIndexingChildren(Node n)
2.
3.  children_volume = 0.0;
4.  children_overlap_volume = 0.0;
5.
6.  for ( i = 0; i < #children; i++ ) do
7.        childrens_volume += child[i].getVolume();
8.        for ( j = i+1; j < #children; j++ ) do
9.           children_overlap_volume +=
    child[i].overlapVolume(child[j]);
10.       end for
11. end for
12.
13. benefit = n.getVolume() – (children_volume –
    children_overlap_volume);
14. return benefit;
```

ALGORITHM 2. SELECT INDEXING NODES

```
1.  procedure SelectIndexNodes( Node n, SET<BoundingBox>
    indexNodes,  int remainingSpace, double minBenefit)

2.  if ( n.getBenefitForIndexingChildrens() > minBenefit &&
    !n.hasLeafChild()        && n.#childrens <=
    remainingSpace ) then
3.      int currentSize = indexNodes.size();
4.      int more_free_slots = 0;
5.      int perChild = remainingSpace / n. #children;
6.      for ( int i = 0; i < n. #children; i++ ) do
7.          SelectIndexNodes(n.child[i], indexNodes, perChild,
    minBenefit);
8.          more_free_slots = (indexNodes.size() – currentSize);
9.          perChild += more_free_slots / (n. #children – i);
10.         currentSize = indexNodes.size();
11.     end for
12. else
13.     indexNodes.add(n.boundingBox);
14. endif
```

expected benefit for indexing children. If we index only the root, we have a large number of high false positives, and if we index all tree nodes above leaves, we have a high processing cost for locating relevant nodes.

It is also noted that we will not index leaf nodes because we will end up with all the data in the global index, if we set the update space high and the benefit low. Algorithm 2, in the worst case, has a complexity of O(|V|), where V is the number of nodes in the R-tree.

Upon starting the system every node executes Algorithm 2 to determine which tree nodes should be distributed. The selected nodes are distributed to the master nodes along with the Bloom filter of the node. An update is executed at time O(|V|)+Θ(n), where n is the number of master nodes running on the system.

C. Partial Updates

The aforementioned two algorithms describe how to select nodes for distribution. Upon insertion or deletion, a change on a local R-tree may occur. To deal with these changes we use partial updates.

Upon a deletion request we delete appropriate records from the tree and also keep a counter of deleted records. When counter reaches a threshold we update Bloom filter and invoke algorithm 2 to send a new update. It is recommended that the threshold is set dynamically according to work load on the system. To deal with insertions each node keeps an extra hyper bounding box. If newly inserted record is not cover by the previously sent update, the extra bounding box is expanded to cover it. As partial update this extra bounding box is sent with the Bloom filter if it changed. Because the volume of this extra bounding box may get very big we set a threshold. If the threshold is reached Algorithm 2 is invoked to send a new update and reduce the number of false positives.

V. OPERATIONS ON MASTER NODES

A. Construction of Global Index

In this section we describe how master nodes handle the updates and build the global index. The approach used here is to keep it as simple as possible in order to obtain fast processing and low complexity as well as low storage space.

Every master node maintains an array of updates received by slave nodes. Each index on this array is assigned to a slave node and holds the last update that was sent from this node. Organizing updates in an array allows access to a particular node's update in O(1) as well as avoiding the need for more space for pointers. This is extremely useful when processing a newly arrived update.

Upon an update *u* arrival from node with id *n_id* the only required action is to store it: *global_index[n_id] = u*. Similar action is required for a partial update with the difference that only the extra bounding box is changed and maybe the Bloom filter too. It is noted that this is lowest possible cost for handling updates.

Even though the number of master nodes is small compared to the number of slave nodes, master nodes must keep the same copy of the global index. To achieve data consistency, some already proposed methods can be used [18].

B. Processing of Point and Range Queries

According to the described framework, processing a request at a master node is equivalent to locating relevant nodes and forwarding the request to them.

