

Q-Tree: A Multi-Attribute Based Range Query Solution for Tele-Immersive Framework

Md Ahsan Arefin, Md Yusuf Sarwar Uddin, Indranil Gupta, Klara Nahrstedt

Department of Computer Science

University of Illinois at Urbana Champaign

Illinois, USA

{marefin2, mduddin2, indy, klara}@illinois.edu

Abstract

Users and administrators of large distributed systems are frequently in need of monitoring and management of its various components, data items and resources. Though there exist several distributed query and aggregation systems, the clustered structure of tele-immersive interactive frameworks and their time-sensitive nature and application requirements represent a new class of systems which poses different challenges on this distributed search. Multi-attribute composite range queries are one of the key features in this class. Queries are given in high level descriptions and then transformed into multi-attribute composite range queries. Designing such a query engine with minimum traffic overhead, low service latency, and with static and dynamic nature of large datasets, is a challenging task. In this paper, we propose a general multi-attribute based range query framework, Q-Tree, that provides efficient support for this class of systems. In order to serve efficient queries, Q-Tree builds a single topology-aware tree overlay by connecting the participating nodes in a bottom-up approach, and assigns range intervals on each node in a hierarchical manner. We show the relative strength of Q-Tree by analytically comparing it against P-Tree, P-Ring, Skip-Graph and Chord. With fine-grained load balancing and overlay maintenance, our simulations with PlanetLab traces show that our approach can answer complex queries within a fraction of a second.

1. Introduction

Tele-Immersion (TI) is a multi-site interactive system where sites (TI rooms) are geographically separated. A large number of correlated devices can be connected at each site and they are accessed via a single entry point (called a site-gateway). Some applications are 3D-collaborative dancing,

immersive video conferencing, immersive patient diagnosis, immersive physical therapy, immersive sports and games. With the increase in their scale in terms of number of devices and components connected at each site, it has become really hard to manage and monitor the whole TI system from a single administrative point.

Queries in such systems are not like the traditional database queries with a single key value, instead they are given in a high level description which are transformed into multi-attribute composite range queries. Some of the examples include “which site is highly congested?”, “which components are not working properly?” etc. To answer the first one, the query is transformed into a multi-attribute composite range query with constrains (range of values) on CPU utilization, memory overhead, stream rate, bandwidth utilization, delay and packet loss rate. The later one can be answered by constructing a multi-attribute range query with constrains on static and dynamic characteristics of those components. Queries can also be made by defining different multi-attribute ranges explicitly. Another mentionable property of such systems is that the number of site is limited due to limited display space and limited interactions, but the number of data items to store and manage can be extensively large due to large number of components in a single site to improve the immersive effect and thus shows scalability bottleneck. Data traffics are real-time large multimedia traffics and consume significant part of the communication bandwidth. So, the query plane should be light-weight, scalable in terms of data items and capable of answering queries in low latency.

We propose Q-Tree, a multi-attribute range based query solution considering all these requirements. One of the significant properties of our approach is that it injects only a single query to the overlay for any size of composite

multi-attribute queries without any preprocessing and still ensures the optimal number of node traversal. It can handle significant amount of attribute churn in the TI system and also scales with the number of data items.

While there have been several promising P2P range index structures that have been proposed in the literature, they have certain limitations for TI setup. Skip Graphs [1] and P-Trees [2] can only handle a single data item per peer, and hence, are not well-suited for large data items as found in tele-immersion. The index structure proposed by Gupta et al. [3] only provides approximate answer to range queries and load balancing, even when the P2P system is fully consistent. Our approach is a new P2P range index structure. It provides exact answers to range and aggregated queries (MAX, MIN, COUNT, AVG, SUM) by in-network aggregation and works for multiple data items using a single overlay structure.

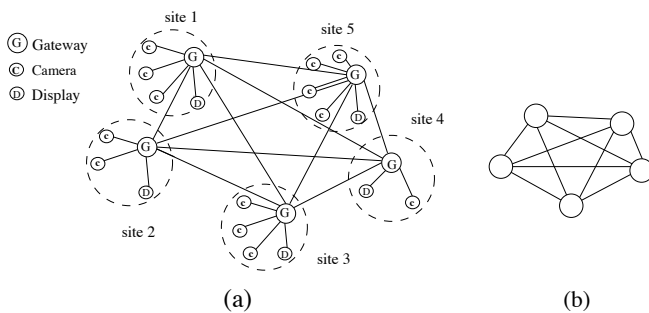


Figure 1. Tele-immersive environment. (a) Tele-immersive setup (b) Corresponding Overlay.

2. Related Work

The problem of designing a distributed query plane comprises of two aspects – an overlay construction containing all sites and a distributed search. Here, we review several systems to seek how they could be related to the attribute based range queries in tele-immersion. CAN [4], Chord [5], Kelips [6], Pastry [7] and Harren et al. [8] implement distributed hash structure to provide efficient lookup of a given key value. Since a hash function destroys the ordering in the key value space, these structures can not process range queries. P-Ring [9] supports both equality and range queries with logarithmic search performance using hierarchical hash. But it does not support aggregation, and also it is designed

for only single attribute range queries. Unlike Q-Tree, all these approaches build the management overlay depending on the node ids (IP address and Port number) in a *Top Down* approach as opposed to the *Bottom Up* approach that considers the end-to-end delay between sites.

There have been some other systems that are built to generate aggregated results on top of a large distributed overlay. Astrolabe [10] provides the abstraction of a single logical aggregation tree with administrative hierarchy for autonomy and isolation. Scalable distributed information management system (SDIMS) [11] is implemented based on the same approach and provides an aggregation abstraction on top of an overlay satisfying scalability, flexibility, autonomy, isolation, and robustness. SDIMS organizes nodes as an aggregation tree where leaf nodes are the physical machines of the system, and internal nodes, represented as virtual nodes, correspond to an administrative domain for information management. Astrolabe and SDIMS do not provide enough efficient provision for range queries.

There are also other monitoring and management tools developed for PlanetLab. CoMon [12] provides monitoring result both at a node level and a slice level. It is actually used to monitor the performance of nodes and to examine the resource profiles of individual experiments. Plush [13] is another application management tool for PlanetLab. It is designed to deploy and manage naturally distributed tasks. None of these support aggregation or range queries. MON [14] is an on-demand monitoring service for PlanetLab. To reduce the cost of maintenance, MON constructs a multicast tree on the fly to serve a query. MON is a better solution for multicast rather than range queries, because it has no prior knowledge about the attributes. Moara [15] is able to resolve queries efficiently by lowering response time and reducing message overhead. But its composite query plane needs to transform into canonical form to provide lower latency which eventually increases the overhead. Q-Tree allows any form of boolean expressions to query by propagating a single query message in the overlay.

3. Model and Architecture

Q-Tree intends to provide a multi-attribute based query solution for hierarchically clustered environments. In this section, we first define the system model properly. Then, we present the system architecture of Q-Tree along with its

metadata model.

