Efficient Support for Range Queries in DHT-based Systems

Jun Gao Peter Steenkiste

December 2003
CMU-CS-03-215

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored in part by the Defense Advanced Research Project Agency and monitored by AFRL/IFGA, Rome NY 13441-4505, under contract F30602-99-1-0518, and in part by the NSF under award number CCR-0205266. Additional support was provided by Intel. Jun Gao is also in part supported by a Siebel Scholar award.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the funding agencies.

Abstract

In recent years, Distributed Hash Tables (DHTs) have been proposed as a fundamental building block for Internet-scale distributed applications. Important functionalities such as searching has been added to the DHT's basic look up capability. However, supporting range queries efficiently remains a difficult problem. In this paper, we describe an efficient algorithm that decomposes a range query with length R_q into $O(\log R_q)$ sub-queries that are resolved by separate nodes corresponding to each sub-query. Our system utilizes a novel logical tree data structure, the Range Search Tree (RST), which automatically groups registrations based on their values explicitly. Coupled with a dynamic load balancing mechanism, our system can handle both point and range queries efficiently regardless of the distribution of the registrations and queries. Our algorithm is fully distributed and we avoid conventional bottleneck problems encountered by tree-based systems. Extensive simulation results validate the effectiveness of the system.

Keywords: Distributed Hash Table, Content Discovery System, Range Queries, Range Search Tree

1 Introduction

Overlay networks based on Distributed Hash Table (DHT) [1] provide a scalable and robust data location and lookup substrate for large scale distributed applications and services, including wide area file and storage systems ([4], [16]), content/service discovery systems ([3], [6]), information retrieval systems ([15], [18]), and distributed databases [11]. A DHT supports exact name lookup only: a publisher registers a data item with a node in the system using an identifier that is typically the hash of the name of the data item, e.g., the name of a file. When searching for the data item, a client must know the "name" of the item, and then generate the same identifier by applying the hash function to the name.

To make a DHT-based application practically useful, the system must provide searching capability. As an example, imagine a nationwide road monitoring service that is built on top of a DHT. Devices such as cameras and sensors are installed along the road side or mounted on patrol cars to monitor traffic status and road conditions. Users of such a service may pose a variety of queries, e.g., "*list the locations of the cameras that are in Virginia and overlook a beach*".

Recently several groups [3, 15, 7] have studied how to use DHTs to support searching based on subset matching. In particular, a descriptive name is defined as a set of attribute value pairs (AV-pairs), and the system returns any name that matches the AVpairs specified in a query exactly. The basic idea is to register a name at multiple nodes according to each of its AV-pair, and a query that is the subset of the name may get resolved by visiting any of these nodes. While DHTs can be very efficient for point queries that require an exact match, it is a poor fit for **range queries** [9], [11], where a range rather than a specific value is specified for an attribute.

Range queries are common and extremely important for discovery and exploration purposes, as a user may not know exactly what he is looking for. For example, a driver may issue the query "*return the speed observed by cameras that are between exit 10 and exit 50* ($\texttt{10} \leq \texttt{exit} \leq \texttt{50}$)", so that he may choose to get off the highway early to avoid congestion further down the road. A police patrolling a highway section with speed limit of 55 mph may ask the system to "*return the list of cameras whose observed speed is higher than 75 mph* ($\texttt{speed} \geq \texttt{75}$)". A naive way to resolve a range query would issue separate point queries to nodes that correspond to each possible value within the query range. For large range queries, which are typical for exploration purpose, this becomes very expensive.

In this paper, we propose an efficient range search scheme that utilizes an implicit logical tree data structure, the Range Search Tree (RST). Each level of the RST corresponds to a different data partitioning granularity. Registrations are sent to nodes at different levels. A range query is decomposed into $O(\log R_q)$ sub-queries, where R_q is the range length, and resolved by nodes that correspond to each sub-query. The RST is an implicit data structure that is only "filled in" as it is needed to support range queries. The RST is mapped onto overlay network nodes in a completely distributed fashion by application end points. The RST works in a synergistic way with a dynamic load balancing mechanism that spreads the load associated with popular names across multiple nodes.

The rest of the paper is organized as follows. Section 2 provides an overview of the system. We present the registration and query algorithms supporting range queries in Section 3. We quantitatively analyze our system in Section 4. We describe system optimizations in Section 5. Section 6 presents the results of our simulation study. We discuss related work in Section 7 and conclude in Section 8.

2 System Overview

We describe the challenges associated with supporting range searches in the context of a DHT-based content discovery system (CDS).

2.1 Content Discovery

A CDS [7] is a distributed system that enables content discovery. Contents are represented with descriptive names, known as *content names*. A content name is a set of AV-pairs in the form of $CN : \{a_1 = v_1, ..., a_n = v_n\}$. The specific meaning of a content name depends on the application. In the traffic example, it is used to represent a camera, and may have attributes such as `speed, location, view` etc. Queries also consist of AV-pairs. We differentiate two types of queries: point queries and range queries. In a point query, all the AV-pairs are equality predicates, while in a range query, some of the AV-pairs are inequality predicates.

A CDS can be built on top of a DHT such as Chord [17]. Each node in the system is assigned a node ID as its overlay network address. Registration and query messages are delivered in the system based on node IDs. To enable search, content names must be registered first. In our system, a content name is registered with each node whose ID is numerically closest to the hash of one AV-pair in the name ($N_i = \mathcal{H}(a_i = v_i)$). When a node receives a registration, it inserts the name into its local database. To resolve a point query, the system applies the hash function to one of the AV-pairs in the query, and sends the query to the corresponding node. When a node receives a query, it will compare each AV-pair in the query against the AV-pairs in each name in its database, and return the querier the set of names that match the query. A content name matches a query if it satisfies all the AV-pairs in the query simultaneously.

It is worth noting that contents in our system may be dynamic, and they must periodically refresh and update themselves. Refresh messages are also important as a fault tolerance mechanism: if a node that is responsible for a content name leaves or crashes, the update messages will store the name at a live node whose ID is now closest to the hash. Before we discuss issues related to range queries, we first explain how CDS handles load balancing, which is critical to the system's performance.

2.2 Load Balancing

In real applications, the distribution of AV-pairs is often skewed. For example, some AV-pairs are common in many content names, or maybe popular in queries. As such, the node that is responsible for this type of AV-pair will be overloaded by registrations or queries, and will become a "hot spot", or bottleneck of the system. Without proper load balancing, the system's performance will degrade quickly. We briefly describe the particular load balancing mechanism we use. More details can be found elsewhere [7].

The idea is that when a node corresponding to a data item, e.g., an AV-pair, gets overloaded by registrations or queries, the system recruits more nodes to share the load. This set of nodes is organized as a logical matrix, termed *load balancing matrix* (LBM): Each column of the matrix holds a partition of the content names that contain this AV-pair (which helps spread registration load), and nodes in different rows within a column are replicas of each other (which helps spread query load).

An LBM starts with one node and expands and shrinks depending on the registration and query load. The number of partitions is proportional to the registration load: $P = \lceil L^R / C_R \rceil$, where L^R is this data item's registration load, e.g., the number of registrations, or the registration rate, and

C_R is the capacity of each node. Similarly, the number of replicas is proportional to the query load corresponding to this data item.

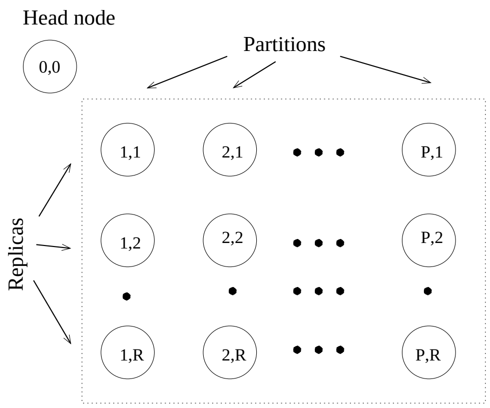

Figure 1: Load balancing matrix for a data item.

