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

Andreas Papadopoulos and Dimitrios Katsaros

Abstract—Efficient querying of huge volumes of multi-dimensional 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 present the A-Tree, a scalable distributed indexing scheme for multi-dimensional data capable of handling both point and range queries, appropriate for cloud computing environments, to also support insertions and deletions of multi-dimensional data. A performance evaluation of the A-Tree against the state-of-the-art competitor attests its superiority, achieving significantly lowers latencies for both point and range queries as well as insertions.

Index Terms—Distributed architectures, Indexing methods, Query processing

1 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 [9]. 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 [5]. The amount of stored data and the rate of querying them, calls for new data structures that can satisfy the needs of a cloud system.

The majority of current cloud storage systems e.g., Google's GFS and BigTable [7], 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 fulfil 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 multi-dimensional in nature and spread among several cloud nodes.

As an example, in a cloud based employees database a query could be a request for all the 35 years old employeers with $30K salary (point query). In another real world scenario, a user would ask for all the employees aged above 25 with salary between $25K and $35K (range query).

- *Andreas Papadopoulos is with the Department of Computer Science, University of Cyprus. E-mail: andpapad@cs.ucy.ac.cy*
- *Dimitrios Katsaros is with the Department of Computer and Communication Engineering, University of Thessaly. E-mail: dkatsar@inf.uth.gr*

Even though the past literature on databases and distributed systems is full of data structures capable of dealing with point and range queries for multi-dimensional 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 (usually) 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 [20], P-Grid [2], BATON [14], are similarly not very efficient since they possess one or more of the following drawbacks: they do not support multi-dimensional 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 [27], incur high resources consumption and 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. A-Tree is capable of answering both point and range queries with low latency. A-Tree further supports the insertions and deletions of data also with low latency. It is based on the combination of R-Tree [11] and Bloom filters [6].
- We describe algorithms that distribute the index nodes to the cloud nodes, as well as well as the relevant insertion and deletion algorithms.
- A performance evaluation of the proposed struc-

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

ture against the competing state-of-the-art structure is conducted, which attests the superiority of the proposed structure.

The rest of the article is organized as follows: Section 2 describes the relevant work; in Section 3 we introduce a request framework to describe the basic concepts of this work. Section 4 and 5 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 6 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 multi-dimensional data in cloud environments supporting point and range querying as well as insertion and deletion of records. Finally, Section 7 concludes the present article.

2 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 [6]. 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. Bloom filters are being widely used in many areas, such as databases and distributed systems [24]. Using Bloom filter a point query can be answered in $O(1)$ with a very small probability of false positives that mostly depends on its size. Figure 1 demonstrates a simple Bloom filter with size ten bits and two independent hash functions are used: modulo three and modulo ten. Initially all values are set to 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.

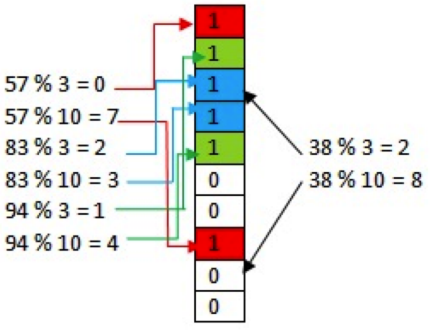


Fig. 1: Example: Bloom Filter

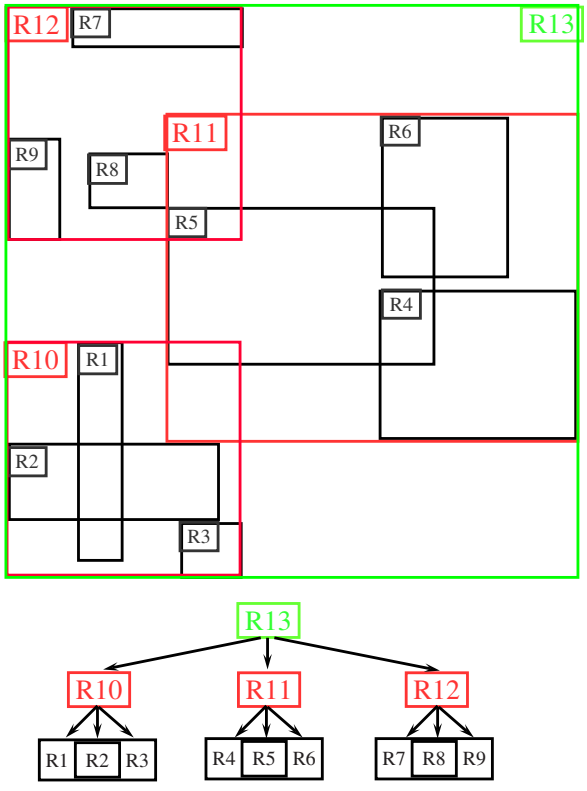


Fig. 2: Example: R-Tree

The second structure A-Tree is based on is R-Tree [11], [17]. R-Tree is a widely used data structure for multi-dimensional data. R-Tree is the extension of B-Tree to multi-dimensional data. Each node covers an area in the multi-dimensional space, usually represented as a hyper bounding box. In two-dimensional space the hyper bounding box is a rectangle; in three-dimensional space is a cuboid etc. A hyper bounding box is often represented by the pairs of minimum and maximum values for each dimension it covers. For R-Tree non-leaf nodes, this bounding box covers all of its children's bounding boxes. Lastly, leaf nodes also contain pointers to the actual data. The main disadvantage of R-Tree is that non-leaf nodes' bounding boxes overlap. Thus, in contrast to other traditional tree structures, searching requires a set of paths to be explored rather than following a specific path from root to the leaf that contains the data. This R-Tree specificity lead to a lot of varieties to be studied, including R^+-Tree [21] and R^*-tree [4], with most of them trying to minimize the overlap between nodes' coverage. Both R^+-Tree and R^*-tree approaches successfully minimizes this overlap. On the other hand, maintenance and insertion cost increases. Figure 2 demonstrates a simple example of an R-Tree in two-dimensional space that uses 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 [26].