3.1. System Model

We design our system model considering the TI interactive systems. There is a set of sites where multimedia and computing devices are connected to a gateway at each site. Gateway keeps information of local devices as well as other system attributes of the local nodes. Each Gateway can contact other gateways across sites, and so we represent each site by its gateway in the overlay¹, as shown in Figure 1. One practical example of TI application is TEEVE (Tele-immersive Environment for Everyone) [16]. It creates TI 3D multi-camera room environments both locally and remotely. These types of environment represent a next generation of TI where the ultimate goal is to give the broader audience a 3D tele-immersive experience over their available computing and communication infrastructure [17].

The system has very small churn and failure is fairly limited which can be instantaneously handled by the system administrator. Two types of data attributes are usually found in TI system: static and dynamic. Static attributes (such as static camera parameters) do remain fairly unchanged over an entire TI session, whereas dynamic data attributes (such as framerate, end-to-end delay, CPU utilization, bandwidth etc.) changes frequently. Any arbitrary site may initiate a query either providing a high level description of the query or explicitly defining any combination of (attribute, value/range) pair.

The underlying requirement of serving queries is to retrieve data items from nodes. Q-Tree organizes gateways in an overlay tree structure and assigns ranges to nodes in some hierarchy. Data items are disseminated into the overlay to be stored remotely in some other nodes according to the value. When a query is made for items specifying the range for certain attributes from any arbitrary node, a distributed search is initiated across the overlay. The primary objective of the query is to locate those nodes that store the requested data items. The tree structure with hierarchical ranges makes this query to be served efficiently. Data items with dynamic attributes higher than the bounded rate of update are handled via multicast. We explain the bound in Section 6.2.3.

1. From now on, we will use gateway and node interchangeably, though 'local node' represents the local device at each site.

Table 1. Comparative Analysis

Case	Chord	P-Tree	P-Ring	Skip Graph	Q-Tree
Overlay Choice	top-down	top-down	top-down	top-down	bottom-up
Performance	$O(\log N)$	$O(\log_d N)$	$O(\log_d N)$	$O(\log N)$	$O(\log_{k-1} N)$
Value per-node	multiple	single	multiple	single	multiple
Range Query (R)	no	yes	yes	yes	yes
Multi-Attribute R	PAO	PAO	PAO	PAO	yes

3.2. Basic Working of Q-Tree

Let us see a simple example to understand how Q-Tree works, as shown in Figure 2(a). Let us consider at first a single component such as a camera with a single attribute *frame rate* and let's assume, the possible value for frame rate in the system can be an integer between 1 and 20. As we see in figure, each node is assigned a range interval that specifies which items it stores (say, node *C* is given a range (6, 9], that means *C* stores information of cameras with frame rate within that range). Each node also knows the entire range of its subtree, such as node *C* knows that cameras with frame rate within (6, 20] are stored somewhere in its subtree. Now, a node, say *B*, has a camera with frame rate 15 in its local site and it tries to push this information as a data item into the tree. The data item then traverses the tree and reaches the node *E* where the item should be stored, as it contains the target range. Suppose, *G* makes a query for this item (*i.e.*, camera with frame rate 15). This time the query gets propagated to the node *E* in the same way and *E* returns the item information. In some cases, the node *E* needs to communicate to the original data source *B* to validate the staleness of the item before it delivers the result to the query initiator. We use the same strategy for answering the multi-attribute range queries. The detail of it is given in Section 6.2.

We analytically compare our system with P-Ring, P-Tree, Chord and Skip Graph, as shown in Table 1. Although they have the same $O(\log N)$ performance, none of them originally supports multi-attribute range queries and in-network aggregation suitable for TI structure. Furthermore, Q-Tree uses single overlay structure for all attributes as opposed to the *per attribute overlay* (PAO) found in other approaches mentioned above. As, the overlay maintenance cost is amortized over all attributes, our approach provides higher scalability in terms of number of attributes.

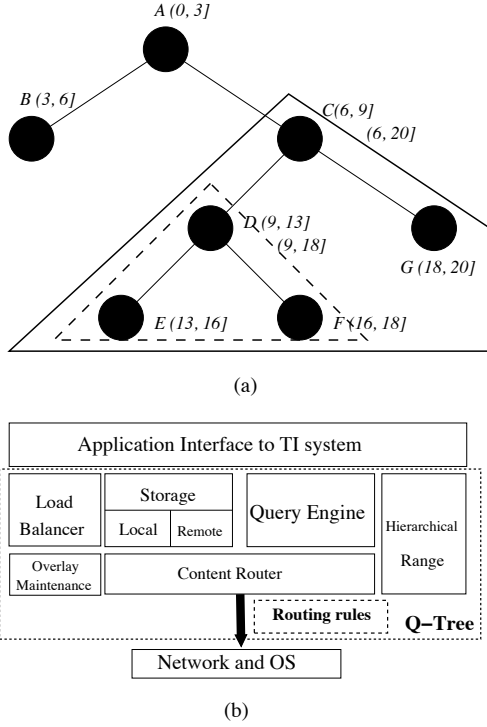


Figure 2. (a) Example (b) Q-Tree system architecture.

3.3. System Architecture

Q-Tree is a middleware service that sits in between the OS and application layer. All communications with the TI application are done through socket connection. We develop Q-Tree as a modular framework, shown in Figure 2(b). Functionalities of each modular components are given in the next section and a summary is given in Table 2.

Overlay Maintenance: The failure and churn of TI systems are fairly low. The Fault Tolerant Tree organizes the nodes in a tree structure. We build a latency optimal tree. Each node in the tree maintains a *nodebuddy* list, a list of all nodes whom it monitors for detecting failures via a periodic *heartbeat* message. We discuss detail about the node join, leave and failure in Section 7.

Metadata Store: The Metadata Store is responsible for storing the metadata in the local gateway node. It collects the metadata of items of the local devices and components on demand basis and stores it in the local storage to fit into the overlay. Also, it stores the metadata of remote items of its own range in the remote storage. One of the main contributions of this paper is to show how nodes can do multi-attribute queries efficiently, hence how to assign

hierarchical ranges to store remote metadata of data items on its own range to make the search efficient. In our system, the metadata is actually equivalent to the data item to be queried and so we use these two terms interchangeably.

Hierarchical Range: In many TI systems like TEEVE, all the participating sites join the system at the very beginning. That's why we consider the range assignment after the system reaches its stable state. This may not be true for other applications. Nodes can join incrementally in the tree and take some part of the ranges and corresponding data from their parents. Eventually the load balance algorithm organizes the ranges according to the load of the whole system.

Query Engine: As we explained earlier, queries are normally given in a high level description. It is the responsibility of the Query Engine to resolve the description into a single multi-attribute composite range query. One of the significant properties of Q-Tree is that it injects only a single query to the overlay for any size of a composite multi-attribute query without any preprocessing. Also, users can query with any form of boolean expressions without any transformation.

