Distributed Segment Tree: A Unified Architecture to Support Range Query and Cover Query

Guobin Shen, Changxi Zheng, Wei Pu, and Shipeng Li

March 2007
Technical Report
MSR-TR-2007-30

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

1 Introduction

Cover query, which is defined as to find all the intervals (or ranges) currently maintained in the system that cover a given value, is highly desired functionality but has been rarely touched. Cover query has crucial impact on the performance of the distributed systems, in which maintenance of file integrity or object consistency is required, because the overall performance is critically determined by the effectiveness in handling the few specific objects or segments. For instance, the performance bottleneck of BitTorrent is well perceived to be the "last-block problem" [1] which refers to the excessive downloading time for the last few blocks. Similarly, in a distributed file storage system, all the hosts responsible for (typically few) specific updated file segments need to be reconciled. In all these example, how to efficiently *locate* the peers or hosts that have the specific block(s) or segment(s) affects the bottom line of the overall system performance.

On the other hand, range query, defined as to find all the values in a certain range over the underlying structure, arises from many critical real applications such as range partitioning in parallel databases and longest prefix match in routing schemes, has received considerable attention [2–4]. Recently, the research work on range query has been extended to support multi-attribute range queries where queries have multiple attributes and need to find the data items fitting in a certain range for each attribute [5, 6].

Efficient support of range query is a challenging task and usually requires new architectural design. Efficient support of cover query is also as challenging as that of range query if designed from scratch. Even provided the availability of many proposals that handle range query efficiently, it requires non-trivial extension or adaptation of those proposed solutions to support cover query.

In this paper, we present *distributed segment tree* (DST), a unified architecture that supports both range query and cover query in a uniform way and equally efficient. DST is designed to base on DHT so as to take advantage of all the merits of DHTs like self-organizing capability, scalability, robustness etc., and their simple interfaces. More specifically, DST is formed by overlaying a special data structure, *segment tree*, which is very effective for representing ranges (or intervals, segments), on top of the underlying DHT. Segment tree possesses a prominent feature - *computability*, that is, it allows an arbitrary range to be uniquely decomposed into the union of a *minimum* number of supported intervals (also called element subranges). By distributing these supported intervals to responsible DHT nodes, we implicitly reestablish the connection between the structural information of the segment tree and the underlying storage and routing substrate - DHT, which has originally been stripped off by the random hash operation. Because of the computability of the segment tree, a query (either range query or cover query) can be fulfilled via multiple parallel underlying DHT operations in a deterministic way, which lead to significantly shortened user observed latency. DST can be extended to support for multiple dimensional queries in a straightforward way.

We summarize our research contribution as follows:[1]

- Systematic study of the properties of segment tree. We identify the *computability* feature of segment tree and develop an efficient segment splitting algorithm to achieve the optimal decomposition. The unique decomposition enables the exploitation of parallelism among operations on subranges and helps to significantly reduce the user observed latency.
- Recognition of the duality between range query and cover query and the design of the distributed segment tree architecture to efficiently support both range query and cover query in a uniform way. We also extend DST to support multi-attribute range/cover queries and cover query for ranges. To the best of our knowledge, this is the *first* architecture that can achieve this.
- Proposal of novel load balancing mechanisms for both storage and query traffic considerations, and extensive evaluation of DST on the PlanetLab.

The rest of the paper is organized as follows. Section 2 describes related work. Then segment tree data structure and its properties are presented in Section 3. We present the DST construction mechanisms and the protocols in Section 4. Section 5 talks about various load balancing techniques. Experimental results are reported in Section 6, followed by discussions in Section 7. Finally, we conclude this work in Section 8.

2 Related Work

Proposals that support range query can be divided into two categories: those based on DHT, such as locality sensitive hashing [2] [3] and prefix hashing tree [6], and those do not, like Skip Graphs [4], Mercury [5] etc. Works based on DHT usually has good load-balancing performance, but requires non-trivial extension to support range query because of the exact key matching mechanism of DHT. On the other hand, works do not rely on DHT usually have more native support of range query since they do not use randomizing hash, but requires explicit consideration of load-balancing.