A solution to provide query services over peer-to-peer networks is the Distributed Hash Table (DHT) and some variances [1]. Another relevant data structure is known as a scalable distributed B-tree [3]. With these data structures though, we are limited only to answering point queries and range queries over multi-dimensional data cannot be processed. Furthermore, as the Map-Reduce framework is widely used on cloud systems, a decent index structure was proposed [15]. 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 [19]. Other relevant indices include the RT-CAN [25], the BR-Tree [12] and the QT-Chord [10]. 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 excessive communication overhead, may exhibit weak even no consistency, and moreover they ignore the inherent partitioning of the cloud nodes as master and slaves. Therefore, they are not appropriate (native) for cloud computing environments. Google' Spanner [8], and the SQL-like distributed database – named F1 [22] – ontop of it, use traditional database indexes, tailored for key-value pairs.

The most relevant work to ours is the EEMINC index structure [27]. It exploits the separation of the cloud nodes as masters and slaves. All nodes with data, i.e., the slaves, make use of KD-tree as local indexes. Every slave node is described by a node cube, i.e., a set of ranges for every attribute 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 in order to avoid forwarding queries to irrelevant nodes, each node splits its own node cube to smaller ones. The divisions-node cubes that 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 that comprises the global index. Upon a request arrival at a master node, the local R-Tree is searched to find the slave nodes where potential results are stored. After accumulating a set of candidate nodes, the request is forwarded to all of them for further processing. Final results can be accumulated by a central node or sent directly to the requester.

Although EEMINC took a first step towards addressing the problem of indexing multi-dimensional data in clouds in a native way, it is inefficient since in the relative nodes locating phase, EEMINC chooses all the slave nodes in the cluster as the candidates of the query since it does not have the information about the data distribution on each slave node, and thus chooses all the possible slave nodes as the candidate set. This selection results in a high number of false positives, especially for point queries, which in turn implies larger latencies during query processing and increases the amount of waisted resources such as CPU processing time and bandwidth.

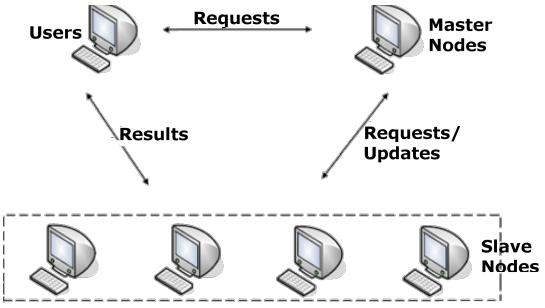


Fig. 3: The framework for query processing

3 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. Although this selection contradicts with the characteristic of cloud systems that central servers are not needed, almost all the available cloud platforms require the existence of central servers, usually referred as controllers. Controllers are responsible to hold the platform metadata and implement the interfaces to interact with the users. This is a typical approach in the clouds, e.g., in Map-Reduce frameworks, in infrastructure and platform as a service (IaaS and PaaS) approaches such as OpenStack and Google AppScale; 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 any of the master nodes that implement the user interaction interface. This node will process the request and forward it to the appropriate nodes on the platform for further processing. 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 and its execution results in an empty set, 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 less latencies and energy consumption as unnecessary calculations are avoided.

In this framework (presented in 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. In this way further resource consumption on

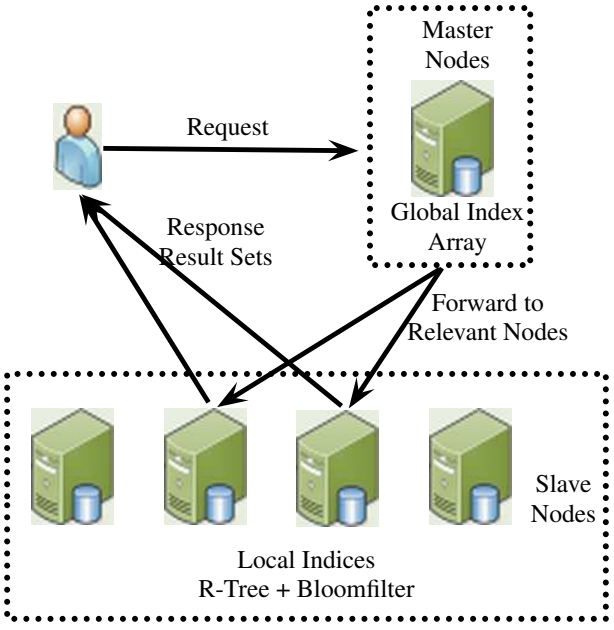


Fig. 4: A-Tree framework and data flow for query processing

master nodes, which in a large-scale system have heavy load by accepting and assigning queries to slave nodes, is minimized.

Every slave node is responsible to share any data updates with all master nodes that 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 made partial or full updates will be exchanged. In case of deletions and insertions, it is mandatory that master nodes are informed in order to keep the global index updated.

Due to the distributed and replicated nature of our method, we detect inconsistencies using Merkle trees [18], similar to other distributed replicated services, such as Amazon's Dynamo. Another important aspect, which is beyond the scope of this paper, is load balancing. Load balancing must be also given attention, and there are already some proposals such as PASSION [16], 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 4. 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. Our global index can be updated fast and respond to the demands of modern cloud indices where virtual machines (VMs) can be dynamically added or removed on demand in order to boost utilization and cost. Furthermore, clouds provide infinite resources, both in memory and cpu, and as shown in experiments section, adding more servers-vms (slaves and/or masters) in more cases (except deletions) results in lower or the same latencies.

4 Operations on Slave Nodes

4.1 Local Operations on Slave Nodes

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 R-Trees use hyper 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.

Every slave node is capable to process point and range query, insertion and deletion requests. Upon receiving a request the R-Tree operations and algorithms for searching, inserting and deleting data are those that are involved. Any of these operations may trigger the need to inform the master nodes about the changes. The information sent to master nodes are referred as updates and the process for constructing them is described in the next section.

For every insertion on the data set, we moreover add the record to the Bloom filter of the node. If necessary an update is sent to master nodes.

Point and range queries are processed locally and the result set are published directly to the user. 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 since Bloom filters are used the processing cost is less. If a point does not belong to 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 queries.

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. The deletion handling process firstly deletes the appropriate record from the tree and secondly increases a counter of deleted records by one. When counter reaches a threshold we update the node Bloom filter and send a new update to master nodes. It is recommended that the threshold is set dynamically according to workload on the system. The intuition is that the higher the counter is the higher the probability of a false positive occurrence. Thus, depending on the workload of the system, an update maybe sent in order to avoid a small or a high number of false positives.