Due to the fact that an update is a combination of a Bloom filter and some bounding boxes, the process for finding relevant nodes for a point query or a range query are not identical. Using Bloom filters, included in the updates, the process of finding relevant nodes can be performed with complexity Θ(n) where n is the number of nodes in the system. For range queries, Bloom filters cannot be used and the bounding boxes must be searched. Furthermore if a value for an attribute is not specified then the lowest and the highest values of the data in the dataset are used. In the worst case the

ALGORITHM 3 FIND RELEVANT NODES FOR A POINT QUERY

```
1.  procedure SET GetRelativeNodesPointQuery(Point p)
2.
3.  SET nodes = {};
4.  for each (UPDATE u in global Index) do
5.      if u.bloomfilter.membershipTest(p) then
6.          nodes.add(node_id);
7.      end if
8.  end for
9.  return nodes;
```

ALGORITHM 4. FIND RELEVANT NODES FOR A RANGE QUERY

```
1.  procedure SET GetRelativeNodesRangeQuery(Query r)
2.
3.  SET nodes = {};
4.  for each (UPDATE u in global Index) do
5.      for each (Bounding Box box in u) do
6.          if (box.overlaps(q) then
7.              nodes.add(u.node_id);
8.              continue with next Update;
9.          end if
10.     end for
11. end for
12. return nodes;
```

complexity is Θ(N*B) and in the best case Θ(N), where B is the number of bounding boxes in a stored update. Algorithm 3 and Algorithm 4 show the process for finding the set of relevant nodes to a point query and a range query, respectively.

C. *Insertion of New Records*

To process an insertion request the same procedure is followed. Master node which received the request must first find a set of relevant nodes and forward it to them. Algorithm 1 is used in order to avoid triggering a new update. As nodes which already cover the record are selected the insertion will not trigger a new update, except if the number of relevant nodes is less than the replication factor of the platform in which case more nodes can be added randomly or by choosing nodes that cover near data, using k nearest neighbors algorithm over the bounding boxes of the updates in the global index. If a new update is sent then it is handled as described previously with O(1) time cost at master nodes.

D. *Deletion Of Records*

In order to process a deletion request the steps are similar to the above. Relevant nodes are located and request is forward to them for further processing. Each node after deletion will check if it is time to send partial update as described above.

VI. Performance Evaluation

In this section we evaluate the performance and scalability of A-tree in comparison to EEMINC, since this is the state-of-the-art cloud computing-native indexing structure. The GridSim toolkit [19], an open source and widely used simulation tool, was used to simulate a data intensive grid environment. Simulation allowed us to measure point and range queries latency taking into consideration network delay according to links capacity as well as the size of the dataset and the number of nodes in the system.

In all experiments, the EEMINC's uniform cutting for node cubes was used and three-dimensional uniformly distributed data. R-tree fan out was set to fifty and for A-tree and Bloom filter the size was 30.5 bytes. Each node was a simulation resource, extending GridSim resource class, with network capabilities. Network was consisted of five routers and four links. All master nodes were connected to one router with a link of 100Mbps. This router had links to two other routers where all slave nodes were connected with 100 Mbps speed. The speed of the links between the routers was initially 1 Gbps and altered later to measure the effect of network speed on performance. Users were connected to two other routers with

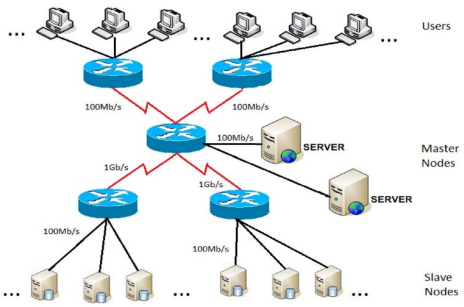


Figure 4. Network topology used for the experiments.

links to the master nodes' router with speed 100 Mbps. Each user' link to the router is 100 Mbps. MTU was set to 1500 bytes for all links. Network topology is shown on figure 4. A user sends requests to a specific master node, defined at initialization state, and receives results from any slave nodes. After the result for the previous request returns then another request is sent until the predefined number of requests is reached. The latency for each request is recorded and when all users finish the last one calculates the total average latency. All master nodes receive request from equal number of users. Each slave node also keeps two counters for false positives for point and range queries.