Figure 1 shows the layout of the matrix for a data item. Each node in the matrix has a column and row index, (p, r), and node IDs are determined by applying the hash function, $\mathcal{H}$, to the data item (AV-pair in this case) and the column and row indices together:

$$N_i^{(p,r)} = \mathcal{H}(data, p, r). \qquad (1)$$

To register a data item, the registering node selects a random partition from the matrix and registers with each node in that partition. To retrieve all the possible matches, a query must be sent to all the existing partitions in the matrix, however, only one node is selected from each partition because of the replicas. An important property of the LBM is that both the registration and query load are spread evenly within the matrix.

As an optimization, each matrix may use a *head node*, with ID $N_i^{(0,0)} = \mathcal{H}(data, 0, 0)$, to store its current size and coordinate the matrix expansion and shrinking. In [7], fault tolerant mechanisms on restoring the size information when the head node fails are discussed. An end point can retrieve the size of the matrix from the head node. To ensure that the head node will not become a bottleneck, end points can also obtain the size efficiently by conducting binary probing along the two dimensions, and cache the size for future use.

2.3 The Range Query Problem

In a range query, at least one attribute is specified with a range instead of a single value. If the query also contains AV-pairs with equality predicates, we may choose one of them for hashing, and the inequality comparison will be done at the corresponding node. This way we essentially treat the range query as a point query. However this choice may not always be available, e.g., when all AV-pairs in the query contain ranges. In addition, the AV-pair with fixed value may be popular and correspond to many partitions, and we may not want to query that matrix for performance reason. In this paper, we focus on the scenario where an AV-pair with a range is used for query resolution purpose.

There are two straightforward ways of supporting range queries, depending on how registration is done. First, we still apply the hash function to the attribute and value together as we did before. The option is efficient for point queries, but to resolve a range query, $Q : [v_s, v_e]$, it must be broken up to $R_q (= v_e - v_s + 1)$ sub-queries, and sent to each node that corresponds to a value in the range. This approach works well if registrations are dense and queries cover relatively small ranges. However, it is an $O(R_q)$ approach and the number of query messages will become unmanageable when the query range increases. Moreover, if the registrations within this range are sparse, most of the query messages are wasted.

Alternatively, we can apply the hash function to the attribute only ($N_i = H(a_i)$). In this case, all the content names that share the same attribute will end up registering at the same node, irrespective of the value; and at query time, all point queries and range queries will also be sent to the same node for resolution. This approach performs well under light load in that no matter what the range size is, all queries will have the same overhead. However, this node will become overloaded as registration and query load increase. Fortunately, the load balancing mechanism will help by recruiting more nodes to distribute the load. But it will not be efficient for queries, since each partition contains registrations with random values, every query including point queries will have to be sent to all the partitions.

We observe that both approaches work well in some cases, while they perform poorly in other cases. The problem is that neither solution partitions the registrations according to their values in a way that matches the nature of the queries. An ideal solution would behave similarly to the first approach for attributes that experience mostly point queries or queries over small ranges, but would behave similarly to the second approach for attributes that experience light registration load and large query ranges. In the next section, we describe a tree-based approach that exhibits this adaptive behavior.

3 Range Search Mechanisms

Our design to support range search is centered around the range search tree (RST) data structure. We first describe the organization of the RST and then describe how it is used to support range queries.

3.1 Range Search Tree (RST)

For a given attribute a that takes on numerical values and may be searched for by range queries, the RST is a logical tree that is structured based on the domain of the attribute. We assume the domain of a is D_a; values can be continuous or discrete. We break up D_a into n sub-ranges and represent each sub-range by its lower bound. D_a is thus the union of the sub-ranges $\{v_0, v_1, ..., v_{n-1}\}$, where $v_i < v_j$, if $i < j$. For example, if the attribute is `speed` in mph, we could break up the speed range into sub-ranges of 5 mph, and the value 35 would represent the range $35 \leq$`speed`< 40. Note that the size of each sub-range may not be equal if we have prior knowledge of the set of values a can take on.

The RST is a complete and balanced binary tree with n leaf nodes, and $\lceil \log n \rceil + 1$ levels. (We assume n is a power of 2; Otherwise, we round it up to the next power of 2 by filling extra values.) We label each level with a number, with leaf level being level 0. Each node in the tree represents a different range, and is denoted by its range, $N[v_s, v_e]$. Each leaf node corresponds to a single value in the domain, with the leftmost node corresponding to v_0. Each non-leaf node corresponds to the union of its two children's ranges. At level l, the range of the ith node from the left represents the range $[v_i, v_{i+2^l-1}]$. We observe that at every level, the union of all ranges covers the full domain.

A special case of an RST is a "unit RST" in which the domain is within the integer domain and each leaf node represents one integer value. Figure 2 is an example of such an RST for an attribute with domain $[0, 7]$. In the rest of the paper, we will present algorithms for a unit RST, but they generalize easily to a general RST: the unit RST can be considered as the index tree of a general RST.

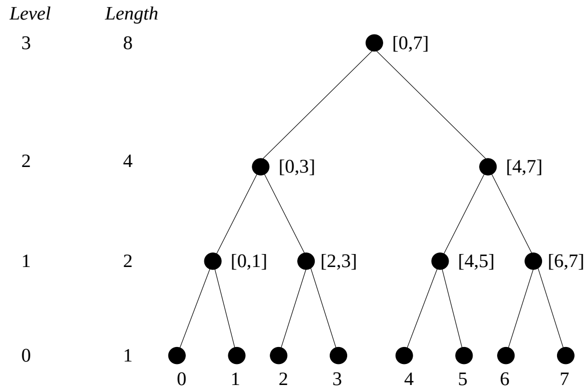

Figure 2: An example unit range search tree. The length of a node in unit RST at level l is 2^l.

The overall goal of our system design is to minimize the number of registration and query messages. By utilizing the RST, we can aggregate registrations based on their values at different levels of the tree. As a result, queries can be directed to the proper levels according to their ranges, and avoid the problems encountered by the two naive approaches in Section 2.3.

3.2 Registration

The goal of the registration algorithm is to have content names registered at different levels of the RST, so that queries can be resolved efficiently. We first present a proactive registration approach that registers an AV-pair with all levels. This algorithm serves as the basis for discussion on the query mechanisms (Section 3.3). We describe a more space efficient registration approach in Section 5.

REGISTER-RST($\{a = v\}, CN$) {
    $max\_levels \leftarrow \lfloor \log(m) \rfloor$;
    foreach $0 \le l \le max\_levels$ {
        $s \leftarrow \lfloor \frac{v}{2^l} \rfloor$; // 2^l: node length at level l
        $e \leftarrow s + 2^l - 1$;
        $range \leftarrow [s, e]$;
        $(P, R) \leftarrow find\_matrix\_size(a, range)$;
        $p \leftarrow random(P)$;
        foreach $1 \le r \le R$ {
            $N^{(p,r)}_{range} \leftarrow \mathcal{H}(a, range, p, r)$;
            $register(N^{(p,r)}_{range}, \{a = v\}, CN)$;
        }
    }
}

Figure 3: The proactive registration algorithm used by each registering node to register $\{a = v\}$.

Figure 3 lists the registration algorithm used by a registering node. To register a name, CN, that contains AV-pair $\{a = v\}$, first, the height of the RST corresponding to a is computed based on the domain size (Line 2, where m is the length of D_a). For value v in D_a, since each node in the RST covers a unique range, v is thus within exactly one node's range at each level. This set of nodes forms a **path** from leaf node $[v, v]$ to the root. Lines 4-6 determine the node at level l in the RST whose range covers v, in particular, the starting point, s, is determined by dividing v with the length of the level l nodes.

Once the RST node in each level is determined, we map that node onto the overlay network. It is important to remember that one RST node corresponds to an LBM in the network. The *data* used in Eq.(1) to define this LBM is the tuple $(a, range)$. For registration, the registering node first determines the corresponding LBM's size (Line 7), and then picks a random partition (Line 8). Finally it sends the name to each node in the selected partition (Lines 9-12). The node IDs are determined by applying the hash function, $\mathcal{H}$, to the tuple along with the column and row indices (Line 10). As an example, Figure 4 illustrates the registration algorithm when registering pair $\{a = 3\}$. It is registered with each node within a selected partition of each level's LBM.