Content Router: The Content Router is responsible for efficiently routing the query messages and replying the result back in the aggregate manner. We support MAX, MIN, AVG, COUNT and SUM in-network aggregation. Another contribution of this paper is to show how the Content Router works in the range-assigned overlay for multi-attribute queries using a single overlay.

Load Balancer: We assign ranges to the nodes in the tree depending on the distribution of the attributes. But during the run time, the distribution can change and the load can be skewed. So, we use a Load Balancer similar to the Direct Neighbor Repeated [18] load-balancing approach. Each node periodically exchanges messages to its neighbor about its own load along with the heartbeat message and the load information is eventually propagated to the root. If the ratio of maximum load and minimum load in the system is greater than a threshold, the root initiates a load balancing algorithm where the highly loaded nodes transfer the load to its candidate range neighbors. We explain the load balancing technique more elaborately in Section 8.

Application Interface: Application interface works at the top of the Q-Tree system. It interacts with the user and TI application to pose different queries to the Query Engine.

Table 2. System Architecture

<i>System Component</i>	<i>Functionality</i>	<i>Reference</i>
Overlay Maintenance	Uses periodic heartbeat for handling failure and maintaining tree structure	Section 7
Hierarchical Range	Stores and maintains the hierarchical range at each node	Section 5
Metadata Storage	Stores remote and local metadata at the gateway according to the range	Section 3.4
Query Engine	Injects single and multi-attribute queries into the overlay	Section 6
Content Router	Performs efficient query routing and provide support for in-network aggregation	Section 6
Load Balancer	Performs load balancing similar to Direct Neighbor Repeated [18]	Section 8

3.4. Metadata Model

Sites store the metadata information regarding the devices they connect with or resources they maintain. Devices can be cameras, lights, local hosts, displays, sensors, remotes or any other devices. Resources may be LAN connections, delay and bandwidth across sites, packet loss rate, stream rate, CPU utilization, memory utilization, number of active cameras, number of active displays or so on. Information about each device, resource or site is expressed as a data item defined as a set of (attribute, value) pairs, e.g. a camera information at gateway ‘teeve1.cs.xyz.edu’ can be expressed as *Camera-node*:{(framerate,10), (shutter,120), (gain,133), (zoom-level, 12), (3D-reconstruction time, 10), (avg memory usage, 78%), (avg CPU utilization, 89%)} along with an *iteminfo*={(*node-info*=camera-node, *camera-id*=2, *host-gateway*=teeve1.cs.xyz.edu, *OS*=RHL, *Kernel version*=2.2.6)}. Here, *iteminfo* is not treated as an attribute, because a camera cannot be searched based on this part of information, whereas it can be searched by frame rate or gain or any arbitrary combinations of them. We define a data item d as tuples of attribute value pairs, $d = \{(a_1, v_1), (a_2, v_2), \dots, (a_l, v_l), I\}$, where $v_i =$ value of attribute a_i and I is a tuple of gateway *iteminfo* and the corresponding local node *itemInfo*. Although attribute values are usually continuous (sometimes, may be discrete) and can take wide range of values, we assume that all values can be normalized to (0.0, 1.0]. This can be easily done if the domain experts can somehow specify the minimum and maximum possible values for any attribute. Then, a simple equation $\bar{v} = \frac{v-v_{min}}{v_{max}-v_{min}}$ normalizes the value v to (0.0, 1.0].

4. Overlay Construction

Q-Tree uses a hierarchical organization of nodes in a tree-overlay, because tree provides in-network aggregation

of query results. As a default overlay, Q-Tree constructs a k -MST, a degree bounded minimum spanning tree (DBMST) where no node has degree greater than k . k -MST’s choice is due to the observation that in a tele-immersive system, sites may have resource constraints to serve arbitrarily many other sites. k -MST limits the number of sites connected to a single node to be at most k . Q-Tree’s query engine is, however, designed orthogonal to the overlay and it can be mounted on top of any tree. Of course, since queries will be traversing along the tree, we wish to design a locality aware and latency optimal tree.

Q-Tree’s k -MST is constructed centrally in the administrative gateway, called admin-gateway (pre-selected for a single TI session) of the TI system. The intuition behind this is that most of the TI applications like 3D collaborative dancing or immersive video conferencing require all sites to join at the very beginning of a session. Admin-gateway maintains each node’s joining and leaving, and keeps link latencies among the all pairs of sites. This can be done by asking nodes to report their measured latencies to other nodes periodically, or it can be computed from network coordinates of nodes. Knowing the entire topology, the admin-gateway computes the optimal overlay structure for the entire system and informs the structure to all nodes. This works well when the scale of the system is small and all nodes join at the beginning. Some applications may require large number of sites and belated joining of nodes. In that case, we construct the tree with the available nodes and allow other nodes to join using node joining algorithm presented in Section 7.

Finding a k -MST is an NP-hard problem for any given k [19]. For $k = 2$, it turns out to be an instance of Hamiltonian Path problem which is also a popular NP-hard problem. We devise a ‘neighborhood substitute’ based heuristic to construct an k -MST. At first, we compute an MST. In MST we identify the vertices that have more than

k incident edges, called *loaded* vertices, and the vertices that have degree less than k , called *candidate* vertices. Loaded vertices have to drop some of their edges, whereas the candidate vertices can add a few more. Then, we pick a loaded vertex, say x , and find an edge $e(x, y)$ such that the subtree rooted at y is the smallest sum of edge costs. This edge $e(x, y)$ is to be eliminated. Now, find a candidate vertex z which does not lie in the subtree rooted at y and for which weight $e(y, z)$ is minimum. Then, the subtree rooted at y under the loaded vertex x is moved to under the vertex z . The procedure is repeated until the loaded set becomes empty. In short, the concept is that we iteratively remove links from the node where the degree is greater than k and attach the removed nodes to the other nodes where the degree is less than k and ensure that there is no partition. The cost of substitution is determined by the link latencies of the moving subtrees.

Let, $ST_y^x(T)$ denotes the subtree rooted at y in the tree T that does not contain the edge $e(y, x)$, $w(T)$ denotes the sum of weights of all edges in T , i.e., $w(T) = \sum_{e \in T} w(e)$, and $deg_T(x)$ denotes the degree of vertex x in the tree T . k -MST construction is presented in Algorithm 1.