Experiments were run on a computer system equipped with four Quad-Core AMD Opteron(tm) Processor 8350, 16 gigabytes ram and 320 gigabytes storage capacity. Operation system was SUSE Linux Enterprise Server 10 (x86_64) running kernel version 2.6.16.21-0.8-smp. All experiments ran five times and average is used for plotting the graphs.

In the first experiment, the latency for point and range queries is measured as the number of nodes in the system increases. The total average of false positives is also measured as it is very important for minimizing network usage, and resulting in faster query processing if data set does not contain any response results. Fifteen users are connected to the system and each one executes one hundred point queries and one hundred range queries. There are five master nodes, each one responsible for three users. Each slave node is responsible for ten thousand records meaning that the size of the collection increases as the number of nodes increases. Due to the use of Bloom filters, point queries are forwarded to slave nodes with a very small probability of false positive, resulting in a dramatic decrease of resource consumption and lower average latency for point queries. Unfortunately for range queries Bloom filters cannot be used, but still the A-tree performs faster especially for large number of slave nodes showing that the proposed algorithm for distributing R-tree nodes is efficient. Figure 5(a) and figure 5(b) show the average latency for point and range queries respectively revealing that A-tree can handle point and range queries efficiently with very low latency, especially for point queries.

In order to evaluate how the number of master nodes affects queries' latency, the same experiment was repeated five more times with the difference that ten master nodes exist now in the system. Results are shown in figure 5(c) and figure 5(d). Firstly we notice that latency is lower for A-tree in contrast to EEMINC where latency is higher when more master nodes exist in the system. Furthermore we notice that for small number of nodes with ten master nodes, latency for range queries is almost the same with EEMINC but as the system gets larger A-tree performs better.

Concerning point queries, EEMINC performs worst when more nodes exist in the system, in contrast to A-tree where a little performance is gained. Table I shows the total average number of false positives for range and point queries. False positives for range queries are very close but the difference for point queries is very high because of the use of Bloom filters. Although the difference of false positives rates for range queries is not high, the latency is getting higher for more than one hundred nodes in the system. Even few less false positives can have a huge impact on large systems where possible disk