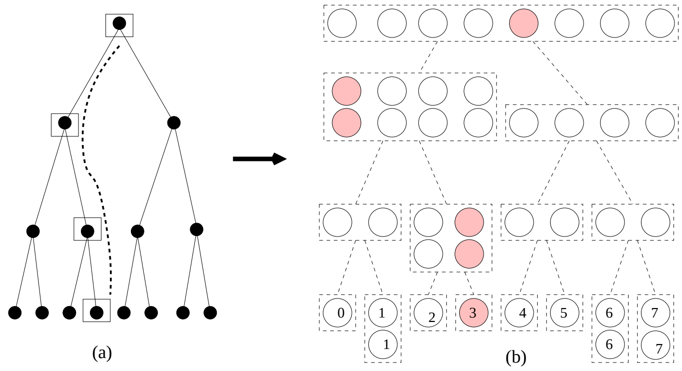

Figure 4: (a) The logical RST. The nodes in a box denote the ones that are identified by the registration algorithm, and the dotted curve illustrates the Path for value 3. (b) The set of physical nodes this RST is mapped to. Each circle represents a physical node and each dotted rectangle is an LBM that corresponds to a node in the RST. LBMs may have different shapes depending on the load it receives. The filled nodes are the ones that are selected to receive the registration of $\{a = 3\}$.

Since the structure of an RST is determined by the domain of the corresponding attribute, known to all nodes in the overlay network, this algorithm is carried out in a fully distributed fashion: a node locally computes the IDs of the set of nodes in the network that it should register with. It does not consult any node or traverse the tree from the root to determine where to register. Thus we avoid creating any bottleneck in the system.

The registration process automatically aggregates content names with different values at different granularity. Each non-leaf node collects all the registrations within its subtree. As the level number increases, since there are fewer nodes in the RST, the LBM corresponding to one RST node may contain multiple partitions. As an example, in Figure 4 each LBM at the leaf level occupies one partition, and the LBM at the root level consists of 8 partitions.

3.3 Query

Based on the above registration mechanism, there are many ways to resolve a range query $Q : \{a = [s, e]\}$. For example, the query may be sent to the root level. Alternatively, it may be sent to nodes in the leaf level. Finally it may be sent to some combination of nodes from different levels.

Different query algorithms not only affect the query efficiency, but they indirectly also have an impact on the registration efficiency. To evaluate the efficiency of a query algorithm, we define a parameter, *relevance*. Given Q with length $R_q = e - s + 1$, and suppose it is decomposed into k sub-queries which corresponds to k nodes in the RST, $N_1, ..., N_k$. The relevance of the algorithm is defined as:

$$r = \frac{R_q}{\sum_{i=1}^{k} R_i},$$

where R_i is node N_i's range length.

A low relevance query algorithm such as sending a point query to the root level, may incur high cost for both registrations and queries. The query cost may be high because the point query must be sent to the potentially many partitions at the root level. From registration's point of view, under high query load, the root level nodes must replicate, which means the contents that are not in the query range will be replicated unnecessarily.

In contrast, decomposing the query to leaf level nodes has a relevance of 1. From registration's point of view, there will be no unnecessary replications, but from the query's point of view, there may be too many sub-queries.

3.3.1 Range Decomposition Algorithm

We now describe our decomposition algorithm that simultaneously maximizes the relevance ($r = 1$) (to minimize the registration cost) and minimizes the number of sub-queries (to minimize the query cost).

In Appendix A, we prove the **decomposition theorem** that states for any query $Q : [s, e]$ with range R_q, it can always be decomposed to $O(\log R_q)$ disjoint sub-ranges and each sub-range corresponds to the exact range of a node in the RST. This set of nodes is known as the minimum cover (MC) of range $[s, e]$, and the highest node's level in MC is $\lfloor \log R_q \rfloor$.

Figure 5 is a decomposition algorithm that computes the MC for a given query. The RANGE-DECOMP function recursively decomposes a given range $[s, e]$ by finding one sub-range at a time. It first makes sure the range is valid (Line 2), and then computes its length (Line 3) and the maximum level that this range's MC corresponds to (Line 4). In the MC, the first node must be the highest node with a starting value s. Lines 5-13 find this node by examining the tree in a top-down fashion. Once that node is found, its range is inserted to the output list of sub-queries (Line 7), and the function recursively decomposes the rest of the range (Line 8). Figure 6 is a decomposition example.

3.3.2 The Query Algorithm

Figure 7 is the pseudo code used by a query issuer to send its query to the overlay nodes. To resolve a query $Q : [s, e]$, the querying node locally calls the function RANGE-DECOMP to decompose Q to a list of sub-queries (Line 3). For each sub-query (Lines 4-12), based on its range, the query issuer retrieves the corresponding LBM's size (Line 6), and then sends it to a random node in each partition (Lines 7-11).

This query algorithm is fully distributed and deterministic, in that the decomposition is done by the querying node itself based on its query range, and no traversal of the tree is needed. The algorithm naturally separates queries with different query range size. A point query will be sent to the leaf level, and a large query will use high level nodes in the tree, but no higher than level $\lfloor \log R_q \rfloor$ as defined by the MC.

```
RANGE-DECOMP(Q : [s, e], query_list * ql) {
    if (s > e) return;
    R_q ← e − s + 1; // range length
    l ← ⌊log(R_q)⌋; // maximum level
    while (l ≥ 0) {
        if (remainder(s / 2^l) == 0){
            insert_to_query_list(Q : [s, s + 2^l − 1], ql);
            RANGE-DECOMP(Q : [s + 2^l, e], ql);
            break;
        } else {
            l ← l − 1;
        }
    }
    return;
}
```

Figure 5: The local range decomposition algorithm.

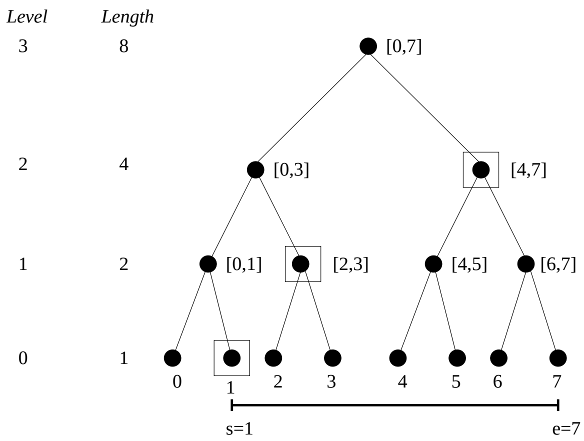

Figure 6: Range [1, 7] is decomposed to 3 sub-ranges indicated by the nodes with a box, which form the MC for this range.

4 System Analysis

In this section, we analyze the cost of the registration and query algorithms described in the previous section. We assume the domain $D_a = [0, m-1]$, and the total number of levels in the RST is $T = \lceil \log m \rceil + 1$.

4.1 Number of registration messages

From Section 3.2, we know that the number of registration messages needed to register pair $\{a = v\}$, N_{R_v}, is determined by the query load that comes to this RST. More specifically, it equals to the sum of the number of replicas (rows) in the LBMs that are on the path, $Path(v)$, that covers value v in the RST (See Figure 4 for an example). Consider an arbitrary query load that consists of L^Q queries. Suppose the query load on the node at level t in $Path(v)$ is L_t^Q, the number of replicas in its LBM is $\lceil \frac{L_t^Q}{C_Q} \rceil$, with C_Q being the query capacity of a node (assume homogeneous nodes). The

```
QUERY-RST(Q : [s, e]) {
    init_list(query_list * ql);
    RANGE-DECOMP(Q : [s, e], ql);
    foreach Q : [s', e'] ∈ *ql {
        range ← [s', e'];
        (P, R) ← find_matrix_size(a, range);
        foreach 1 ≤ p ≤ P {
            r ← random(R);
            N_range^(p,r) ← H(a, range, p, r);
            send_query(N_range^(p,r), Q : [s', e']);
        }
    }
}
```

Figure 7: The query algorithm.

total number of registration messages needed for $\{a = v\}$ is then:

$$N_{R_v} = \sum_{t=1}^{T} \lceil \frac{L_t^Q}{C_Q} \rceil.$$

Proposition 1

$$T \leq N_{R_v} \leq T + \lceil \frac{L^Q}{C_Q} \rceil$$

Proof:

The property of the ceiling function gives,

$$1 \leq \lceil \frac{L_t^Q}{C_Q} \rceil \leq \frac{L_t^Q}{C_Q} + 1$$

Apply sum to all terms,

$$T \leq N_{R_v} = \sum_{t=1}^{T} \lceil \frac{L_t^Q}{C_Q} \rceil \leq T + \frac{\sum_{t=1}^{T} L_t^Q}{C_Q} \qquad (2)$$

We divide L^Q into two sets, L^{Q_v}, which consists of all the queries that contain v, and $L^Q - L^{Q_v}$, which is the rest of the queries. Queries in $L^Q - L^{Q_v}$ do not contribute load to the nodes in $Path(v)$. Each query in L^{Q_v}, after decomposition, contributes exactly one sub-query to one node in $Path(v)$, since nodes in the MC have disjoint ranges (by definition). Therefore,

$$\sum_{t=1}^{T} L_t^Q = L^{Q_v} \leq L^Q.$$

Substitute this into (2), we get Proposition 1. □

Proposition 1 gives the upper and lower bounds of the number of registration messages needed for any value under an arbitrary query load. The maximum occurs when the value is contained in every query in L^Q. The minimum is T, which corresponds to when the value is not contained in any query, and the cost comes from registrating with each level of the RST. In Figure 8, the area between the two solid lines shows where N_{R_v} lies.

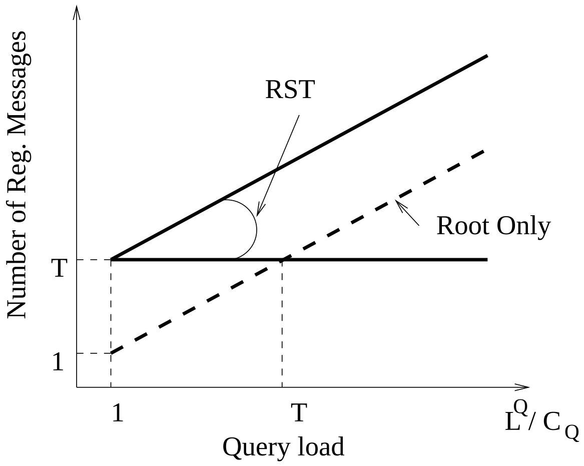

Figure 8: Number of registration messages needed for a value v as a function of query load.

We now compare the RST scheme with the scheme where we apply the registration and query load to the root level nodes only. In the Root only case, the number of registration messages needed for any value is determined by the number of replicas the root level matrix has: $N'_R = \lceil \frac{L^Q}{C_Q} \rceil$, which, indicated by the dotted line in Figure 8, grows linearly with the query load.

When query load is low, e.g., $L^Q/C_Q \leq T$, i.e., the number of replicas at the root level is smaller than the height of the RST, and the Root Only case incurs less cost than using RST from registration's point of view. However as load increases beyond that point, the average cost of RST is generally much smaller than Root only case.

4.2 Number of query messages

Similarly we can compute the number of query messages needed to resolve a given query, $Q : [s, e]$, with $R_q \leq m$. Assume the size of Q's MC is K, from the decomposition theorem, we know $1 \leq K \leq 2\lfloor \log R_q \rfloor \leq 2\lfloor \log m \rfloor$.

Consider an arbitrary registration load that consists of L^R registrations. The number of query messages needed to resolve Q equals to the sum of the number of partitions in the LBM that corresponds to each node in the MC:

$$N_Q = \sum_{k=1}^{K} \lceil \frac{L_k^R}{C_R} \rceil$$

where C_R is the registration capacity of each node, and L_k^R is the registration load observed on the kth node. Since nodes in the MC are disjoint from each other, each registration in L^R occurs in L_k^R at most once, thus,

$$N_Q = \sum_{k=1}^{K} L_k^R \leq L^R,$$

Similar to the previous derivation, we have,

Proposition 2

$$1 \leq K \leq N_Q \leq K + \lceil \frac{L^R}{C_R} \rceil$$

Proposition 2 gives the upper and lower bounds of the number of query messages needed for any query under an arbitrary registration load. The maximum occurs when all the registrations are within the query's range, and the minimum occurs when the query's MC contains one sub-query and its corresponding matrix has only one partition.

Similar to the above analysis, if we send the query only to the root level nodes, and the number of query messages needed equals to the number of partitions the root level matrix has: $N'_Q = \lceil \frac{L^R}{C_R} \rceil$. For low registration load, $L^R/C_R \leq K$, the Root only case performs better than RST.

4.3 Summary

We make the following two observations that will help in optimizing the algorithms presented in Section 3. First, since queries will not be sent to any level higher than $\lfloor \log R_q \rfloor$, if no queries have a range larger than R_q, all the registration messages to levels higher than $\lfloor \log R_q \rfloor$ are unnecessary. Second, if both the registration and query load for a subtree are low, using the RST for this subtree results in more registration and query messages than if we collapse the subtree to just the root of the subtree.

In the next section, we will present a protocol that is based on the first observation and it ensures registrations do not go to levels that are not queried. Similar optimizations can be done based on the second observation to ensure that registrations and queries will be sent to the root of a sub-tree when load is low to further reduce the cost. However, this part of the protocol (the sub-tree collapsing algorithm) is not presented and evaluated in this paper.

5 System Optimization

We now present a query-driven registration algorithm that is more space efficient in that it ensures that registrations only go to the levels that queries are occurring. It has three main phases: path instantiation, content pull, and path refreshing.

5.1 Path Instantiation

For an AV-pair, $\{a = v\}$, instead of registering it with all levels of the RST as described in Section 3.2, it is sent to the leaf level only. If the corresponding leaf node has not seen this value before, it will ask its matrix's head node to send a *path instantiation* (PI) message to the head node of its parent's matrix. The PI message informs the parent head node the existence of registrations on this child. Each non-leaf (head) node keeps a simple data structure that marks which child it has received a PI message from before. If the node has not sent a PI message to its parent before, it will in turn send a PI message to its parent. As such, a PI message may traverse all the way to the root node. We call the propagation of PI messages, the **instantiation** of a path. We set the number of levels that must be travelled up a tunable parameter, so that a PI message may stop before reaching the root. At this phase, registrations are not populated. Figure 9 is an example of path instantiation.

The PI is a light-weight process in that each node may send out the PI message at most once regardless of the number of registrations it receives, and a node receives at most two PI messages from its children. From registration's point of view, registering at the leaf level is beneficial, since that is the level where concurrent registrations least likely occur.

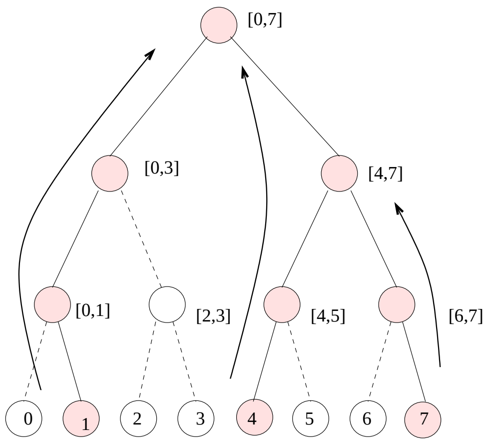

Figure 9: Three instantiated paths. Nodes shown are corresponding head nodes of the LBM for each range.

5.2 Content Pull

Queries are carried out the same way as before: they are issued to the matrices corresponding to the query's MC for resolution. However, a non-leaf node that receives a query may not have content names, even when the leaf nodes in the sub-tree have. In this case, the query initiator can decompose the query into linear number of sub-queries and send them to leaf level nodes for resolution.

Each non-leaf node maintains a query threshold, e.g., number of queries or query rate, and the content pull process starts when the threshold is crossed. It informs its head node the need of contents. The head node then issues a Content Pull (CP) message to the head nodes of its instantiated children. If the child does not have contents, it will recursively forward the CP message down its instantiated paths. The message will stop when it hits a node that has contents available. In the worst case, it will reach the leaf nodes. If a node has contents, it will send its contents to the parent's matrix. The parent node that previously forwarded the CP message waits until it gets contents back from its children, and then forwards the contents to its parent. The content pull process terminates at the node that initiates the CP process.