Algorithm 1 *Construct-k-MST()*

```

Find MST by Prim algorithm;
kmst ← MST;
Find loaded = {v | degkmst(v) > k}, and candidate =
{v | degkmst(v) < k};
while loaded ≠ ∅ do
    Pick x ∈ loaded;
    Find e(x, y) where y ∉ loaded and w(STyx(kmst)) is
    minimum;
    Find z ∈ candidate so that z ∉ SByx(kmst) and w(e(y, z))
    is minimum;
    kmst ← kmst - e(x, y) and kmst ← kmst + e(y, z);
    If (degkmst(x) ≤ k) loaded ← loaded - {x};
    If (degkmst(z) = k) candidate ← candidate - {z};
end while

```

5. Hierarchical Range

Once the overlay is constructed, nodes are assigned with ranges. A range r is denoted as $r(l, h]$, where l and h are respectively the lower and higher limit of the range and $0 \leq l < h \leq 1$. v , the value of attribute a , lies within r , i.e., $v \in r$, if and only if $l(r) < v \leq h(r)$. A range r_1 is contained within r_2 , hence $r_1 \prec r_2$, if all values in r_1 also

lie in r_2 , i.e., $l(r_2) \leq l(r_1) < h(r_1) \leq h(r_2)$. Two ranges r_1 and r_2 are said to be equal if they have the identical lower and higher limits, and are said to be consecutive if $h(r_1) = l(r_2) \vee h(r_2) = l(r_1)$. Two consecutive ranges can be ‘unioned’ to form a larger range like $r = r_1 \cup r_2$ and $r = (l(r_1), h(r_2)]$. We denote $|r| = h(r) - l(r)$ as the length of the range. Let T be the rooted tree where $p(x)$ and $C(x)$ are the parent node and the set of child nodes of node x . Each node x is assigned with two ranges, *self-range* $\delta^a(x)$ and *subtree-range* $\Delta^a(x)$ for each attribute a . Subtree-range $\Delta^a(x)$ specifies the range assigned to the entire subtree rooted at node x . An item d with an attribute-value pair (a, v) is stored at node x if $v \in \delta^a(x)$. If $v \in \Delta^a(x)$, then d is stored somewhere in the subtree rooted at node x . By definition, $\delta^a(x) \prec \Delta^a(x)$.

The following four properties are held by the ranges assigned to nodes for any attribute a :

- i) **Disjointness** $\delta^a(x) \cap \delta^a(y) = \emptyset$, for any pair of nodes x and y .
- ii) **Subtree range** $\delta^a(x) \prec \Delta^a(x)$ and $\Delta^a(x) = \delta^a(x) \cup_{y \in C(x)} \Delta^a(y)$.
- iii) **Hierarchy** If x is an ancestor of y , $\Delta^a(y) \prec \Delta^a(x)$.
- iv) **Entirety** $\Delta^a(\text{root}(T)) = (0.0, 1.0]$, and $\bigcup_{x \in T} \delta^a(x) = (0.0, 1.0]$, where $\text{root}(T)$ is the root node of T .

5.1. Range Assignment

Algorithm 2 shows how ranges are assigned to nodes. The assignment is initiated by the root by invoking *Assign-Range_{root}*(0.0, 1.0). Then, individual node assigns range to itself and to nodes in its subtrees, similar to a preorder traversal of the tree.

We assumed that all values within the entire range (0.0, 1.0] of an attribute are equally likely. So, each node gets a self-range of equal size $|\delta^a(x)| = \frac{1}{N}$ for each attribute. This ensures that each node stores nearly the same amount of data items. But if it happens that certain attribute follows some distribution other than uniform, the range should be partitioned depending on that distribution function (if known before-hand). For any given distribution function $F_a(v)$ for an attribute a , we need each node to get the equal number of data items to store, that is $P\{v \in \delta^a(x)\} = \frac{1}{N}$. Hence, $P\{l < v \leq h\} = F_a(h) - F_a(l) = \frac{1}{N}$, i.e., $h = F_a^{-1}(F_a(l) + \frac{1}{N})$, and for $\Delta_a(c)$, it would be

Algorithm 2 $AssignRange_x(a, l, h)$

Input:

x : Node to which assignment is being made;
 a : Attribute;
 l, h : real numbers, range bound,

```

 $\Delta^a(x) \leftarrow (l, h]$ ;
 $\delta^a(x) \leftarrow (l, l + \frac{1}{N}]$ ;
 $l \leftarrow l + \frac{1}{N}$ ;
for all  $c$  in  $C(x)$  do
   $n_c \leftarrow c.size$ ; /* size of the subtree at child  $c$  */
   $h \leftarrow l + \frac{n_c}{N}$ ;
   $AssignRange_c(a, l, h)$ ;
   $l \leftarrow h$ ;
end for

```

$h = F_a^{-1}(F_a(l) + \frac{n_c}{N})$. Figure 3 shows examples of two range allocations for two different distributions. The first range of each node shows its own range and the second range indicates the subtree range. For example, node E in Figure 3(b) has its own range $(0.68, 0.74]$ and its subtree range is $(0.68, 1.0]$.

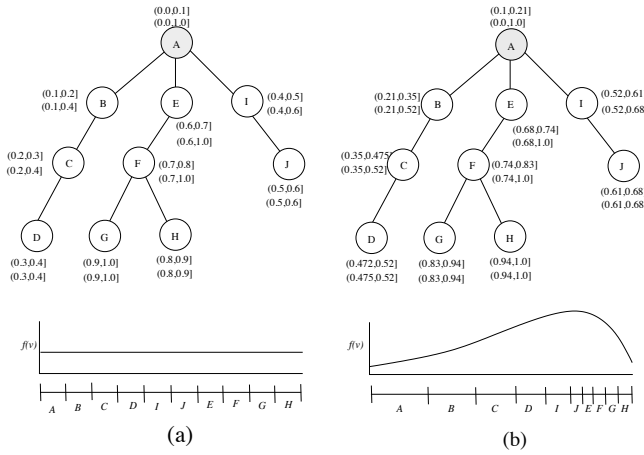


Figure 3. Range assignment (a) Uniform (b) A right skewed distribution.

If all attributes have identical distributions, then at a particular node x , self-ranges for all attributes become equal, *i.e.*, $\delta^{a_1}(x) = \delta^{a_2}(x) = \dots = \delta^{a_l}(x)$. This is also true for the subtree-ranges. In that case, nodes require to keep only a single self-range and subtree-range for all attributes instead of ranges per attribute. Therefore, each node keeps $|C| + 2$ ranges – self-range and subtree-ranges for parent and $|C|$ children. If all attributes are treated differently, it becomes $L \times (|C| + 2)$ ranges for a system of total L attributes. In

our current implementation, we, however, assume similar and uniform distribution for all attributes.

6. Q-Tree Query Engine

Once ranges are assigned for each attribute, the system becomes ready to fuse data items into the overlay and to serve queries. In the next two subsections, we present how data items are fused into the overlay and range queries are made.

6.1. Inserting Data Items

Data items originating in gateway nodes are fused in the overlay to be stored by remote nodes. Algorithm 3 shows the insertion operation. Whenever new data items are created or any attribute of an inserted item changes, they are fused in the overlay. An item d is stored at a node x if any of d 's attribute value lies within the self-range of node x for that attribute. If any attribute value lies within the child's subtree range, the item is forwarded to that child. The same happens for the parent node. In Figure 4(a), we show how the insertion work for data item $d = [(a, 0.73), (b, 0.59), I]$.