Gupta et al. designed a locality sensitive hashing function to ensure that, with high probability, similar ranges are mapped to the same node [8]. Similarly, Li et al. adopted a locality-preserving geographic hash function to provide multi-dimensional range queries support in sensor networks [3]. However, the localization of hash functions seems to work against load-balancing. Chawathe et al. proposed prefix hash tree (PHT) [6], which is essentially a new layer laid on top of a normal DHT, to maintain the range information and in return provides efficient support for range query. By laying upon DHT, it enjoys all the inherent

[1]The basic concept of DST has been presented in IPTPS 2006 workshop [7] for idea exchange purpose among research P2P community. In this paper, we perform more comprehensive study of DST, extend to multi-attribute case, address several optimization issues, and conducted more extensive evaluation.

benefits of DHT such as scalability, robustness and load-balancing etc. However, in the trie-based structure of PHT, keys are stored *only* at the *leaf* nodes that share the same prefix. As a result, a client has to spend several ($O(\log D)$ for a D-bit key) DHT `get` to reach the leaf nodes with the longest matched prefix, using binary search which is intrinsically *sequential*. The support to multi-dimensional range query is achieved through space-filling curves such as Hilbert curve and Gray code.

Skip Graphs [4], unlike the DHTs, does not necessitate randomizing hash functions and are therefore capable of range queries. Unfortunately, load balance between nodes becomes a serious problem. Possible improvement proposed is to either increase the number of virtual servers [9] or use an additional Skip Graph to track the load on each node. However, they have focused on the skewed storage load and the performance under skewed query load is unclear. Mercury [5], which does not use hash functions either, adopts circular overlays (the design philosophy is similar to that of DHT except not using hash) and stores data continuously in them (one circular overlay for one-dimensional data) to support multi-attribute range query. But, it requires explicit connections among the multiple circular overlays and the load balancing problem has to be explicitly addressed.

DST, while leveraging DHTs in a similar way as PHT does, differs from PHT in their essential data structures. DST maintains a highly regular segment tree data structure. Unlike PHT, the intermediate nodes of DST store keys as well. The regularity of DST allows a client to uniquely decompose an arbitrary range into a union of a minimum number of subranges that expands the range. The parallelism among the resulting subranges can be exploited using parallel DHT operations. Consequently, it can significantly reduce the query latency to, for example, that of a single DHT `get` for moderate query range with a certain small number of parallel network connections. Moreover, DST can be extended to directly support multi-dimensional range query in a straightforward way, which is in sharp contrast to other techniques that relies on linearization techniques or space-filling curves to map multi-dimensional data into one dimension [10] [11].

To the best of our knowledge, there is few work that supports cover query. The only work we are aware of is Interval Skip Graph (ISG) [8]. As shown in their work, it requires nontrivial extension to Skip Graph, which can efficiently support range query, in order to support cover query. The worst case insertion complexity of ISG is $O(n)$, where n is the node number. For queries, the ISG needs to take $O(\log n)$ steps, as inherited from Skip Graphs, to locate the first interval that matches a particular value or range, and conduct *sequential* queries for each successive intervals thereafter. DST differs from ISG in that DST is based on top of DHT and therefore enjoys all the merits of DHT such as scalability and robustness etc. Also, the data structure of DST is the highly regular segment tree while that in ISG is the random skip graph. As a result, DST can possesses excellent parallelizability while ISG has to perform query sequentially.

As will be shown in later sections, the DST can support both range query and cover query in a uniform way and equally efficient. The reason is due to the regularity of segment tree data structure, its efficiency in representing ranges and the computability of the optimal range representation.

3 Segment Tree

In this section, we first describe the one-dimensional segment tree data structure and its properties, and then extend the concept to multi-dimensional case.

3.1 Segment tree concept and properties

The basic data structure of DST, *segment tree*, comes from the Computational Geometry [12], which is essentially a full binary tree with each intermediate node represents a segment (or subrange, interval, and will be used interchangeably). It possess some excellent properties listed below. Note that, from practical interests, we only consider integers in segment tree.