Algorithm 1 Benefit for indexing children of node n

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

4.2 Distributing R-Tree Nodes

In this section we describe our algorithm for constructing and publishing the aforementioned updates, that involves 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. Based on the information provided by the updates, master nodes will select the appropriate slaves for query processing. As a hyper bounding box describes every R-Tree node, 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 space covered by 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. We define the benefit for indexing a node as the volume of the space that we will avoid searching if we index its children. That is its volume minus its children volume plus their overlap. Thus, the benefit for indexing the children of a node n is given by the following equation:

$$Benefit(n) = Volume(n) - (Volume(children) - Overlap(children)) \quad (1)$$

Now that we are able to determine the expected benefit of distributing a node, the recursive Algorithm 2 is proposed for selecting the R-Tree nodes that will be distributed. Algorithm 2 takes three parameters: (a) an R-Tree node; (b) the benefit threshold; and (c) the number of nodes that can be indexed (remaining space)

Algorithm 2 Select indexing nodes

```
1: function SelectIndexNodes (Node n, SET<BoundingBox> indexedNodes, int remainingSpace, double minBenefit)
2:     if getBenefitForIndexingChildrens(n) > minBenefit and !n.hasLeafChild() and n.#childrens ≤ remainingSpace then
3:         currentSize = indexedNodes.size();
4:         more_free_slots = 0;
5:         slotsPerChild = remainingSpace / n.#children;
6:         for i = 0 → n.#children do
7:             SelectIndexNodes(n.child[i], indexNodes, slotsPerChild, minBenefit);
8:             more_free_slots = indexedNodes.size() − currentSize;
9:             perChild+ = more_free_slots / n.#children−i;
10:            currentSize = indexedNodes.size();
11:        end for
12:    else
13:        indexNodes.add(n.boundingBox);
14:    end if
15: end function
```

and returns a set with the selected nodes. Algorithm 2 is based on the idea that available space is equally split to the nodes children. Since a recursive call may select fewer nodes than the remaining space, remaining space is calculated prior every call. In that sense, initial remaining space is an upper bound of the final update size. In real cloud computing systems, an R-Tree may be configured with a high fan out, e.g. 100. We face a trade-off for the number of nodes that will be indexed, as more nodes will result to higher resource consumption on master nodes for finding relevant nodes to a query, while on the other hand false positives and resource consumption on slave nodes are further minimized. Specifically, if we index only the root of the R-Tree, we have a large number of false positives, and if we index all tree nodes above leaves, we have a high processing cost for locating relevant nodes. As this is mostly depended on the system, we use as parameters the maximum number of hyper bounding boxes and the minimum expected benefit for indexing children.

It is also noted that even if we set the update space high and the benefit low leaf nodes are not indexed. We will not index leaf nodes because such an indexing will end up with all the data in the global index. Algorithm 2, in the worst case, has time complexity $O(|V|)$, 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. The time complexity for publishing an update is $O(|V|) + \Theta(n)$, where n is the number of master nodes running on the system.

Algorithm 3 Point Deletions

```
1: function DELETE (Point p)
2:     local_rtree.delete(p);
3:     if ++deleted_records > thresshold then
4:         deleted_records = 0;
5:         rebuilt local bloom filter;
6:         Invoke Algorithm 2;
7:     end if
8: end function
```

Algorithm 4 Point Insertions

```
1: function INSERT (Point p)
2:     current_state = get_state (p);
3:     local_rtree.insert(p);
4:     bf_change = local_bloom_filter.insert(p);
5:     if current_state == (b) then
6:         publish(local bloom filter);
7:     end if
8:     if current_state == (c) || current_state == (d)
   then
9:         extra_bb.add(p);
10:        if extra_bb.volume > threshold then
11:            Delete extra_bb;
12:            Invoke Algorithm 2;
13:            return;
14:        end if
15:        if bf_change then
16:            publish(local bloom filter);
17:        end if
18:        publish(extra_bb);
19:    end if
20: end function
```

4.3 Partial Updates

The aforementioned two algorithms describe the process of selecting the R-Tree nodes for distribution. Upon insertion or deletion, a change on a local index (R-Tree) occurs. Any change must be taken into consideration if it is going to lead to inconsistency in the global index. Hence, a new update should be constructed and published. Instead of constructing new updates to deal with these changes we propose the use of partial updates. A partial update is an update that is necessary to be published in order to keep the global index updated while it uses less network traffic and resource consumption on slave nodes.

As mentioned earlier, upon a deletion request a counter of deleted records is increased by one. Based on the counter threshold nodes' Bloom filter is rebuilt and Algorithm 2 is invoked to send a new update. Algorithm 3 presents the pseudocode for deletion requests.

In case of an insertion the possible situations are: (a) newly inserted record is completely covered by the last sent update; (b) newly inserted record is covered by the bounding box sent with the last update and is not covered by the Bloom filter; (c) newly inserted record is not covered by the bounding box sent with the last update and is covered by the Bloom filter; and (d) newly inserted record is not covered at all. For each of these four cases different type of partial update is sent to the master nodes. For the first case, no update is required. For the second case the node sends only the updated Bloom filter to master nodes. For the rest two cases, a node keeps an extra hyper bounding box. Upon insertion, and since the inserted record is not covered 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 as the system is running may get very big we set a threshold. If the threshold is reached then the extra bounding box is ignored and Algorithm 2 is invoked in order to calculate and send a fresh new update from scratch. Algorithm 4 presents the steps of the insertion process.

5 Operations on Master Nodes

5.1 Construction of Global Index

This section describes how master nodes handle the updates in order to build the global index and how the relevant nodes are located. The approach used here is to keep it as simple as possible in order to obtain fast processing, low complexity as well as low storage space.

Every master node maintains a copy of the global index. The global index is represented as an array that stores the updates received by slave nodes. Each position in 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 if the updates were stored in a tree structure. The array approach is also extremely useful and fast when processing a newly arrived update.

Upon the arrival of an update u from node with id n_id the only required action is to store it at the appropriate position: $global\_index[n\_id] = u$. Similar actions are required for a partial update with the difference that only the extra bounding box is changed and maybe the Bloom filter too. Hence, at every position of the array we store a set of hyper bounding boxes and the id and the Bloom filter of the slave node. It is noted that this is lowest possible cost for handling updates since the processing of a new update has time complexity $O(1)$.

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 [13].