Algorithm 3 $Insert-item_x(d)$

x : The node at which insertion is made;
 d : Date item to be inserted;
 $\mathcal{R}(x)$: The set of items stored at x , inserted by others;

```

if (There exists  $(a_i, v_i) \in d$  such that  $v_i \in \delta^{a_i}(x)$ ) then
   $\mathcal{R}(x) \leftarrow \mathcal{R}(x) \cup d$ ;
end if
if  $(v_i \in \Delta^{a_i}(c)$  for any child  $c$ ) then
   $insert-item_c(d)$ ;
end if
if (there exists  $j, v_j \notin \Delta^{a_j}$ ) then
  /* Not all values lie in this subtree, forward to  $p(x)$  */
   $insert-item_{p(x)}(d)$ ;
end if

```

In Figure 4(a), lets assume that the ranges given in the query correspond to $framerate(= a)$ and $resolution(= b)$ attributes. Node D has a data item with $(framerate = 0.73, resolution = 0.45, \langle iteminfo \rangle)$ ². Now, D inserts the insertion request once into the overlay along with the associated values and *iteminfo*. This request moves along the

2. Attribute values are normalized to $[0, 1.0]$.

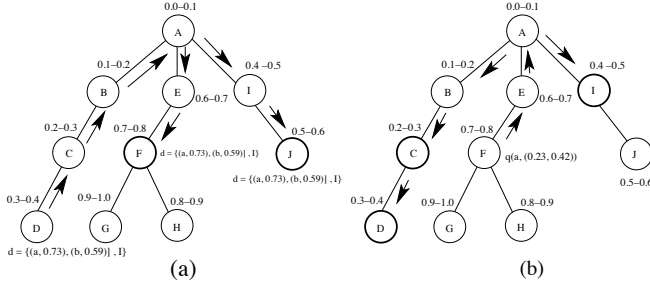


Figure 4. (a) D inserts $[(a, 0.73), (b, 0.59), I]$, the item is inserted at F and J (b) F initiates query $q(a, (0.23, 0.42))$, query results are returned by C, D and I .

path $D \rightarrow C \rightarrow B \rightarrow A$ to node A . Node A forwards the insertion request to node E (as $SubtreeRange_E = (0.6, 0.9]$ that satisfies the range of $framearte$ in the query) and node I (as $SubtreeRange_I = (0.4, 0.6]$ that satisfies the range of $resolution$ in the query). The final values are inserted at node F and J , where $SelfRange_F = [0.7, 0.8)$ and $SelfRange_J = [0.5, 0.6)$.

6.2. Query

Once data items are fused, the overlay is now ready to serve queries. Q-Tree is especially designed for range queries with single or multiple attributes. Any node can originate a query by passing the query statement to the Q-Tree query engine which initiates a distributed search to locate nodes where the requested data items are stored. Sometimes an aggregate function, say COUNT, on the result is also requested. In all cases, replied items are returned to the query source being aggregated in the intermediate nodes along the path of the tree. An example is given in Figure 4(b).

6.2.1. Range Query. Q-Tree looks for data items where the attribute value lies within the queried range. A range query with a single attribute is expressed as a predicate $q(a, r)$ that becomes true for a data item whose value v for attribute a is within r , i.e., $l(r) < v \leq h(r)$. In that case, we say d satisfies q , denoted as $d \stackrel{s}{\Rightarrow} q$. The query propagates via the tree to reach the nodes where the requested items are stored. A node x contains candidate items for $q(a, r)$ if $\delta^a(x) \cap r \neq \emptyset$. Similar check ($\Delta^a(c) \cap r \neq \emptyset$) can be made to see whether any of its subtree contains the result. A camera with

$framesize=0.45$, evaluates $q(framesize, (0.3, 0.5))$ to true, but one with $framesize=0.7$ does not. Figure 4(b) shows a query example where all values are normalized between 0 and 1.

A query can also be of type less-than ($<$) or greater-than ($>$). In that case, only one end of the range is given; another end of the range can be chosen as appropriate. The query for attributes $a \leq u$ (where $0 < u \leq 1$) is equivalent to $q(a, (0.0, u])$, where the query for $a > u$ is equivalent to $q(a, (u, 1.0])$. For equality (like, frame rate = 0.5) range becomes singular. This is handled by a special tag passed with the query statement. Note that, in the user-level, queries are given with real values in ranges, but Query Engine transforms those ranges in between 0 and 1.

One of the interesting aspects of Q-Tree is that it can handle complex range queries with multiple attributes as efficiently as it does for a single attribute range. For a complex query, Q-Tree propagates a single query into the overlay rather than propagating separate queries for each simple part. An example of a complex multi-attribute query can be ‘find cameras with (frame rate within (5, 20) OR frame size ≥ 1024) AND (shutter ≤ 60 OR gain > 123)’. Q-Tree handles this kind of complex queries by expressing the query as a composite query predicate. A composite query predicate \mathcal{P} is a boolean expression that contains a set of single attribute query predicates joined by boolean operators. For example, $\mathcal{P} = q_1(a_1, r_1) \wedge q_2(a_2, r_2) \vee \neg q_3(a_3, r_3)$. In general, composite query predicate is expressed as $\mathcal{P} = q_1 \diamond q_2 \diamond \dots \diamond q_m$, where \diamond is any arbitrary boolean operator like \wedge, \vee, \oplus , and each q_i can be either q or $\neg q$. At each node, decisions are taken by the following rules. The algorithm for this is shown in Algorithm 4.

$$d \stackrel{s}{\Rightarrow} \mathcal{P} \equiv (d \stackrel{s}{\Rightarrow} q_1 \diamond d \stackrel{s}{\Rightarrow} q_2 \diamond \dots \diamond d \stackrel{s}{\Rightarrow} q_m)$$

$$d \stackrel{s}{\Rightarrow} q(a, r) \equiv \exists_{(a,v) \in d} (v \in r)$$

A query (single or multi-attributed), initiated at any node, traverses the tree to reach those nodes where requested data items are stored. When a node receives a query along with the predicate, it checks whether any items in its data store satisfy the predicate. If it satisfies, it identifies those items to return as query results and forwards the query depending on the satisfiability of the neighboring range.