With path instantiation, a node knows which child it should contact, and thus we avoid broadcasting within the tree. This is especially important when the paths are sparse. Note that limited broadcasting is tolerable at high levels of the tree.

We let the content-holding node forward contents up along the path rather than send contents directly to the CP initiating node for the following reasons: (1) The intermediate nodes may cache the contents in case they themselves are queried in the future. (2) The intermediate nodes may condense or forward a subset of contents to the parent. (3) We avoid message implosion at the CP initiating node.

Compared to the proactive registration approach, the advantage is that we do not register at levels higher than what the queries need. The disadvantage is that it takes time for contents to propagate up the tree, and it is less efficient for the first set of queries that come to an internal node: they have to be resolved in two rounds and use more messages. However, we consider the content pull process as a transient stage, and the initial failures at these nodes for the first set of queries are similar to "cache misses".

5.3 Path Refreshing

Periodically, a leaf node that still holds contents sends a path refreshing (PR) message up the tree similar to the PI message. Each intermediate node will refresh its state. The top level node replies this PR with a PR reply (PRR) message down the tree. A PRR message may split. Each node along the path fills in their intention on whether they need contents anymore depending on whether it has seen enough queries in the recent past.

Once a leaf node receives the PRR, it will send along the path an update that includes the new registrations it has received over the last time period together with the old ones that are still valid. The update does not include the expired content names, since they will expire automatically themselves on the parent nodes. The update messages go to nearest ancestor that needs contents first, and that ancestor will merge and relay to its ancestor that need contents.

The purpose of the path refreshing is three-fold: (1) Restore states if parent crashes; (2) If the refreshing message did not arrive on time from a child, that means either the child node has no more contents or it has crashed, and the parent will remove this branch. This path will not be searched in the future; (3) Ensure contents are placed only at nodes that are being queried.

6 Evaluation

In this section, we present evaluation results obtained from simulation. We implemented the range query mechanisms in an event-driven simulator developed in [7]. The simulator allows us to set up an overlay network with a configurable number of nodes, each of which can handle events related to registrations and queries. The simulator supports the load balancing mechanisms described in Section 2.2, and it assumes the existence of a DHT-based overlay mechanism for routing and forwarding. We use the cryptographic function SHA-1 as the system-wide hash function, $\mathcal{H}$.

6.1 Methodology

In our experiments, we set up an overlay network with $10,000$ nodes. Each node is configured with $500Kbps$ link bandwidth (DSL level) for registrations and queries. Correspondingly, each node sets thresholds of $50reg/sec$ and $200q/sec$ as the maximum sustainable registration and query rate (assume 1000-byte registration packet size and 250-byte query packet size). We assume a node's performance in handling registrations and queries is limited by its link bandwidth rather than its computation power, since a modern PC can easily process $1000reg(query)/sec$ with a reasonable sized database. A similar assumption is used in [11]. As such, when a node observes that one of these thresholds is crossed, its corresponding matrix will start to expand.

6.1.1 Work load

We use synthetic work loads to drive our simulations. We consider one attribute, a, which can take on 200 different values. Its corresponding RST has 9 levels ($= \lceil \log 200 \rceil + 1$). Each registration load consists of a set of content names, and each name contains one AVpair. The sender of a name is selected randomly from the nodes in the system and the arrival times for names are modeled with a Poisson distribution. Query loads are similar, except that instead of one value, a range may be specified.

In the registration and query load, the possibility that a value occurs in a name or is the middle value of a query range is equally likely. This is a conscious decision in that having skewness in the load will exercise the system's load balancing mechanism, not the RST mechanism, which is the focus of our evaluation.

6.1.2 Schemes Evaluated

We focus on the following schemes:

- Root Only: All registrations and queries are sent to the root level nodes.
- Leaf Only: Registrations are sent to corresponding leaf nodes, and queries are decomposed to sub-queries that correspond to leaves.
- RST: Queries are decomposed into $O(\log R_q)$ number of sub-queries. As described in Section 5, registrations are propagated only to the highest level needed by the queries.

We use two metrics to evaluate the system: the number of registration and query messages needed for each registration and query, as defined in Section 4. To ensure consistency across experiments, we conduct our measurements when the system operates in a stable state: the sizes of the LBMs stop changing and contents are propagated to the proper level in the RST.

6.2 Query Cost

We first examine the query cost, which is determined by two factors: the query range length and the registration load.

6.2.1 Query cost as range increases

In this set of experiments, we first inject a registration load into the system, and then issue a query load into the system with rate $200q/s$ (The query rate does not affect the query cost here). In each query load, there are 10,000 queries, and the range lengths of the queries are the same. After each run, we compute the average number of query messages. We vary the range length from 1 (point query) to 100 (50% of the domain). We use two registration loads with low and high arrival rates: $25reg/sec$ and $2000reg/sec$.

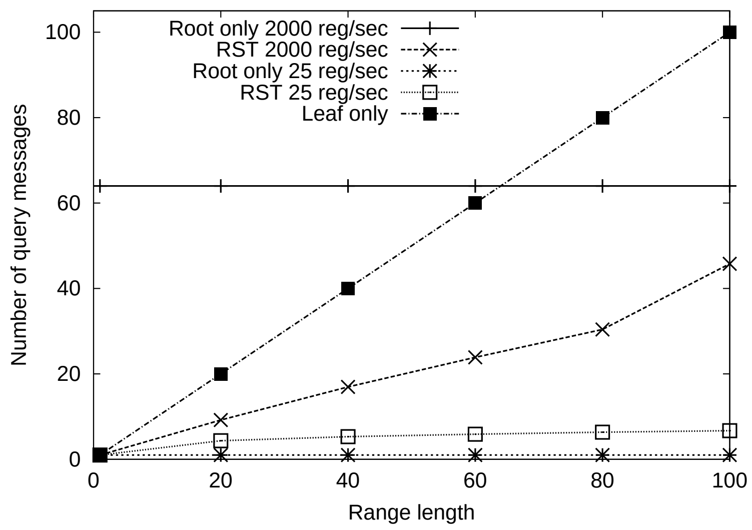

Figure 10: Query cost vs. Range length

Figure 10 shows the number of query messages as the query range increases under five different scenarios. For the Root Only case, with low registration load, it performs the best, since regardless of the range length, the number of query messages needed is always 1. The reason is that under the low registration rate (1/2 of the threshold), the root level has only 1 partition. However it behaves poorly under the high load, where 64 partitions are created at the root level (larger than the theoretical value of 40 due to the Poisson distribution). Irrespective the range length, 64 query messages are needed for all queries, since all partitions must be visited.

In the Leaf Only case, the number of query messages grows linearly with the range length under both registration loads, since no partitions were created at leaf level. For example, even under the high load, the average rate a leaf node observes is $10reg/sec$ $(=\frac{1}{200}2000)$, which is much smaller than the registration threshold $(50reg/sec)$.

In the RST case, for point queries the performance is the same as Leaf Only case (1 query). As the range increases, for low registration load, the performance of RST is slightly worse than the Root Only case, as the number of query messages grows logarithmically as we expected. However, for the high registration load, the RST scheme requires far less number of query messages than the Root Only case due to the query decomposition, specially when the range is smaller than the domain size. The number of query messages in RST will degenerate to the Root Only case only when the range of a query is equal to the full domain, in which case, the query will be sent to the root level.

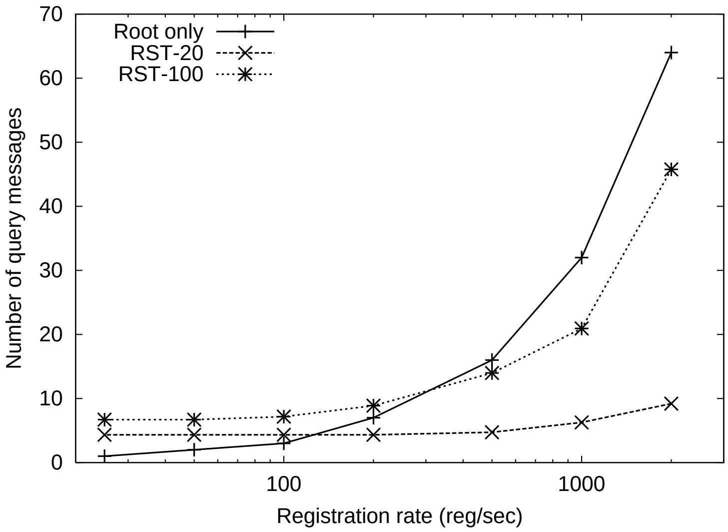

Figure 11: Query cost vs. registration load (in log scale)

6.2.2 Query cost v.s. registration load

Using the same setup as above, Figure 11 plots the average number of query messages needed under different registration loads. We use two query loads, with a range length of 20 and 100 respectively. In the Root Only case, for both ranges, as the registration load increases, the number of query messages increases linearly, since the number of partitions at the root level increases linearly.