5.2 Processing of Point and Range Queries

According to the described framework, processing a request at a master node is equivalent to locating

Algorithm 5 Find relevant nodes for a point query

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

Algorithm 6 Find relevant nodes for a range query

```
1: function SET GetRelativeNodesRangeQuery (Query rq)
2:     SET nodes = ∅;
3:     for all UPDATE u in global Index do
4:         for all Bounding Box box in u do
5:             if box.overlaps(rq) then
6:                 nodes.add(u.node_id);
7:                 continue with next Update;
8:             end if
9:         end for
10:    end for
11:    return nodes;
12: end function
```

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 processes for finding relevant nodes for a point query or a range query are not identical. Using Bloom filters, included in each update, the process of finding relevant nodes can be performed with complexity $\Theta(n)$ where n is the number of updates in the global index and equals to the number of slave nodes in the system. For range queries, Bloom filters cannot be used and the bounding boxes stored in the global index must be searched. It is noted that if a value for a dimension is not specified then the lowest and the highest values of the data in the dataset are used. In the worst case the complexity is $\Theta(n \cdot B)$ and in the best case $\Theta(n)$, where B is the number of bounding boxes in a stored update. Worst case occurs when the master node has to search all the bounding boxes for all the nodes. On the other hand, the best case is when the query is covered by the first bounding box in the update of every node. The upper bound can be lowered to $\Theta(n \cdot \log(B))$ if an appropriate index structure, such as R-Tree, for the bounding boxes of the updates is used. The described approach is preferred over it since such a selection will increase the cost of updates processing.

Algorithm 5 and Algorithm 6 show the processes for finding the set of relevant nodes to a point query and a range query, respectively. Algorithm 5 searches every update u in the global index if its Bloom filters covers the requested point p. If the Bloomfilter covers p then the node that has published the update u is added to the set of relevant nodes. Similarly, for range queries (Algorithm 6) a node is added to relevant nodes set if any of the bounding boxes *box* overlaps with the requested range query rq.

5.3 Insertion and Deletion of Records

To process an insertion or deletion request the same procedure is followed. Master node that received the request must first find a set of slave nodes to forward the request. In order to avoid triggering new updates on insertions and deletions Algorithm 5 for finding relevant nodes to the point 'query' is used.

As soon as the set of candidate nodes is built, the query is forwarded to them. If it is an insertion query, because selected nodes using Algorithm 5 already cover the new record a new update will not be triggered. In case the number of relevant nodes is less than the replication factor of the platform, 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. In the scenario of deletion request the relevant slave nodes process the deletion according to the aforementioned process in Section 4. If any of the relevant slave nodes send an update then it is handled by master nodes as described previously with $O(1)$ time cost.

6 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 [23], an open source and widely used simulation tool for grid networks, was used to simulate a data intensive grid environment and evaluate the performance of A-Tree. Simulation allowed us to measure point and range queries latency taking into consideration network delay according to links capacity, the size of the dataset and the number of nodes in the system as well as the utilization of both master and slave resources.

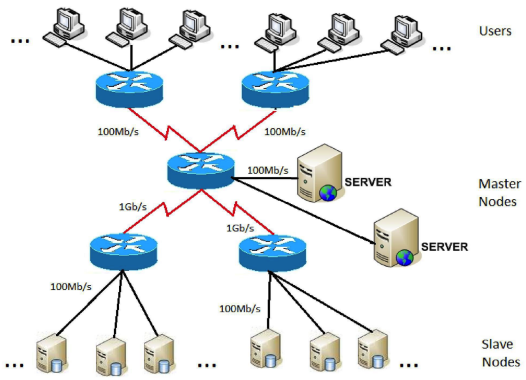
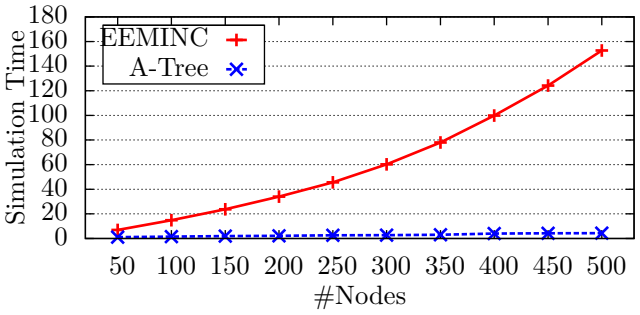
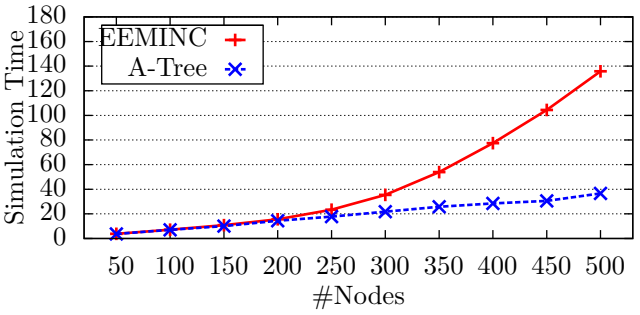


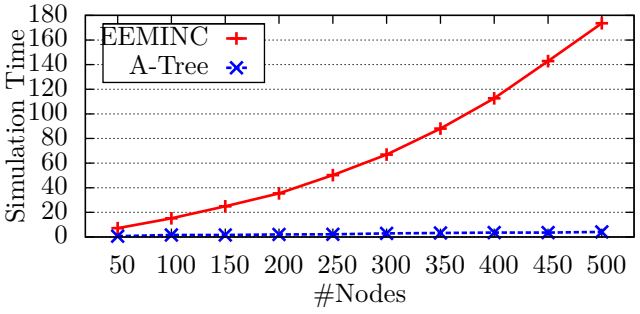
Fig. 5: Network topology used for the experiments



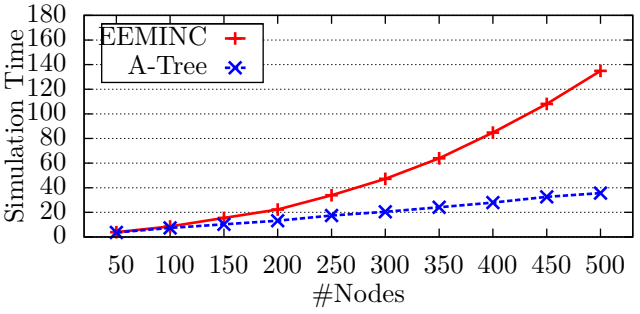
(a) Average Point Queries Latency with Five master nodes