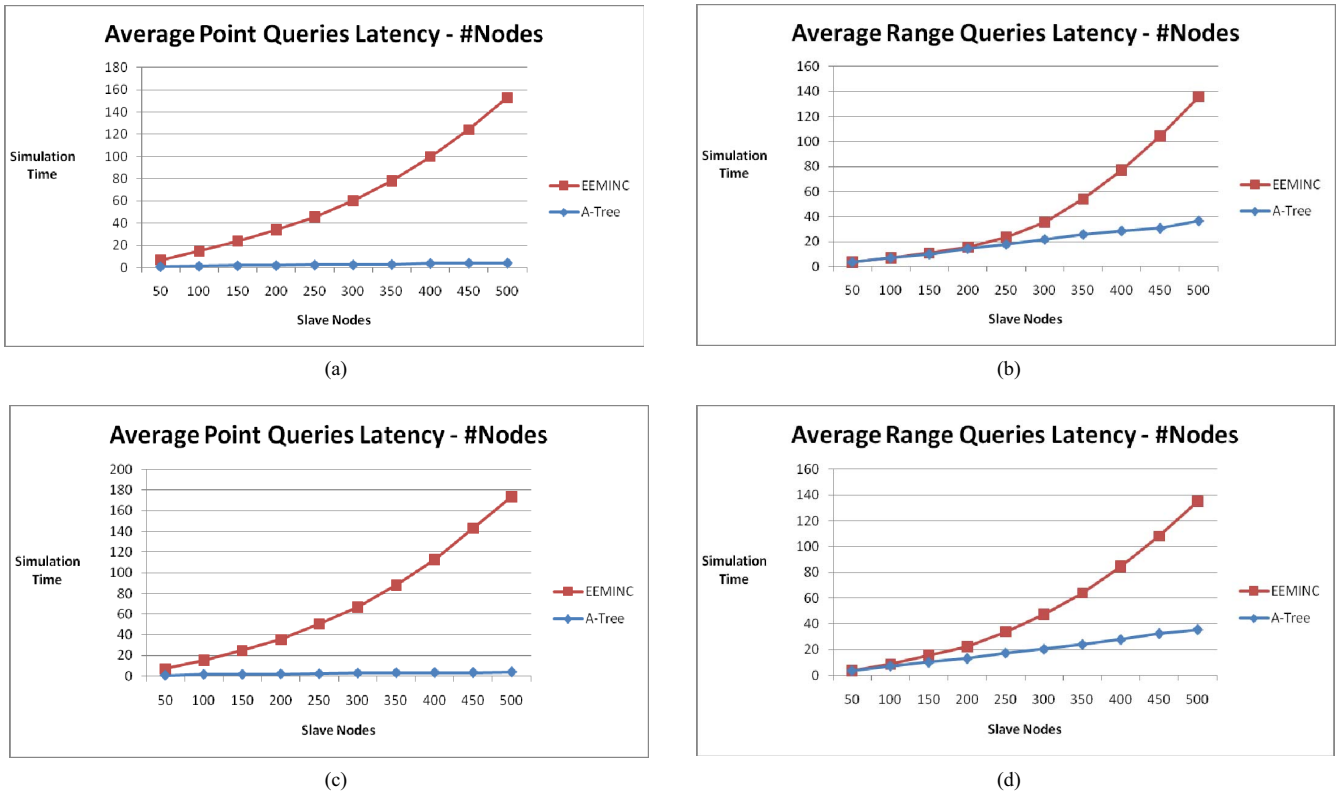


Figure 5. Average point and range queries latency vs. #nodes: (a) average point queries latency with five master nodes (b) average range queries latency with five master nodes (c) average point queries latency with ten master nodes (d) average range queries latency with ten master nodes

TABLE I. AVERAGE FALSE POSITIVES AS NODES INCREASE WITH TEN MASTERS

#Nodes	Range queries		Point queries	
	A-Tree	*EEMINC*	*A-Tree*	*EEMINC*
100	49813	49707	3	49943
200	74809	74972	7	74894
300	99861	99958	10	99931
400	124702	124830	11	124484
500	149583	149716	13	149836

accesses can be avoided as well as network resources. The most important conclusion from the above experiments is that *A-tree scales linearly as system is getting larger and responds faster to requests with more master nodes in contrast to EEMINC where exponential behavior was recorded.*

Another very important ascpect for a global index performanace is how latency is affected by the amount of data records handled by each node. In the following lines, the performance of A-tree is measured as the data set is getting bigger. Although in the previous experiment, the data set was getting larger as nodes increase, now the number of nodes remains the same and the data stored on each node increase. The configuration for this experiment is: five master nodes, fifteen users with each one making fifty point queries and fifty range queries, the number of slave nodes was set to fifty and data size was set from fifteen thousands records to seventy five thousands records per slave node, resulting to total 3750000 records.

We notice that there is a small increase in latency for A-tree as the collection is getting bigger due to high number of false positives, especially for point queries. This is due to the fact that the Bloom filter, which was set to default size of 30.5 Kb, is getting full and false positives probability is getting higher. The average latency for range and point queries is shown on figure 6. A-tree is still capable of answering both point and range queries efficiently and scales as the system and data set are getting larger. Despite the fact that false positives rates, for both point and range queries, are lower in A-tree than in EEMINC, this small increasment for point queries along with the fact that data set is larger on every nodes are the extenuation for the small increasement of latency. This small increase of latency can be easily avoided by using bigger Bloom filters and setting the maximum number of bounding boxes in updates higher.