6.2.2. Aggregation Functions. Sometimes an aggregated result over the items is requested rather than the list of items

Algorithm 4 $Query_x(Predicate \mathcal{P})$

x : Query made at node x ;

$\mathcal{P} : q_1(a_1, r_1) \diamond q_2(a_2, r_2) \diamond \dots \diamond q_m(a_m, r_m)$;

$R \leftarrow \emptyset$;

$\Lambda = (\delta^{a_1}(x) \cap r_1 \neq \emptyset) \diamond (\delta^{a_2}(x) \cap r_1 \neq \emptyset) \diamond \dots \diamond (\delta^{a_m}(x) \cap r_m \neq \emptyset)$;

if ($\Lambda = \text{True}$) **then**

/* This node contains items that satisfy \mathcal{P} */

find item $d \in \mathcal{R}$ such that $d \stackrel{s}{\Rightarrow} \mathcal{P}$

$R \leftarrow R \cup d$;

end if

For each child c compute

$\Lambda_c = (\Delta^{a_1}(c) \cap r_1 \neq \emptyset) \diamond (\Delta^{a_2}(c) \cap r_2 \neq \emptyset) \dots \diamond (\Delta^{a_m}(c) \cap r_m \neq \emptyset)$

if ($\Lambda_c = \text{True}$, for a child c) **then**

$R \leftarrow R \cup Query_c(\mathcal{P})$;

end if

if (If for any a_i in \mathcal{P} , $\Delta^{a_i}(x) \cup r_i \neq \emptyset$) **then**

$R \leftarrow R \cup Query_{p(x)}(\mathcal{P})$;

end if

return R ;

itself. For example, ‘find MIN CPU utilization where memory $\leq 1\text{GB}$ ’, returns the minimum of all CPU utilizations of gateways that are equipped with less than 1GB memory. Q-Tree supports a set of aggregate functions, namely MIN, MAX, COUNT, SUM, and AVG. In that case, each node replies only a single value computed over the resultant items as defined by the requested aggregate function. Some aggregate functions cannot be applied on partial results, like DISTINCT COUNT or MEDIAN. In that case, entire result is accumulated before applying the aggregate function.

6.2.3. Handling Data Dynamics. Whenever a new metadata is inserted or attribute value of an inserted metadata is updated, they are simply fused in the overlay. Yet the old items are not deleted immediately. Instead they are deleted *lazily* whenever they are attempted to be returned in response to a query. When an item is inserted or updated, the remote site sets timestamp on the item along with the original source of the item. If the item becomes older than some *freshness* threshold at the time when it is returned against a query, the remote node contacts to the original source node to check whether any attribute value for this item has been changed. If nothing is changed, the item stays at the remote node and the timestamp is updated. Otherwise the item is deleted.

Some attributes, like CPU utilization or available band-

width change quite frequently invoking frequent insertions. If these items are not searched as frequently as they are changed, these insertions incur substantial communication cost. In that case, items would preferably reside at the original node, and a simple multicast seems more efficient to serve range queries. We analyze below the communication cost for both cases.

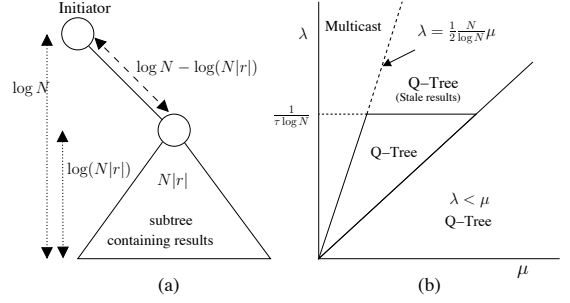


Figure 5. (a) Message cost for the query $q(a, r)$ (b) The operating zone of Q-Tree.

For a network of N nodes, we can verify that the message cost for multicast is $2N$. It simply follows from the fact that the query reaches to every node and returns back to the source. Let us find the message cost for the query $q(a, r)$. If the ranges are assigned uniformly over all nodes, then the number of nodes that contain items satisfying $q(a, r)$ is roughly $N|r|$. The query is pictured in Figure 5(a). The resulting items can be assumed to be confined in a subtree of size $N|r|$, which are returned by a multicast in that subtree. Yet the query has to reach that subtree to initiate the multicast inside. The multicast cost is $2N|r|$ and the cost to reach to that subtree is roughly $\log N - \log(N|r|)$. So, total message cost for the query is $cost(|r|) = 2(\log N - \log(N|r|)) + 2N|r| = 2(N|r| - \log|r|)$. We can check that $cost(1.0) \approx 2N$, (the cost of multicast in the entire tree), and $cost(\frac{1}{N}) = 2 \log N$, which is equal to the cost of query to a single node.

Let the value of attribute a change at rate λ and the range query on a is made at rate μ . Multicast cost is $2\mu N$, whereas in Q-Tree, the cost is the sum of insertion and query cost, that is $\lambda \log N + 2\mu(N|r| - \log|r|)$. Therefore, Q-Tree has smaller message cost than multicast as long as:

$$\lambda < 2 \times \frac{N(1 - |r|) + \log|r|}{\log N} \mu$$

The above inequality gives the maximum rate at which

attribute value can be changed when Q-Tree’s message cost remains smaller than multicast. On average $|r| = \frac{1}{2}$ gives, $\lambda = \frac{N}{\log N} \mu$. For $N = 100$ and $\mu = 1/\text{min}$, we get $\lambda = 50/\text{min}$. Again, if data changes faster than the time it takes to insert into the overlay, then the query may return stale data. This stale return can be avoided if $\frac{1}{\lambda} > \tau \log N$, i.e., $\lambda < \frac{1}{\tau \log N}$, where τ is the average link latency. For $\tau = 200\text{ms}$, we have $\lambda = 150/\text{min}$. With increasing μ , allowed λ can be increased, but it cannot be larger than $\frac{1}{\tau \log N}$. Beyond that, Q-Tree switches to multicast for those data items. We find an operating zone for Q-Tree and multicast as shown in Figure 5(b).

7. Overlay Maintenance and Failure Detection

When a new node joins, the admin-gateway handles the registration and attaches it to an existing node considering its locality. The attached node becomes its parent. Then the parent node halves its self-range and assigns the higher range to that node.

Each node sends periodic heartbeat message to all of its neighbors (children and parent) to detect failure. Each node keeps a *nodebuddy* list, a list of nodes whom the node monitors for failure. In Q-Tree, children of a node are the nodebuddies of the parent, and the root is monitored by one of its child. Node can voluntarily leave or fail. In either case, the parent node creates a process representing the child until the child node reappears in the system. The parent node temporarily appends the child range with its own and acts as the child node in storing items and forwarding queries. In case of voluntary departure, the child transfers all items it was storing before it leaves. In failure, parent fetches all items that child was storing by making a query over the child range. To make this happen, each node keeps states (self-range, children list) of its nodebuddy members.

8. Load Balancing

In Q-Tree, we can place all nodes in a continuous logical range ring. The predecessor and successor of a node along the ring are the nodes that have the subsequent ranges before and after the given node. We call them *range neighbors*. Note that this adjacency may not be related to the parent-child relation in the overlay. If a node wants to reduce its load, it needs to adjust its range with one of its range neighbors (predecessor or successor).

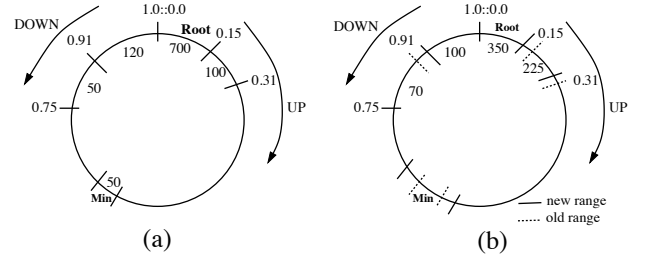


Figure 6. Load balancing for $\alpha = 2$. (a) Initial load (b) After one pass, load and ranges are adjusted.