(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

Fig. 6: Average point and range queries latency to the number of nodes

In all experiments, the EEMINC's uniform cutting for node cubes was used and uniformly distributed data. Used data set resides at 6-dimensional space unless explicitly stated otherwise. R-Tree fan out was set to fifty and for A-Tree Bloom filter 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. Every master node was connected to the master servers' router with a link of 100Mbps. This router had links to two other routers where all slave nodes were connected with 100Mbps speed. The speed of the links between the routers was initially 1Gbps and altered later to measure the effect of network speed on performance. Users were connected to two other routers with links to the master nodes' router with speed 100Mbps. The bandwidth of each user to the router is connected is 100Mbps. MTU was set to 1500 bytes for all links. Figure 5 shows the network topology. A user sends requests to a specific master node, defined at initialization state, and receives results from any, usually more than one, slave node. After the result for the previous request returns then another request is sent until the predefined number of requests (i.e. 100 point queries) for the experiment is reached. In all experiments requests number was set to 50. 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 executed on a computer system equipped with four Quad-Core AMD Opteron(tm) Processor 8350, 16 GB RAM and 320 GB storage capacity. Operating system was SUSE Linux Enterprise Server 10 (x86_64) running kernel version 2.6.16.21. All experiments were executed ten times and the average of the results is used for plotting the charts.

TABLE 1: 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

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 results in significantly faster query processing if data set does not contain the requested points. 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. User requests are uniformly distributed to master nodes; uniform distribution of requests (incoming traffic) realistically represents a real world cloud system that includes a simple and easy to be configured front end implementing a load balancer. Each slave node is responsible for ten

thousand records meaning that the size of the collection also 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 6a and Figure 6b 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 6c and Figure 6d. Firstly, we notice that, similarly to the previous experiment, latency is lower for A-Tree in contrast to EEMINC where latency is more when more master nodes exist in the system. Secondly, we notice that for small number of slave nodes with ten master nodes, the latency for range queries is almost the same with EEMINC but as the system gets larger A-Tree responds significantly faster. It is noticeable that latency increases for EEMINC for both point and range queries while A-Tree performance remains the same. For example when the system is made of 350 nodes EMMINC point queries latency has been increased from 77 to 88 and range queries latency from 54 to 64.

Concerning point queries, EEMINC performs worst when more nodes exist in the system, in contrast to A-Tree where little performance is gained. Table 1 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 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 even with more master nodes in contrast to EEMINC where exponential behavior was recorded.*

Another very important aspect for a global index performance 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, in this setup 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, each users makes fifty point queries and fifty range queries, the number of slave nodes was set to fifty, data dimension was set to two and data size was set from fifteen thousand records to seventy five thousands records per slave node, resulting to total 3750000 records.

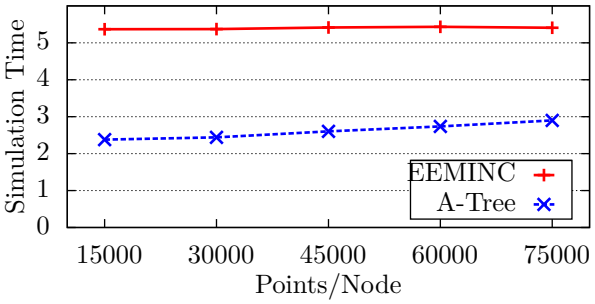


Fig. 7: Average queries latency as data set size increases. Total 50 slave nodes.

TABLE 2: Average False Positives as Data Set Size Increase with Ten Masters

#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

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 30.5Kb, is getting full and false positives probability is getting higher. The average latency for both range and point queries is shown in Figure 7. A-Tree is still capable of answering both point and range queries efficiently and scales as the system and data set are getting larger. Table 2 presents the number of false positives for both point and range queries. It is obvious that there is a great difference in the number of false positives for point queries. Despite the fact that false positives rates, for both point and range queries, are lower in A-Tree than in EEMINC, this small increment for point queries along with the fact that data set is larger on every nodes are the extenuation for the small increment of latency. Using bigger Bloom filters and setting the maximum number of bounding boxes in updates higher can easily avoid this small increase of latency.

In addition, we evaluate A-Tree for insertions and deletions. We use the same configuration as the previous experiment and setting data dimension to six. Figure 8 shows in log scale the insertion latency as the system is getting larger. It is clear that A-Tree also performs significantly faster for insertions. This is somehow expected as the cost of insertions is as low as possible.

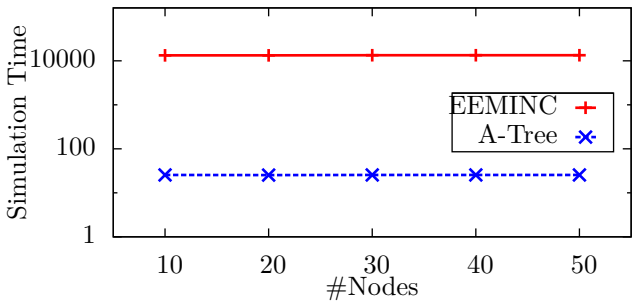


Fig. 8: Insertion Latency vs. #nodes

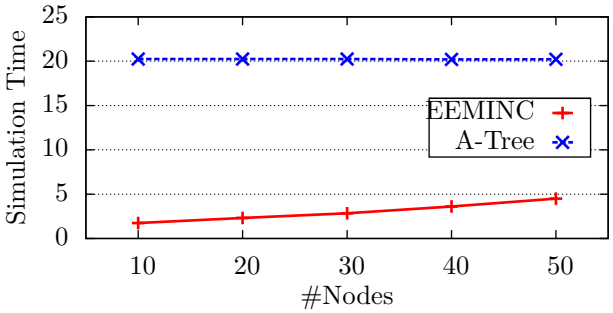
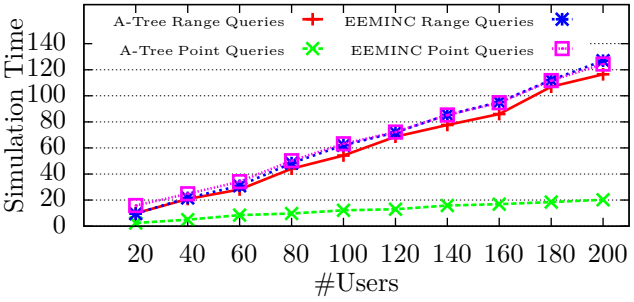
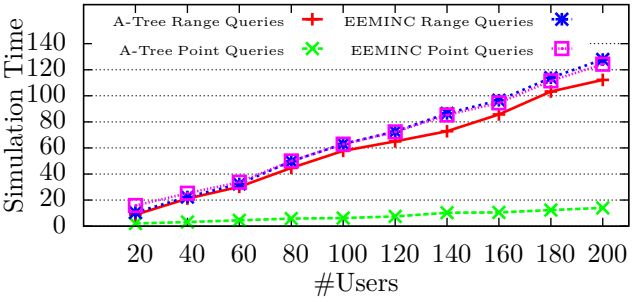