Comparing the results from previous experiments, we notice that we have better performance when less nodes are used with more records on each of them. This behaviour is somehow expceted because when more nodes with less records exist, the whole system is not fully utilized. As far as each node is not overloaded, it is shown that adding more records to the node will lower latencies and increase performance of the whole system.

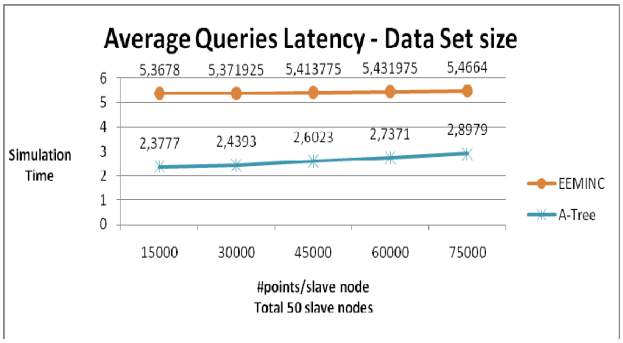


Figure 6. Average queries latency as data set size increases.

TABLE II. AVERAGE FALSE POSITIVES AS DATA SET SIZE INCREASE

#Records/ Node	Range queries		Point queries	
	A-tree	*EEMINC*	*A-tree*	*EEMINC*
15000	37325	37339	199	37434
30000	37189	37212	707	37500
45000	37049	37123	1509	37500
60000	36911	36918	2516	37481
75000	36791	36803	3577	37499

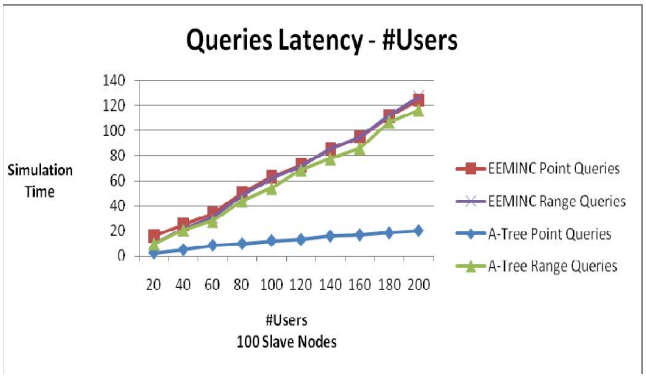
The next experiment's purpose is to measure latency as the number of users on the system increases. The number of users in our simulation scenario is equivalent to the number of queries being concurrently processed on the system. The system size was set to one hundred slave nodes, each one responsible for ten thousand records, and five master nodes. The other parameters are set to the same values as in previous experiments. In this experiment the affection of the number of master nodes is also measured. Figure(a) and Figure(b) show the results of this experiment for five and ten master nodes respectively. According to this experiments, the A-tree also scales as the number of users – concurrent queries increases. We notice the large difference for point queries latency between A-tree and EEMINC for both configurations. Conserning range queries A-tree performs slightly faster. It is also noticeable that the number of master nodes slighly affect the queries latency, driving us to the coclusion that only very few master nodes are necessary and that increasing the number of master nodes will only gain a little to nothing performance. Furthermore, in order to gain more performance and lower latency the search procedure on slave nodes should be further optimized.

Another very important aspect is the capacity and speed of the network. An efficient index should not be affected very much from changes to networks connections and links speed. The purpose of next experiment is to evaluate how the links speed affect latency. The same network topology shown in figure 4 is used with links speed altered. As speed from users to a datacenter is mostly an external factor, only the speed of links between masters and slaves nodes routers has been changed from 1 Gbps to 100 Mbps. For this experiment ten master nodes exists. Figure 8(a) shows the average latency for point queries for both network configurations. As expected latency is higher, because of limited network capacity, but A-tree is not affected very much from this change. In constranst to latency for range queries, shown in figure 8(b), where A-tree is more affected by network change but still performs better especially for large systems.