The load balancer works in several passes. Every node periodically reports the *minload* and *maxload* of its subtree to its parent via the heartbeat messages. Thus the root has the information about *minload* and *maxload* of the whole system. If there is a load imbalance, that is $\text{maxload} > \alpha \times \text{minload}$, root initiates the load balancing algorithm in two directions via its successor (UP) and predecessor (DOWN). If any node in the path finds its $\text{load} > \alpha \times \text{minload}$, it makes $\text{load} = \max(\frac{\text{load}}{2}, \alpha \times \text{minload})$ and adjusts its range and data items accordingly. It adjusts its range and transfers the excessive data items to its successor (if its in UP direction) or predecessor. When the balancing reaches the *minload* node, it notifies the root and the current pass is done. After sometime, root starts another pass. We use, $\alpha = 2$ so that no nodes are allowed to store data items more than double of other nodes. Figure 6 shows an example of load balancing operation.

9. Simulation Results and Evaluation

TI systems are part of a new class of distributed systems including teleconferencing, multiplayer games and interactive multi-party systems. These systems do not scale much and most of them are bounded by 10/20 sites. Scale can be a slightly more (about 100/200 sites) specially for multiplayer games or large scale conferencing. So we restrict our simulation upto 250 nodes, though we believe Q-Tree is scalable for larger number of nodes.

We simulate Q-Tree in an emulated network setup on top of a discrete network event simulator coded in Java. We emulate a ‘virtual’ network with node-to-node latencies obtained from 4 hours PlanetLab traces [20] which consists of 250 distinct nodes. This trace gives us the connectivity information and *rtt* delays among the nodes. We use this

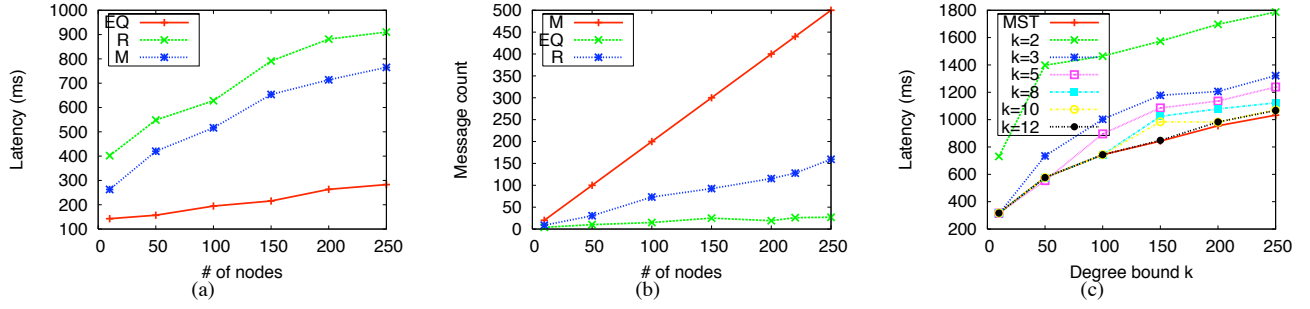


Figure 7. (a) Latency (MST), (b) Message count (MST), (c) Latency for different k with 100 nodes.

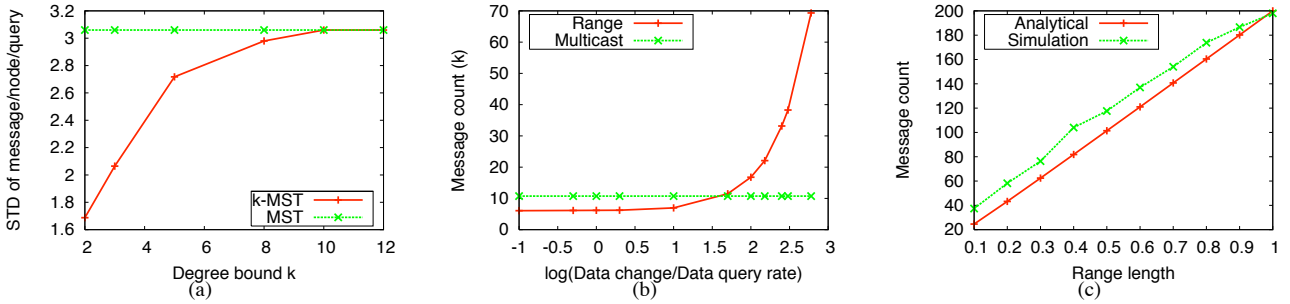


Figure 8. (a) Standard deviation of message per node per query for different k and 100 nodes, (b) Message overhead for different rates of update, (c) Message cost for different range intervals.

information to simulate query plane for TI systems. As an overlay choice, we consider MST and k -MST (degree bounded MST). The reason behind this is that we always want to build a latency-optimal tree, though our solution would work for any other tree construction approaches.

We are interested in mainly three performance metrics: query latency, communication cost, and overhead in metadata maintenance. We consider queries of three kinds, namely equal-to (EQ), multi-attribute composite range queries (R), and multi-attribute multicast queries (M). EQ represents query specified by an ‘*attr = value*’, for example ‘cameras with *framerate=20*’, R specifies limits on attribute values, such as ‘cameras with *framerate* between (10,15) and *gain > 100*’. Range queries (for both static and dynamic attributes) may also optionally be accompanied with any of the aggregate functions like MIN, MAX, COUNT, SUM and AVG, for example ‘what is the MAX *framerate* in current TI session?’. For each run, we build the overlay by taking nodes randomly from the traces and simulate the same event for 100 instances and take their average. Unless stated, each site has 10 devices and each device has 10 attributes.

Figure 7(a) shows the latency of different queries for different number of nodes in the system. Here the overlay

choice is MST. As we see in the plot, the average latency for ‘equal-to’ (EQ) queries is fairly small compared to the other queries in the system. This is reasonable because EQ only searches for a single node in a specific range. ‘Multi-attribute range’ (R) queries are more common in TI system and due to the hierarchical range assignment, those queries can be served within less than a second on the average even with 250 PlanetLab nodes. Another reason for getting lower latency is due to the ‘bottom-up’ approach in overlay construction which ensures a latency-optimal tree and thus further reduces the latency for multicast queries. In Figure 7(b), we show the message count of these three types of query. Message count includes the number of messages used to propagate the query and to reply back the result to the originating node. For multicast, the message count is always the twice the number of nodes in the system because of the aggregated reply. As we see in the figure, for EQ and R, the message count changes very slowly with the number of nodes.

In Figure 7(c), we present the impact of degree bound on overlay for range queries. We compare the latency of multi-attribute multicast queries with MAX function for MST, and several values of k . The result is reasonable: when $k = 2$,

each node is allowed to have only one child in the tree which makes the tree deep enough and thus increases the latency. With the increase of the k value the latency decreases (*i.e.*, the performance increases) and gives the maximum performance in case of MST. However, there is a tradeoff between the message overhead per node and the latency, for the choice of different k values. For MST, the standard deviation of message count per node per query is fairly high, because of wide variation of degree counts of nodes. The result is shown in Figure 8(a). When we bound the degree by k , the message is more evenly distributed among the nodes. So, we leave the choice of k on the administrator depending on the TI system characteristics.