Fig. 9: Deletion Latency vs. #nodes

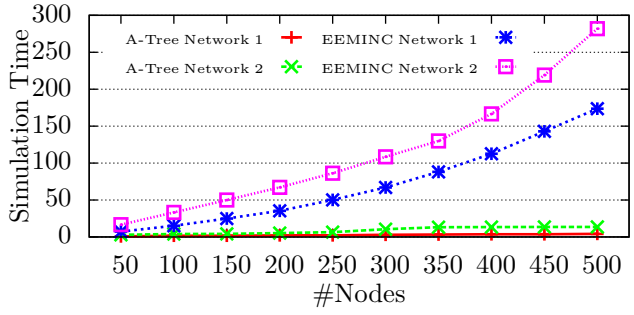


(a) Average queries latency as number of users increases with Five master nodes

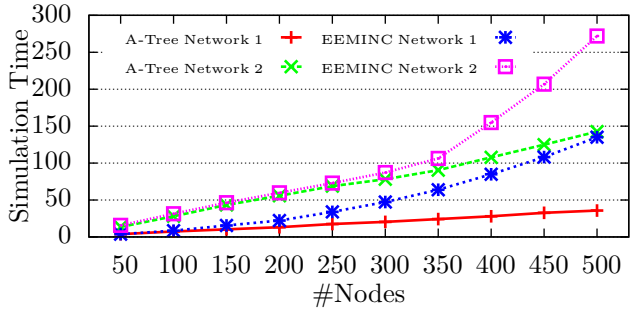


(b) Average queries latency as number of users increases with Ten master nodes

Fig. 10: Average queries latency as number of users increases



(a) Average latency for point queries



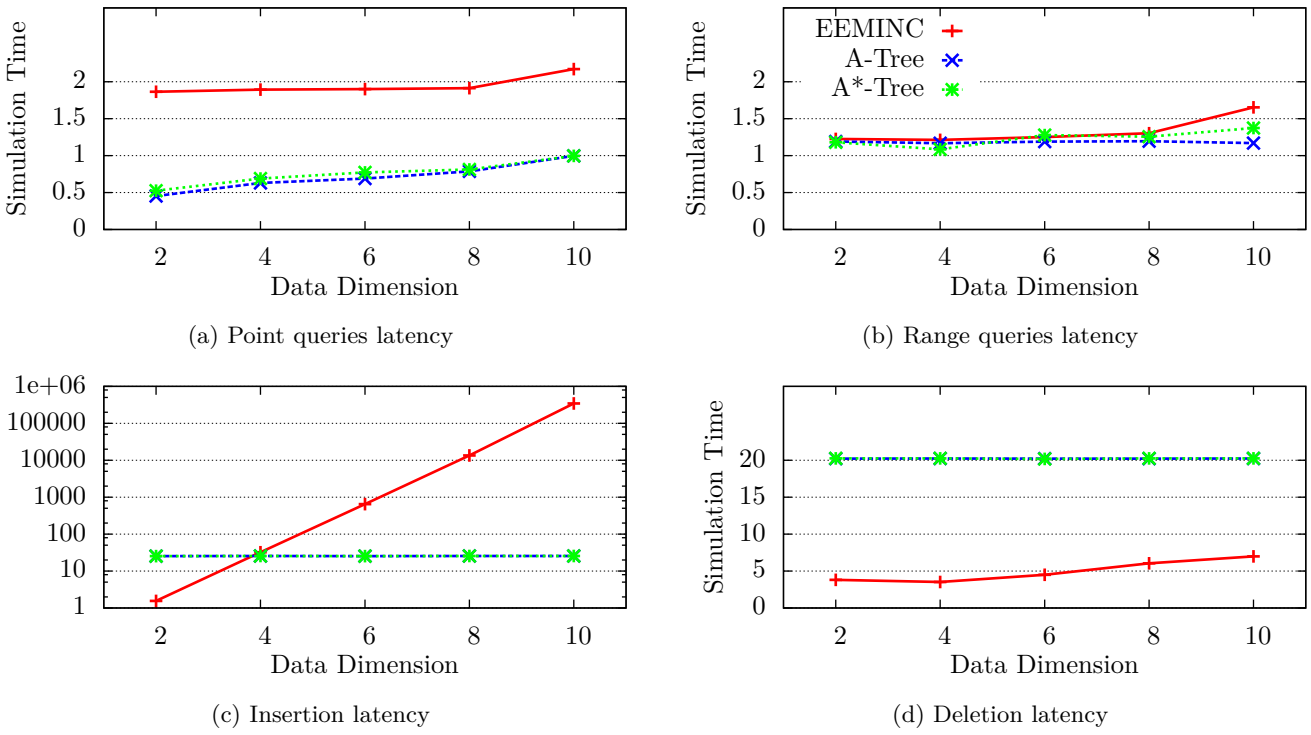
(b) Average latency for range queries

Fig. 11: Average queries latency for different network speeds with ten master nodes.

The insertion cost is low due to the process of selecting relevant nodes and the partial updates process. On the other hand, as shown in Figure 9, A-Tree results in slightly higher latencies for deletions. This slight increase is due to the higher cost of deleting data from an R-Tree than a KD-Tree.

Comparing the results from previous experiments, we notice that we have better performance when fewer nodes are used with more records on each of them. This behavior is somehow expected because when more nodes with less record exist, the whole system is not fully utilized. However, if nodes are overloaded operations will take longer and result in higher latencies. 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.