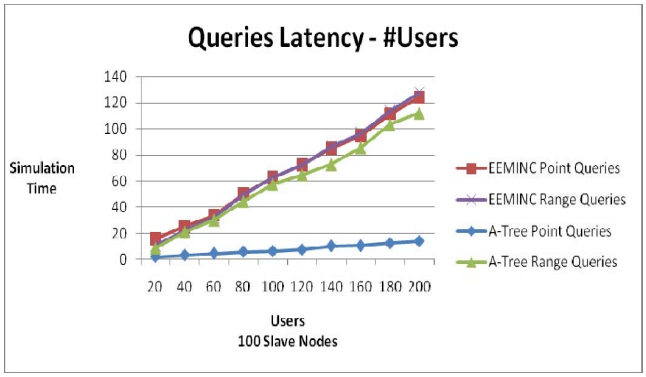
Figure 9 show the latency for point and range queries for both above network configurations. Ten master nodes and one hundred slave nodes, each responsible for ten thousand records, exist in the system. Each user makes one hundred point queries and one hundred range queries. Average latency as the number of users in the system increaces is shown in Figure 9 revealing that A-tree is not affected very much from network changes as more users send requests simultaniously. This behaviour is expected as network usage is limited especially for point queries.

VII. CONCLUSIONS AND FUTURE WORK

In this paper the A-tree is presented. The proposed A-tree is a combination of R-tree and Bloom filter and support storing and querying large multidimensional data sets in large datacenters. Based on the update strategy introduced, it is capable of fast processing of point and range queries. Our experiments proved that A-tree is an efficient, distributed data structure for multidimensional data stored in clouds, which scales linearly as the system grows larger and data volume is

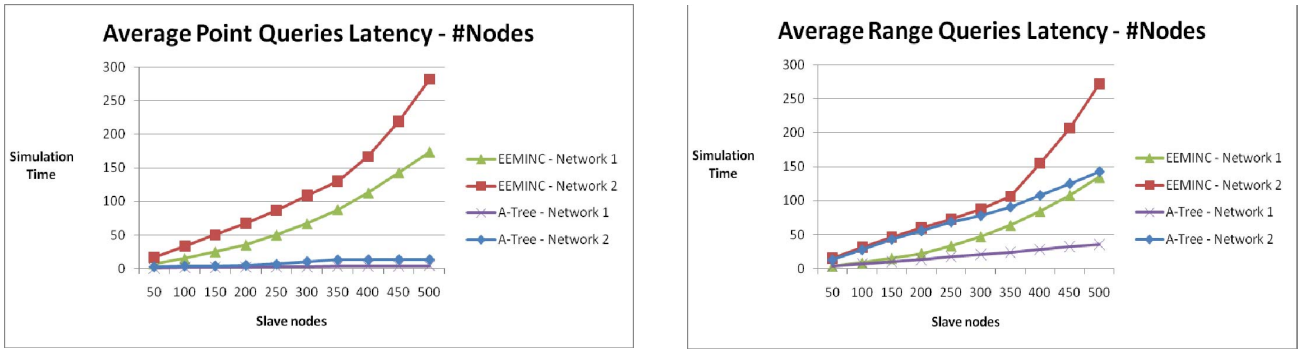


(a)



(b)

Figure 7. Average queries latency as number of users increases. (a) with five master nodes (b) with ten master nodes



(a) (b)

Figure 8. Average queries latency for different network speeds with ten master nodes: (a) average latency for point queries (b) average latency for range queries.

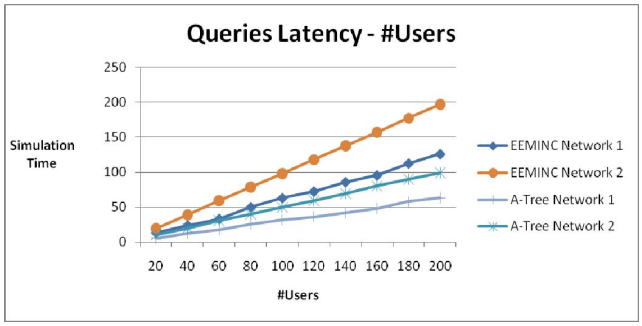


Figure 9. Average queries latency as users increase for different networks.