In the RST case, for both ranges, when registration load is low, the number of query messages is higher due to the logarithmic decomposition. However, as the registration load increases, the number becomes smaller than the Root only case. For example, for range $R_q = 20$, the crossing point occurs when registration load is near $200reg/sec$, and the number of partitions needed in the Root case is 7. This agrees with our analysis in Section 4, since the size of the MC in the RST scheme is $K \leq 2 \log 20 = 8$. When registration load further increases, the number of partitions at root level becomes larger than 8, and the RST scheme performs better.

In summary, for high registration load, the RST scheme is very efficient for queries with any range lengths. We also note that had we incorporated the optimization that collapses towards the root of a subtree, we could have further improved the system's performance for low registration load.

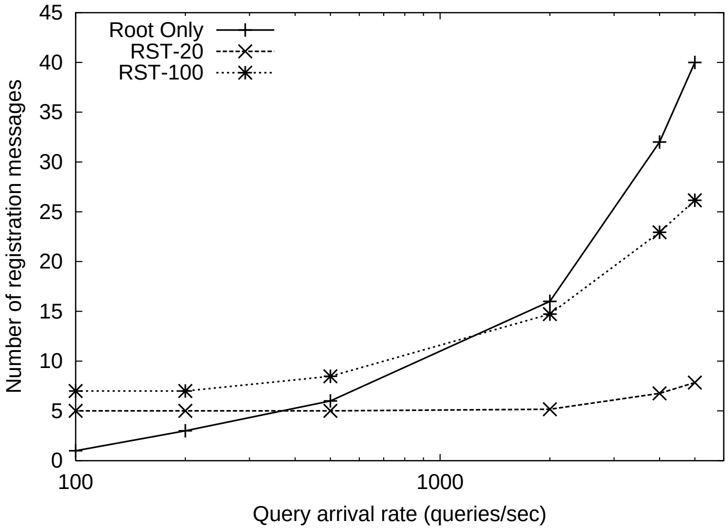

Figure 12: Registration cost vs. query load (in log scale)

6.3 Registration Cost

One factor that contributes to the RST scheme's query efficiency is that the RST requires registering an AV-pair at multiple levels. In this section, we examine the registration cost. In particular, we look at the relationship between registration cost and query load, since the number of registration messages needed per registration depends on the query load.

We first issue a low registration load to the system such that all the registrations can fit in one node. We then inject the query load into the system with different query arrival rates. We use two query loads with range 20 and 100 respectively. We compute the average number of registrations needed for each registration after each run. The results are shown in Figure 12.

For the Root Only case, as we expected, the number of registrations needed grows linearly with the query load regardless of the query range, since the number of replicas grows linearly with the query rate. For RST, when load is low, it uses more registration messages since it has to register at levels corresponding to the query length (5 and 7 levels for $R_q = 20, 100$). As query load increases, the number of registration messages increases slowly for small range (20) and fast for large range (100). The reason is that there are many nodes at the lower levels to share the load, and load on each node grows slowly, which results in less replications.

With a domain of 200, based on the analysis in Section 4, when the number of replicas needed $L \geq 9$, the proactive registration algorithm presented in Section 3.2 will perform better than the Root Only case. In Figure 12, this would correspond to a rate near $1000q/s$. By using the optimization in Section 5, when $R_q = 20$, RST starts to perform better at a much lower rate, since we do not need to register at levels above 5.

6.4 Effect of system parameters

Now we look at some of the parameters' effect on the system's performance.

6.4.1 The effect of *relevance*

First we evaluate the system's performance under several schemes with decreasing relevances.

1. **RST**: denoted as RST(log).
2. **RST(3)**: similar to RST, but instead of further decomposing a range, it uses three nodes from level $\lfloor \log R_q \rfloor$ in the RST as shown in the theorem's proof (Figure 17).
3. **RST(1)**: instead of decomposing a range, we use the node in the RST that corresponds to the common prefix of the two end values of the range; Note in this scheme, even a small range may correspond to a high level node, e.g., range [3,4]'s prefix node is the root [0,7] in Figure 2.
4. **Root Only**.

In experiments conducted here, we use query loads with range $R_q = 20$. Figure 13 shows the number of query messages as registration load increases. When registration load is low, RST uses the most number of query messages. However, as the registration load increases, nodes in the higher level will start to create partitions. Since RST(1) tends to use levels higher than necessary, the number of query messages grows faster and overtakes RST. RST also performs better than RST(3) since it uses more lower level nodes.

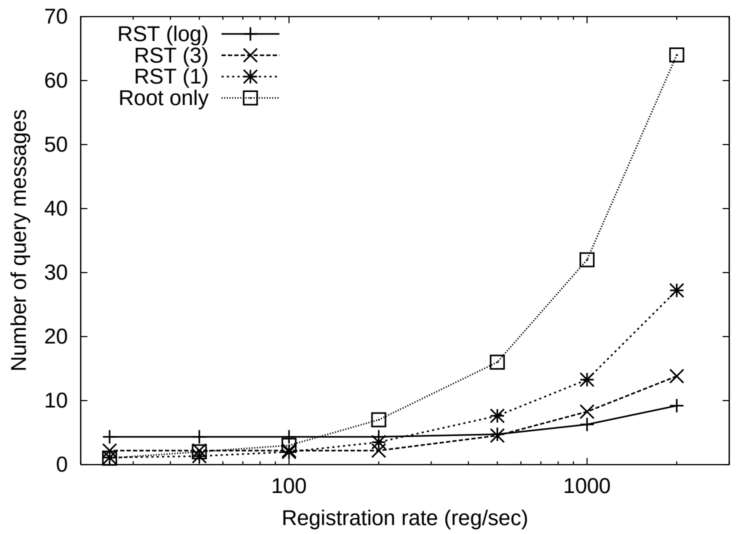

Figure 13: Effect of relevance on query cost. Query range $R_q = 20$

Figure 14 shows the number of registration messages as the query rate increases. When the load is low, all the RST cases start with more registration messages than Root due to the levels. For RST and RST(3), they use less number of messages than RST(1) since they do not need to go to all levels. As query load increases, since the relevance in RST(3) is smaller than RST, more registrations will be unnecessarily replicated, this corresponds to the increase of the average

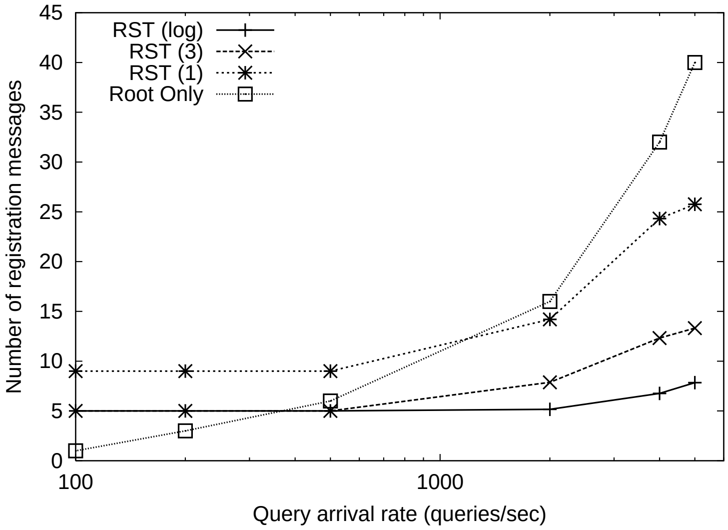

Figure 14: Effect of relevance on registration cost. Query range $R_q = 20$

registration cost. For RST(1), the relevance is even smaller, and the cost grows quickly. The Root Only scheme performs the worst, since it has the smallest possible relevance.