Handling dynamism is an important contribution in our paper. Recall that data items change at rate λ and query is made at rate μ . In Section 6.2.3 we give a bound on λ . To validate the bound, we experiment with $N = 100$ and $\mu = 1\text{query/minute}$ and change the value of λ as some multiple of μ . The message overhead for an hour of inserting data items at rate λ and the range queries at rate μ is shown in Figure 8(b). The straight line shows the message overhead due to the multicast. Two curves intersect at $\log \frac{\lambda}{\mu} = 1.65$, $\lambda \approx 45\mu$ that is pretty close to the bound $\lambda = \frac{N}{\log N} \mu = 50\mu$. When λ exceeds the bound, multicast requires less messages. In that case, those data items are not inserted into the overlay. Earlier we showed that to answer $q(a, r)$ the number of nodes visited is $N|r| + \log(|r|)$. In Figure 8(c) we evaluate it. The upper line indicates the analytical result while the lower one is the simulation result.

The next experiment shows the scalability of Q-Tree with respect to the number of local devices at each TI site. We consider each local device with 10 different attributes in 100 different sites. Figure 9(a) shows the latency of insertion into the overlay. We measure the latency in 3 cases: 1) inserting all data items instantaneously from each site, 2) inserting 500 data items per second from each site and 3) inserting 1000 data items per second.

To measure the performance of load balancing algorithm, we insert data items via a right skewed distribution (beta(4,2)) for 100 nodes in the system. We define γ as the current ratio of maximum and minimum load of the system and set the desired ratio $\alpha = 2$. The system is ‘loaded’ if $\gamma > \alpha$. Figure 9(b) shows the distribution of nodes with respect to the current load at different passes of the algorithm. At the beginning, $\gamma = 6.0968$ and the algorithm

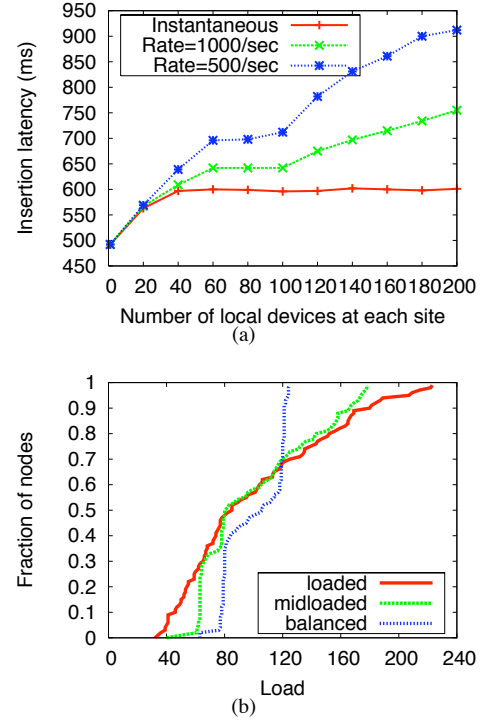


Figure 9. (a) Latency of data item insertion/update (10 sites) (b) Load distribution across nodes.

terminates when $\gamma = 1.98 < 2$, where the load is balanced among the nodes.

10. Conclusion

Tele-immersive interactive systems present new challenges in the distributed systems area. We have designed and evaluated Q-Tree, a multi-attribute query framework for querying dynamic TI systems. Our performance results show that our system scales with the number of local devices and data items and allows sufficient attribute churns. Q-Tree servers any complex multi-attribute range query in low latency and minimum overhead. Beyond TI, any system that needs multi-attribute range queries in time-sensitive, lightweight and efficient manner, will get benefits from Q-Tree.

Acknowledgment

This research is supported by grants NSF CNS 05-20182, NSF CNS 07-20702, and NSF CNS 08-34480. The presented views are those of authors and do not represent the position of NSF.

References

- [1] J. Aspnes and G. Shah, "Skip graphs," in *Proc. of SODA*, 2003.
- [2] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram, "Querying peer-to-peer networks using p-trees," in *Proc. of WebDB*, 2004, pp. 25–30.
- [3] A. Gupta, D. Agrawal, , and A. Abbadi, "Approximate range selection queries in peer-to-peer systems," in *Proc. of CIDR*, 2003.
- [4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in *Proc. of ACM SIGCOMM*, 2001, pp. 161–172.
- [5] I. Stocia, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. of ACM SIGCOMM*, San Deigo, CA, August 2001, pp. 149–160.
- [6] I. Gupta, K. Birman, P. Linga, A. Demers, and R. Renesse, "Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead," in *Proc. of IPTPS*, 2003, pp. 81–86.
- [7] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proc. of the IFIP/ACM ICOSP*, 2001, pp. 329–350.
- [8] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica, "Complex queries in dht-based peer-to-peer networks," in *Proc. of the first International Workshop on Peer-to-Peer Systems*, 2002, pp. 242–259.
- [9] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram, "P-ring: An efficient and robust p2p range index structure," in *Proc. of SIGMOD*, 2007.
- [10] R. Renesse and K. Binnan, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," vol. 21, no. 2, pp. 164–206, May 2003.
- [11] P. Yalagandula and M. Dahlin, "A scalable distributed information management system," in *Proc. of ACM SIGCOMM*, 2004, pp. 379–390.
- [12] CoMon, "<http://comon.cs.princeton.edu/>."
- [13] J. Albrecht, C. Tuttle, A. Snoeren, and A. Vahdat, "Planetlab application management using plush," vol. 40, no. 1, pp. 33–40, 2006.
- [14] J. Liang, S. Ko, I. Gupta, and K. Nahrstedt, "Mon: On-demand overlays for distributed system management," in *Proc. of Usenix WORLDS*, 2005.
- [15] S. Ko, S. Yalagandula, I. Gupta, V. Talwar, D. Milojevic, and S. Iyer, "Moara: Flexible and scalable group-based querying system," in *Proc. of ACM/IFIP/USENIX Middleware*, 2008.
- [16] MONET, "<http://cairo.cs.uiuc.edu/projects/teleimmersion/>."
- [17] Z. Yang, Y. Cui, B. Yu, J. Liang, K. Nahrsterdt, S. H. Jung, and R. Bajscy, "Teeve: The next generation architecture for tele-immersive environments," in *Proc. of ISM*, Irvine, CA, USA, 2005, pp. 112–119.
- [18] A. Corradi, L. Leonardi, and F. Zambonelli, "Diffusive load-balancing policies for dynamic applications," *IEEE Concurrency*, vol. 7, no. 1, 1999.
- [19] M. Goemans, "Minimum bounded degree spanning trees," in *Proc. of FOCS*, 2003, pp. 273–282.
- [20] P. 4hr Traces, "<http://www.eecs.harvard.edu/~syrah/nc/sim/pings.4hr.stamp.gz>."