1. The segment tree representing the range of length L has a height $H = \log L + 1$.

2. Each node on a segment tree represents an interval $[s_{l,k}, t_{l,k}]$, ($l \in [0, \log L]$ and $k \in [0, 2^l - 1]$). Its length is $l_{l,k} = t_{l,k} - s_{l,k} + 1$. Clearly, the root node interval equals to the segment tree range and leaf node interval is one.

3. Each non-leaf node has two children. The left child and the right child represent the intervals $[s_{l,k}, \lfloor \frac{s_{l,k}+t_{l,k}}{2} \rfloor]$ and $[\lfloor \frac{s_{l,k}+t_{l,k}}{2} \rfloor + 1, t_{l,k}]$, respectively. The union of the two children covers the same interval as the parent does.

4. For neighboring nodes on the same layer, we have $s_{l,k} = t_{l,k-1} + 1$ for all $k \in [1, 2^l - 1]$. This property ensures the continuity of the segment tree among neighboring intervals on the same layer

5. All the nodes from the same layer span the whole segment tree range. That is, $\bigcup_{k=0}^{k=2^l-1} [s_{lk}, t_{lk}] = L$ for all $l \in [0, logL]$. This property ensures the integrity of the segment tree.

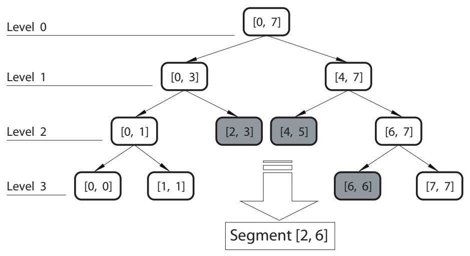

Figure 1: An exemplar segment tree with a range [0,7] and the representation of range [2,6] via a union of (minimum) three subranges in the segment tree.

An exemplar segment tree representing the range $[0,7]$ (i.e., $L=8$) is depicted in Figure 1. It is easy to verify that all above properties hold. From above properties, we can obtain the following theorems on the range representation capabilities of segment tree.

Theorem 1 *Any segment with a range $\mathcal{R}$, ($\mathcal{R} \leq L$), can be represented by a union of some intervals in the segment tree. There may exist multiple possible unions for any range with $\mathcal{R} > 2$.*

`Proof`: From the property 5, all the nodes from the same layer span the whole segment tree range, L. Since $\mathcal{R} \leq L$, it is obvious that the range $\mathcal{R}$ can be represent by $\mathcal{R}$ number of consecutive leaf nodes each of which contains a single value. This proves the first half of the theorem. According to the Drawer Principle, out of all those leaf nodes falling into any range $\mathcal{R} > 2$, we can find at least two consecutive leaf nodes that can be merged into an upper layer non-leaf node (i.e., their parent). Replacing these two leaf nodes with their parent essentially gives an alternate representation of the range $\mathcal{R}$.

Theorem 1 states that there are multiple possibilities to represent a larger range with unions of smaller subranges. Two related questions naturally arise: if or not there exists an optimal representation (i.e., minimal number of subranges involved in the representation) and, if yes, how to obtain the optimal representation? The two questions are respectively addressed by Theorem 2 and Algorithm 1 below.

Theorem 2 *The number of node intervals whose union can represent a range $\mathcal{R}$, ($\mathcal{R} \leq L$), is bounded by $2\lfloor \log R \rfloor$, and the bound is tight.*

`Proof`: It is obvious that R can be represented by $R = \sum_{i=0}^{\lfloor \log R \rfloor}(l_i + r_i)R_i$, where $l_i = \{0,1\}$, $r_i = \{0,1\}$ and $R_i = 2^i$. Since each R_i is exactly the interval represented by one node on the i-th level, then the actual node number equals to $\sum_{i=0}^{\lfloor \log R \rfloor}(l_i + r_i)$. In a worst case, we have both l_i and r_i equal to 1 for all $i = 0, 1, \cdots, \lfloor \log R \rfloor$. Therefore, the maximum number of nodes is $2\lfloor \log R \rfloor$. The tightness of the bound can be easily proved through construction of the optimal union representation of the range [1, L-1].