getting bigger. In addition, the A-tree is capable of handling a lot of requests at the same time taking full advantage of datacenter resources. As network usage is minimized, the A-tree performs very well for slow network configurations, especially for point queries.

For future work, the use of dynamic Bloom filters will be studied as well as the use of R*-tree. It is expected, that both changes will lower false positives' rates for both point and range queries, resulting in a tradeoff between more essential space and preprocessing time and faster query processing. We are also considering deployment to a real cloud environment such as Amazon EC2 for further evaluation and optimizations.

ACKNOWLEDGEMENT

The authors acknowledge the support of the European Commission through the FET STREP project STAMINA (FP7-265496).

REFERENCES

[1] M.D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, A. Vakali. "Cloud Computing: Distributed Internet Computing for IT and Scientific Research", *IEEE Internet Computing*, vol. 13, no. 5, pp. 10-13, 2009.

[2] S. Bibi, D. Katsaros, P. Bozanis. "Business application acquisition: On-premise or SaaS-based solutions?", *IEEE Software*, March/April 2012.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "BigTable: A distributed storage system for structured data," in *Proceedings of USENIX OSDI*, pp. 205–218, 2006.

[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable netetwork", in *Proceedings of ACM SIGCOMM*, 2001.

[5] K. Aberer, P. Cudre-Mauroux, A.D.Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt, "P-grid: A self-organizing structured p2p system", in *Proceedings of ACM SIGMOD*, 2003.

[6] H.V. Jagadish, B.C. Ooi, and Q.H. Vu, "BATON: A balanced tree structure for peer-to-peer networks", in *Proceedings of VLDB*, 2005.

[7] Y.G. Wang, S. Wang and D.L. Zhou, "Retrieving and indexing spatial data in the cloud computing environment", in *Proceedings of IEEE CloudCom*, 2009.

[8] "Distributed Hash Tables Links", http://www.etse.urv.es/cpairot/dhts.html, 2009.

[9] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed B-tree," in *Proceedings of VLDB*, pp. 598–609, 2008.

[10] I. Konstantinou, E. Angelou, D. Tsoumakos and N. Koziris "Distributed indexing of Web scale datasets for the cloud" in *Proceedings of MDAC*, 2010.

[11] J. Qi, L. Qian, and Z. Luo, "Distributed structured database system HugeTable", in *Proceedings of IEEE CloudCom* , 2009.

[12] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi., "Indexing multi-dimensional data in a cloud system", In *Proceedings of ACM SIGMOD*, pp. 591–602, 2010.

[13] Y. Hua, B. Xiao, J. Wang "BR-tree: A scalable prototype for supporting multiple queries of multidimensional data", *IEEE Transactions on Computers*, vol. 58, no. 12, 2009.

[14] X. Zhang, J. Ai, Z. Wang, J. Lu, X. Meng "An efficient multi-dimensional index for cloud data management", In *Proceedings of CloudDB*, pp. 17-24, 2009.

[15] I. Konstantinou and D. Tsoumakos and N. Koziris "Measuring the cost of online load-balancing in distributed range-queriable systems" In *Proceedings of P2P Computing*, 2009.

[16] A Guttman, "R-trees: A dynamic index structure for spatial searching", *ACM SIGMOD Record*, vol. 14, no. 2, pp. 47-57, 1984.

[17] B.H. Bloom, "Space/time trade-offs in hash coding with allowable errors", *Communications of the ACM*, vol. 13, no. 7, pp. 422-426, 1970.

[18] M.A. Islam, S.V. Vrbsky, "Tree-based consistency approach for cloud databases", in *Proceedings of IEEE CloudCom*, 2010.

[19] A. Sulistio, U. Cibej, S. Venugopal, B. Robic and R. Buyya "A toolkit for modelling and simulating data grids: An extension to GridSim", *Concurrency and Computation: Practice and Experience*, vol. 20, no. 13, pp. 1591-1609, 2008