In summary, the RST scheme provides the best performance for both queries and registrations when system load is high.

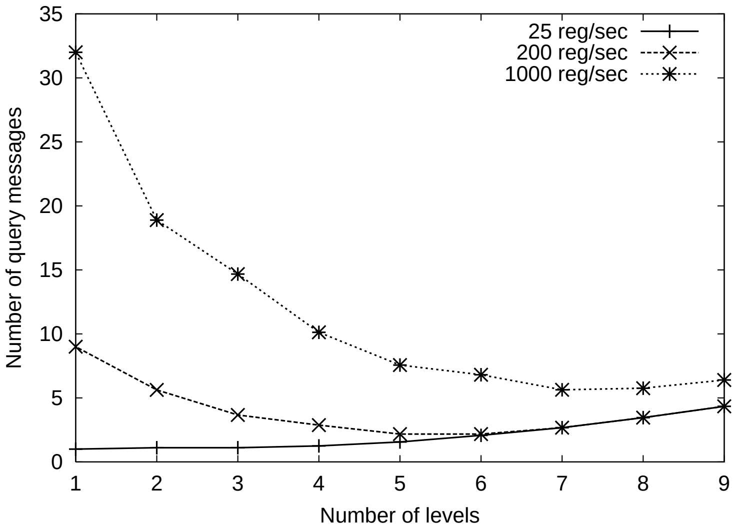

Figure 15: Query cost using different number of levels under different registration load. Query range $R_q = 20$.

6.4.2 The effect of number of levels

In our optimized RST scheme, the tree is built from the ground up, and registrations do not go to levels higher than the queried ranges. However, as we analyzed before, within a subtree, if both the registration and query load are low, we may be able to further reduce the query and registration

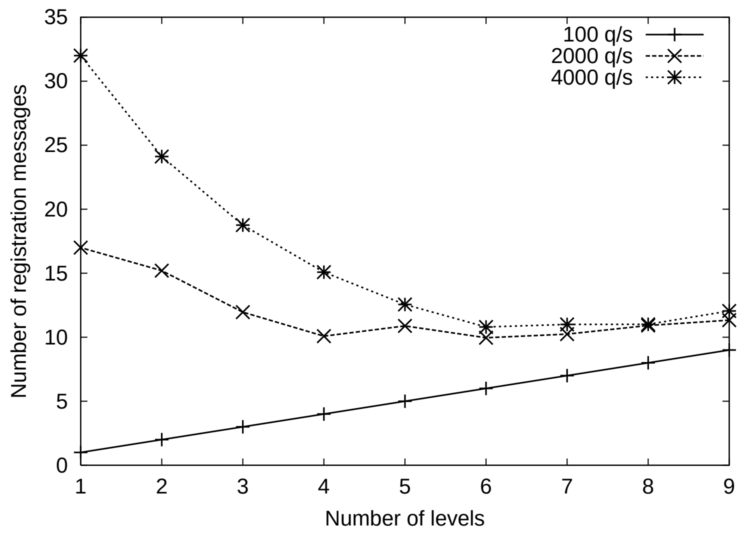

Figure 16: Registration cost using different number of levels under different query load. Query range $R_q = 20$.

cost by collapsing the tree towards the root level. To determine the proper set of levels needed, we conduct another series of experiments using a top-down tree construction approach, i.e., we run experiments with increasing number of levels from the root.

Figure 15 shows the effect on query messages. When registration rate is low, clearly 1 level (the root level) is sufficient, and we use 1 query message. The number of query messages grows logarithmically, since more levels means more sub-queries. For high registration load ($1000 reg/sec$), partitions are created at top levels. Performance improves dramatically as we use more levels. The reason is that since $R_q = 20$, when we do not have enough levels, all the sub-queries will fall to the existing lowest level. The number of nodes at that level doubles every time when a level is added. Each node in the new level corresponds to half of the partitions its parent has. The query cost (determined by the number of partitions) thus decreases exponentially. The cost reaches a minimum with 7 levels. At this configuration, the lowest level corresponds to level 2 in the original RST. Further increasing the number of levels, will start to increase the number of query messages linearly. This means decomposing the query further is not necessary since the load in the subtree rooted at level 2 nodes is fairly low. We should use level 2 as our leaf level, and register at level $2, 3, 4$ only.

Figure 16 shows the effect of the number of RST levels on the number of registration messages under several query loads. We observe a similar phenomenon. The registration cost reaches the minimum value with 6 or 7 levels for high query load. This agrees with the above analysis.

These evaluation results confirmed that the most appropriate algorithm should ensure that registrations are only needed at the levels corresponding to ranges in the queries. We call this set of levels the *band*. Levels above the band are not nececessary since the queries will not go there. Levels below the band are not necessary since at those levels load is low, and the root of the sub-tree should be used. The optimization algorithms in Section 5 ensures no levels above the band will be used, and the sub-tree collapsing algorithm will ensure levels below the band are not used.

7 Related work

Efficiently supporting range queries in DHT-based systems has been posed as an open question in [9],[11]. There have been some recent efforts in addressing this problem.

In a recent report [13], Prefix Hash Tree (PHT) is proposed to support range queries. The PHT is conceptually similar to our RST data structure. However there are important differences between the two systems when using such an implicit data structure. Due to their lack of details, we can only compare the two schemes at a high level. The realization of PHT is in a top-down fashion and the split of a parent depends solely on the registration load. Unlike our scheme, where we store contents and resolve queries at levels corresponding to query ranges, PHT is a trie, and only leaf nodes store contents. Queries are first sent to the node corresponding to the common prefix of the range and traverse down to the leaves. In our scheme, since contents are aggregated at different levels of the tree, no tree traversal is needed. In addition, our evaluation shows that the query decomposition algorithm is more efficient than querying the common prefix node.

In [2], the authors use space filling curves as hash functions in CAN [14] based DHT systems to address range queries. In this scheme, a query is required to be sent to a node within the range first and let that node propagate the query. Our system works with no assumption on the type of DHT used, and queries are sent to nodes that contain potential matches without having to discover them. SkipNet [10] propose a lexicographic order preserving DHT node ID assignment mechanism, and can thus allow data items with similar values to be placed on contiguous nodes. This facilitates range search, but for data that are continuously spread out, the number of nodes must be visited is still linear to the query range due to lack of aggregation. In [8], a mechanism based on locality sensitive hashing function is used for range queries in the context of relational databases. The database is horizontally partitioned, and the hash function is applied to the query range to locate which partitions need to be visited.

In a different but similar context, Li et al [12] proposed a distributed mechanism to partition a multi-dimensional space using a data structure similar to K-d trees to support range queries in sensor networks. Our RST scheme can be readily extended to high dimensions to support multi-dimensional range queries.

In traditional parallel databases, a large relation is often partitioned among multiple disks. The common horizontal partitioning techniques include round-robin, hash partitioning and range partitioning [5]. A good partitioning technique that works well for both point and range queries is to partition using their values. A centralized partition vector must be consulted first before issuing a query. In our systems, consider the collection of all the content names in the system as a large relation, registering at the root level only and registering at the leaf only corresponds to the round-robin and hash partition techniques respectively. The RST based mechanism partitions the relation multiple times at different granularity based on the values. Thus, queries can be resolved distributed without using a centralized partition vector.

8 Conclusions

In this paper, we presented an efficient scheme to address the range query problem in DHT-based systems. Our algorithm relies on Range Search Trees for content registration and query resolution. Registrations are aggregated at different levels of the tree based on their values. Queries can thus be decomposed to a small number, $O(logR_q)$, of sub-queries for efficient resolution. The RST is a logical and implicit data structure, and the mapping of RST to overlay nodes, registrations and queries are carried out in a fully distributed fashion. There is no need to maintain or construct the tree structure. Since we distribute the contents at proper levels, no traversal of the tree is needed for registrations or queries. Our RST mechanism works in concert with a load balancing mechanism and can handle both point and range queries efficiently. Our extensive simulation results verified the effectiveness of our system. We plan to deploy a real implementation of the system on the PlanetLab testbed.