Theorem 2 states that any segment with a range $\mathcal{R}$, ($\mathcal{R} \leq L$), can be represented by a union of no more than $2 \log R$ node intervals, which ensures the efficiency of using segment tree to represent ranges. The segment splitting algorithm shown in Algorithm 1 provides code snippet to obtain the minimum number of node intervals whose union expands the range $[s,t]$. For instance, the segment $[2,6]$ can be minimally expanded by the union of intervals $[2,3]$, $[4,5]$ and $[6,6]$, as also shown in Figure 1. The segment splitting algorithm guarantees the computability of optimal representation which in turn removes the ambiguity for the range splitting and enables concurrent parallel operations.

3.2 Multi-dimensional segment tree

The structure and algorithm for one-dimensional segment tree described above can be extended to the multi-dimensional case in a straightforward way. In

Algorithm 1: Segment splitting algorithm

```
Procedure: SplitSegment( s, t, lower, upper, ret )
Input: s, t /*bounds of input interval to be splited*/
Input: lower, upper /*bounds of current node interval*/
Output: ret
begin
    if $s \leq lower$ && $upper \leq t$ then
        ret.add( interval( lower, upper ) )
        return
    mid ← (lower+upper)/2
    if $s \leq mid$ then
        SplitSegment( s, t, lower, mid, ret )
    if $t > mid$ then
        SplitSegment( s, t, mid+1, upper, ret )
end
```

N-dimensional (N-D) case, a 2^N-branch segment tree is used to maintain the structural information. Each node represents a N-D sub-space labeled by two N-tuples (usually the coordinates of two corners of the sub-space). The sub-space on a node is dichotomized on each of its dimensions, and hence is divided into 2^N sub-spaces represented by its children.

To split a multi-dimensional space into a union of minimum number of sub-spaces on a multi-dimensional segment tree, we can apply the segment splitting algorithm (Algorithm 1) for each dimension separately. Let vector $\vec{s_i}$ be the resulting splitted subranges for the i-th dimension. Then the result for splitting the multi-dimensional space can be computed as the cross product of all the $\vec{s_i}$ for $i = 1, 2, \cdots, N$. From Theorem 2, it is easy to derive that the number of sub-spaces that are involved in the representation of an N-D space is upper bounded by $\prod_{i=1}^{N}(2\lfloor\log R_i\rfloor)$ where R_i is the range on the i-th dimension of the space. Although it is argued in [13] that it would not be possible to adequately partition data, the upper bound actually suggests the applicability of multi-dimensional segment tree for moderate query ranges, as to be detailed in the discussion section (Section 7). Figure 2 illustrates a 2-D segment tree, where the tetrad (a, b, c, d) represents a 2-D range, $a \leq x \leq b$, and $c \leq y \leq d$. The whole key space is $(0, 7, 0, 7)$, represented by the root. A query for range $(2, 7, 0, 3)$ can divided into sub-queries for $(2, 3, 0, 1)$, $(2, 3, 2, 3)$, and $(4, 7, 0, 3)$.

4 Distributed Segment Tree

In this section, we first talk about the construction of distributed segment tree, and then discuss the duality between range and cover queries, followed by the protocols for insertion, removal, maintenance and query, for range query and cover query, respectively.

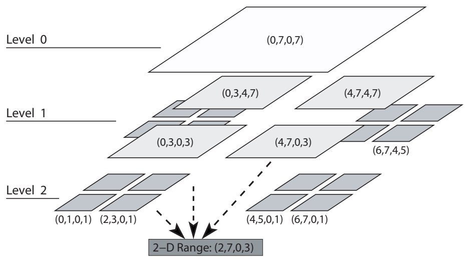

Figure 2: An exemplar two-dimensional segment tree with a space [0,7,0,7].