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 10a and Figure 10b show the results of this experiment for five and ten master nodes respectively. According to this experiment, the



(a) Point queries latency

(b) Range queries latency

(c) Insertion latency

(d) Deletion latency

Fig. 13: Queries latency vs. dimension of data

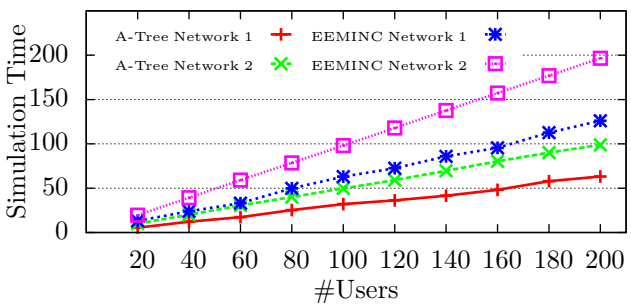


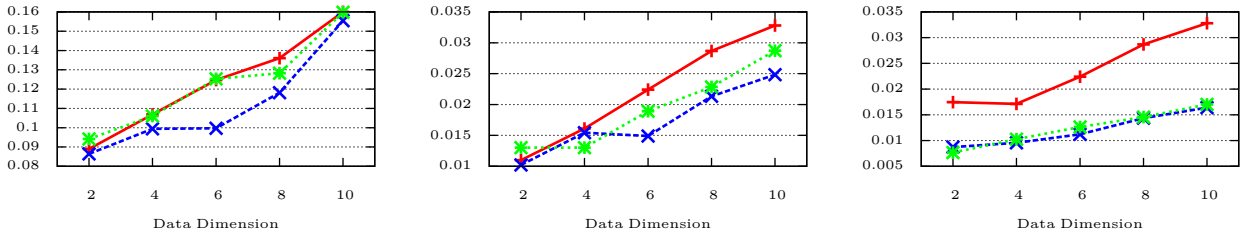
Fig. 12: Average queries latency as users increase for different networks

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. Concerning range queries A-Tree performs slightly faster. It is also noticeable that the number of master nodes slightly affects the queries latency, driving us to the conclusion 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.

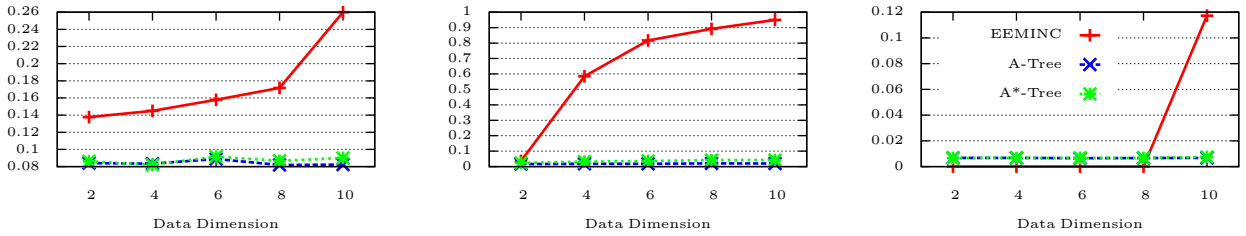
Another very important aspect is the capacity and speed of the network. An efficient index should not be affected very much from changes to networks connections and links speed. The purpose of next experiment is to evaluate how the links speed affects latency. The same network topology shown in Figure 5 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 1Gbps to 100Mbps. The reason for selecting low bandwidth is because for faster connections latency would be much less affected by network traffic. In this experiment ten master nodes exist. We refer to this configuration as Network 2.

Figure 11a shows the average latency for point queries for both network configurations. As expected latency is higher, because of limited network capacity for both approaches. A-Tree is not affected very much from this change since the use of Bloom filters avoids a lot of unnecessary network traffic. Based on this experiment, the use and inclusion of Bloom filters in the updates is proven to be more appropriate than the EEMINC cubes split approach. On the other hand, for range queries latency, shown in Figure 11b, A-Tree cannot take advantage of Bloom filters and is more affected by network change than EEMINC. A-Tree still performs better than EEMINC, especially for large systems.

Figure 12 shows the latency for point and range queries for both the above network configurations. Ten master nodes and one hundred slave nodes, each responsible for ten thousand records, exist in the system. Each user makes a hundred point queries and a hundred range queries. Average latency as the number of users in the system increases is shown in Figure 12 revealing that A-Tree is not affected very much from network



(a) Master Servers Utilization



(b) Slave Servers Utilization

Fig. 14: Resources Utilization for point and range Queries (left), Insertions (center) and Deletions (right) to the Data Dimension

changes as more users send requests simultaneously. This behavior is expected, as A-Tree keeps network usage as low as possible, especially for point queries, because it uses Bloom filters and keeps false positives rate very low.

In our next experiment, we evaluate the performance of A-Tree, A*-Tree and EEMINC to the dimension of data. A*-Tree is A-Tree using the same global index and update strategy but instead of R-Tree slave nodes use R*-Tree for local indices. As expected, dimension of data affects the queries latency since all the tree operations are proportional to the dimension of data they hold. In this experiment we use the network configuration presented in Figure 5 with 50 slave nodes and 5 master nodes. Figure 13a and Figure 13b show the average latency for point and range queries to the dimension of data. It is noticeable that A-Tree and A*-Tree outperform EEMINC for point queries while all scale linearly to the dimension of the data. Concerning range queries, A-Tree not only performs better but also keeps latency low independently of the data dimension while there is a small increase for EEMINC. This is a crucial aspect to the success of the proposed structure. We also note that A*-Tree performs slightly worse than A-Tree for both point and range queries.

Lastly, in Figure 13c the insertion latency is presented. It is noted that simulation time axis is in log scale. A-Tree and A*-Tree keep the insertion latency to the dimension of data steady while A*-Tree performs airily worse because insertions in R*-Tree cost more than in R-Tree. Although for low dimensional data EEMINC is significantly faster (also presented in Figure 8), as data dimension is getting higher A-Tree results in very much lower latencies. It is also shown that A-Tree latency for insertions, as well as the latency for range queries, is not affected by the dimension of data. Figure 13d presents the deletions latency to the data dimension. In this case, EEMINC is faster. This is due to the increment of false positives as data are deleted, which causes unnecessary calculations to take place. Furthermore, deletions sometimes cause R-Tree to perform recursive merges that in the worst case end up to the root of the index. The most important conclusions from this experiment are: (a) A-Tree has approximately the same performance as A*-Tree; and (b) *A-Tree scales linearly as data dimension is getting higher and responds faster to point, range and insertion requests in contrast to EEMINC.*

We further extend our experiments to analyze the resources utilization for A-Tree, A*-Tree and EEMINC as it is another crucial aspect for a cloud index. The utilization factor, presented in Figure 14, is the average ratio of the time that resources are in use to the total time of the simulation. We have seen that A-Tree