References

[1] Project IRIS, http://iris.lcs.mit.edu.

[2] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *Proceedings of P2P2002*, Sept. 2002.

[3] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proceedings of Pervasive 2002*, Zurich, Switzerland, August 2002.

[4] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of SOSP 2001*, Banff, Canada, October 2001.

[5] D. Dewitt and et al. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[6] Jun Gao and Peter Steenkiste. Rendezvous points-based scalable content discovery with load balancing. In *Proceedings of the Fourth International Workshop on Networked Group Communication (NGC'02)*, pages 71–78, Oct. 2002.

[7] Jun Gao and Peter Steenkiste. Design and evaluation of a distributed scalable content discovery system. *IEEE Journal on Selected Areas in Communications*, 22(1):54–66, January 2004.

[8] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of CIDR 2003*.

[9] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In *Proceedings of IPTPS'02*, March 2002.

[10] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of USITS'03*, March 2003.

[11] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *Proceedings of the 29th VLDB*, 2003.

[12] X. Li, Y. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of SenSys'03*.

[13] S. Ratnasamy, J. Hellerstein, and S. Shenker. Range queries over dhts. Technical Report IRB-TR-03-009, Intel Corp., June 2003.

[14] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings of SIGCOMM 2001*, pages 161–172, San Diego, CA, August 2001.

[15] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of Middleware 2003*, Rio de Janeiro, Brazil, June 2003.

[16] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of SOSP 2001*.

[17] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM 2001*, pages 149–160, San Diego, CA, August 2001.

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

A Range Decomposition Theorem

To facilitate our discussion, we define the following terms. Given a query $Q : [s, e]$ with range length $R_q = e - s + 1$, a **Relevant Node (RN)** is a node in the RST whose range is a subset of Q's range. A **Cover** is a set of relevant nodes, $Cover(Q : [s, e]) = \{RN_1, RN_2, ..., RN_n\}$, that satisfies the following two conditions:

(1) $RN_1 \cup RN_2 \cup ... \cup RN_n = [s, e]$

(2) $\forall i, j, RN_i \cap RN_j = \phi$.

The **Height** of a cover is defined as the level of the node in the cover that has the largest level number. The **Minimum Cover(MC)** is the Cover that has the smallest size among all possible covers. We establish the following two lemmas.

Lemma 1

The **MC** has the largest Height, H, among all possible covers, and $H \leq \lfloor \log_2 R_q \rfloor$. In other words, the MC does not contain nodes with level higher than $\lfloor \log_2 R_q \rfloor$.

Proof:

Prove by contradition. Suppose MC's height H is not the largest, then there must exist a different cover C' with height, $H' > H$. It follows that there must exist at least one RN, N', in C' at level H'. And N' is not in MC. Now consider the sub-tree rooted at N'. Suppose the range that corresponds to this sub-tree is $[s', e']$. The MC must contain at least two nodes from this sub-tree to cover this range. It is clear that we can substitute these two nodes in the MC with N' to reduce the size of the MC by 1. This contradicts the MC's definition.

Now we show $H \leq \lfloor \log_2 R_q \rfloor$. Let $h = \lfloor \log_2 R_q \rfloor$. $2^h \leq R_q < 2^{h+1}$. We show any node at level $h + 1$ or higher can not be a RN for R_q. Consider a node N, at level $h + 1$. Suppose its range is R_N with length 2^{h+1}. There are only the following possible relationships between R_N and R_q: (1) $R_q \subset R_N$, (2) $R_q \cap R_N \neq \phi$ and (3) $R_q \cap R_N = \phi$. In any case, node N can not be a RN, since it is never the case that $R_N \subset R_q$.

Lemma 2

For range $[s, e]$, if s is the starting point, or e is the ending point of a node at level l, then $|MC| \leq l + 1$, where $l = \lfloor \log R_q \rfloor$.

Proof:

Prove by induction. We show the statement holds when s is the starting point. When e is the ending point, it can be shown by symmetry.

As the base case, for point query with range length 1, $l = 0$, and $|MC| = 1$. As the induction hypothesis, suppose $|MC| \leq (l - 1) + 1$ for range with length $R_q/2$, where $l = \lfloor \log R_q \rfloor$.

For the induction step, we have range $[s, e]$ with length R_q and s is the starting point of a node at level l, where $l = \lfloor \log R_q \rfloor$. We also have $2^l \leq R_q < 2^{l+1}$.

We break the range into two sub-ranges: $[s, s + 2^l - 1]$ and $[s + 2^l, e]$. The first sub-range corresponds to exactly one node at level l. For the second sub-range, its length $R'_q = R_q - 2^l < 2^l$, and $s + 2^l$ is the starting point of a node at level $l' = \lfloor \log R'_q \rfloor \leq l - 1$. Now use the induction hypothesis, we know for this sub-range, its MC, $|MC'| \leq l' + 1 \leq (l - 1) + 1 = l$.

Thus, the MC for $[s, e]$ is the union of the two MCs for the two sub-ranges. It's size is thus: $|MC| \leq 1 + |MC'| = l + 1$. □

Decomposition Theorem

For a given $Q : [s, e]$, the size of its MC satisfies:

$$|MC| \leq 2 \cdot \lfloor \log_2 R_q \rfloor.$$

Proof:

First we observe that range $[s, e]$ intersects with at most 3 consecutive level $l = \lfloor \log R_q \rfloor$ nodes in the RST. (Figure 17). Let $p = \lfloor \frac{s}{2^l} \rfloor$, and the three nodes are: $N_1[p2^l, (p+1)2^l - 1], N_2[(p+1)2^l, (p+2)2^l - 1], N_3[(p+2)2^l, (p+3)2^l - 1]$.

We name their subtrees T_1, T_2, T_3. There are only three possibilities how the range $[s, e]$ may intersect with the three subtrees. (1) intersect with T_1 only; (2) intersect with T_1 and T_2; (3) intersect with T_1, T_2 and T_3. Now we examine each case.

Case 1: This occurs only when $s = p2^l$ and $R_q = 2^l$. We use node N1. Thus $|MC| = 1$.

Case 2: We know from Lemma 1 that MC includes nodes only from level l and below. We divide the range into two segments: $S_1 = [s, (p+1)2^l - 1]$, and $S_2 = [(p+1)2^l, e]$. Use Lemma 2, $|MC_{S_1}| \leq (l-1) + 1 = l$. Similarly $|MC_{S_2}| \leq l$.

MC_{S_1} and MC_{S_2} are disjoint, hence $|MC| = |MC_{S_1}| + |MC_{S_2}| \leq 2l$.

Case 3: We divide the range into three segments:

$S_1 : [s, (p+1)2^l - 1]$, $S_2 : [(p+1)2^l, (p+2)2^l - 1]$, and $S_3 : [(p+2)2^l, e]$.

We know the length of $S_2 = 2^l$, and at most one of S_1 and S_3 may have length longer than 2^{l-1}. Suppose $S_3 > 2^{l-1}$ and $S_1 < 2^{l-1}$.

$|MC_{S_1}| \leq (l-2) + 1 = l - 1$; $|MC_{S_2}| = 1$, and $|MC_{S_3}| \leq (l-1) + 1 = l$.

Hence $|MC| = |MC_{S_1}| + |MC_{S_2}| + |MC_{S_3}| \leq l - 1 + 1 + l = 2l$. $\square$

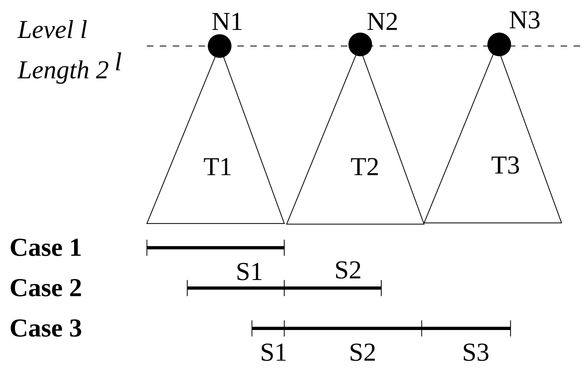

Figure 17: Illustration of range decomposition proof.