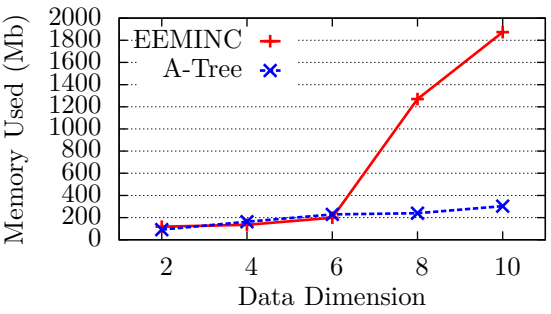


Fig. 15: Actual Memory Used for Simulations

requires less resources for both master nodes (Figure 14a) and slave nodes (Figure 14b) for queries, insertions and deletions while A*-Tree and EEMINC. A*-Tree requires more resources on master nodes because of the update strategy of A-Tree; since R*-Tree minimizes overlap the algorithm for building and distributing updates is often diving to just one level above leafs. Hence, updates are consisted by more bounding boxes than in A-Tree approach resulting in high master resources utilization. EEMINC also fails to keep master utilization low because the use of R-Tree as global index requires more resources than searching the A-Tree global index. On the other hand, A*-Tree requires little more resources on slave nodes for insertions because of the R*-Tree structure being used for local indices. Concerning deletions A-Tree and A*-Tree perform the same because of the partial updates strategy for deletions since unnecessary computations are avoided. EEMINC is by far the most resource hungry at both master and slave nodes.

Finally, Figure 15 presents the actual memory used during the simulations. The used memory can be considered as a good indicator of the actual memory needs. We notice the high memory consumption of EEMINC especially for data in high dimensional spaces. The memory consumption for A*-Tree is the same as A-Tree. Concluding the last two experiments *A-Tree is much more efficient concerning both memory and computational resources.*

7 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 multi-dimensional data sets in large datacenters. Based on the update strategy introduced, it is capable of fast processing of both point and range queries. Our experiments proved that A-Tree is an efficient, distributed data structure for multi-dimensional data stored in clouds, which scales linearly as the system grows larger, data volume is getting bigger and data dimension is getting higher. 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 and results in less energy consumption as a lot of unnecessary calculation as well as packages transmissions are avoided. Furthermore, our experimentation has shown that A-Tree not only minimizes network usage but also both the master and slave nodes utilization as well as memory usage.

We pose our future directions to the use of dynamic Bloom filters and smart data balancing algorithms. It is expected that these changes will lower false positives' rates for point and range as well as deletion queries, resulting in a trade-off between more essential space and preprocessing time and faster query processing.

Acknowledgments

The authors appreciate and thank the three anonymous reviewers for their valuable comments and suggestions, which have considerably contributed in improving the paper's content, organization and readability.

References

[1] "Distributed Hash Tables links," http://www.etse.urv.es/cpairot/dhts.html, Jul 2009.

[2] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt, "P-Grid: A self-organizing structured P2P system," *ACM SIGMOD Record*, vol. 32, no. 3, pp. 29–33, 2003.

[3] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed B-tree," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 598–609, 2008.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1990, pp. 322–331.

[5] S. Bibi, D. Katsaros, and P. Bozanis, "Business application acquisition: On-premise or SaaS-based solutions?" *IEEE Software magazine*, vol. 29, no. 3, pp. 86–93, 2012.

[6] 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.

[7] 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," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 4:1–4:26, 2008.

[8] J. C. Corbett et al, "Spanner: Google's globally-distributed database," in *Proceedings of USENIX Symposium on Operating Systems Desing and Implementation (OSDI)*, 2012, pp. 251–264.

[9] M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali, "Cloud computing: Distributed Internet computing for IT and scientific research," *IEEE Internet Computing magazine*, vol. 13, no. 5, pp. 10–13, 2009.

[10] L. Ding, B. Qiao, G. Wang, and C. Chen, "An efficient quad-tree based index structure for cloud data management," in *Proceedings of the International Conference on Web-age Information Management (WAIM)*, 2011, pp. 238–250.

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

[12] Y. Hua, B. Xiao, and J. Wang, "BR-tree: A scalable prototype for supporting multiple queries of multidimensional data," *IEEE Transactions on Computers*, vol. 58, no. 12, pp. 1585–1598, 2009.

[13] M. A. Islam and S. V. Vrbsky, "Tree-based consistency approach for cloud databases," in *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2010, pp. 401–404.

[14] H. V. Jagadish, B.-C. Ooi, and Q. H. Vu, "BATON: A balanced tree structure for peer-to-peer networks," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2005, pp. 661–672.

[15] I. Konstantinou, E. Angelou, D. Tsoumakos, and N. Koziris, "Distributed indexing of Web scale datasets for the cloud," in *Proceedings of the Workshop on Massive Data Analytics on the Cloud (MDAC)*, 2010.

[16] I. Konstantinou, D. Tsoumakos, and N. Koziris, "Fast and cost-effective online load-balancing in distributed range-queriable systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 8, pp. 1350–1364, 2011.

[17] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, *R-Trees: Theory and Applications*, ser. Advanced Information and Knowledge Processing. Springer-Verlag, 2006.

[18] R. Merkle, "A digital signature based on a conventional encryption function," in *Proceedings of the Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO)*, 1988, pp. 369–378.

[19] J. Qi, L. Qian, and Z. Luo, "Distributed structured database system HugeTable," in *Proceedings of the International Conference on Cloud Computing (CloudCom)*, 2009, pp. 338–346.

[20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," *SIGCOMM Computer Communications Review*, vol. 31, no. 4, pp. 161–172, 2001.

[21] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-tree: A dynamic index for multi-dimensional objects," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1987, pp. 507–518.

[22] J. Shutte et al, "F1: A distributed SQL database that scales," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1068–1079, 2013.

[23] 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.

[24] S. Tarkoma, C. Rothenberg, and E. Lagerspetz, "Theory and practice of Bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[25] J. Wang, S. Wu, H. Gao, J. Li, and B.-C. Ooi, "Indexing multi-dimensional data in a cloud system," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2010, pp. 591–602.

[26] Y. Wang, S. Wang, and D. Zhou, "Retrieving and indexing spatial data in the cloud computing environment," in *Proceedings of the International Conference on Cloud Computing (CloudCom)*, 2009, pp. 322–331.

[27] X. Zhang, J. Ai, Z. Wang, J. Lu, and X. Meng, "An efficient multi-dimensional index for cloud data management," in *Proceedings of the International Workshop on Cloud Data Management (CloudDB)*, 2009, pp. 17–24.