4.1 Mechanisms for DST construction

4.1.1 One-dimensional DST

The distributed segment tree is formed by overlaying the segment tree structure on top of a DHT. An interval $[s, t]$ is assigned to the peer associated with the key $Hash([s, t])$, and all the peers that have been assigned a node interval collaboratively form a tree with parent-children relationships being indicated by the segment tree. For instance, the peer responsible for interval [4,7] is the parent of the nodes that are responsible for interval [4,5] and [6,7], and simultaneously one child of the node responsible for interval [0.7]. This assignment implicitly reestablishes a connection between the structural information (intervals) of the segment tree and the underlying randomized (due to the hash operation) storage and routing substrate - DHT. Thanks to the computability of segment tree that an arbitrary range can be uniquely and deterministically represented by the union of a minimum number of intervals, a query (both range query and cover query) can be fulfilled at significantly reduced user observed latency by exploiting the parallelism between underlying DHT operations.

4.1.2 Multi-dimensional DST

For multi-dimensional range query and cover query, while DST can certainly be utilized in the way similar to other works that transform multi-dimensional queries into 1-D queries via linearization techniques or space-filling curves, DST can provide direct support of multi-dimensional queries by virtual of multi-dimensional segment tree. An N-D DST can be formed by distributing the N-D segment tree onto the DHT in the same way as in 1-D DST case, assigning the sub-space to the peers according to the key hashed from the two N-tuples of the sub-space. Like 1-D case, an N-D space can be uniquely decomposed a series of sub-spaces and all the operations on those resulting sub-spaces can be conducted in parallel. Note that because of the straightforward extension, the protocols and load balancing techniques talked below can be directly applied to multi-dimensional cases and will not be explicitly mentioned due to space limit.

4.1.3 Relation between DST and DHT

DST is designed to base on DHT so as to take advantage of all the merits of DHTs like self-organizing capability, scalability, robustness etc., and their simple interfaces. The key operation is the hashing of intervals and the assignment of the hashed key to responsible nodes using DHT routing mechanism (e.g., a DHT `put`).

The integration of DST with DHT can be achieved in two different fashion, depending on which level the parent-children relationship of DST is maintained. In the first tightly coupled fashion, the routing tables of DHT nodes are modified by adding extra entries containing pointers between a parent and children. However, such entries must be clearly flagged to be effective only for DST operations and not to affect the normal DHT operation. The pros of this method is the efficiency for DST operation since parent and children are always in one (overlay) hop. However, the cons is that it requires changes to all aspects DHT operations including routing and maintenance.

A loosely coupled fashion is to completely build DST on top of DHT. The parent-children relationship of DST is implicitly established by virtual of DHT routing (i.e., using DHT `get` or `put` to communicate between parent and children). The primary merit of this method is its simplicity. The relationship between parent and children can be trivially built using the simple DHT interfaces and DST can operate agnostically to the underlying DHT as long as some basic interfaces such as `get` and `put` are supported. Moreover, the node dynamics (e.g., churn) is completely handled by DHT and becomes transparent to DST.[2] However, the merit comes at the cost of the efficiency: every message is conveyed via DHT `get` or `put` operation which typically travels across multiple nodes. Note that such cost is not unique to DST. It is unavoidable for any system that are built on top of DHT. In this paper, we adopt the loosely coupled design.

4.2 Duality between range query and cover query

We mentioned above that DST can support both range query and cover query equally efficient, which is exactly one of our design target. The primary reason is the duality between range query and cover query.

From their definitions, we know range query is to find all the keys falling into a specific range, and cover query is to find all the ranges covering a specific key. Note that what being maintained by the system is keys (covered by ranges) and ranges (covering keys) for range query and cover query, respectively. They become a dual problem[3] with the logical reasoning of exchanging the roles of keys and ranges, and the insertion and query operations.

Noticing the duality between range query and cover query, one may think

[2]Our protocols do not handle node failure. Instead, we rely on the robustness of the underlying DHT.

[3]The meaning of dual problem here differs from the “duality principle” commonly referred to in optimization problems stating the solution to one problem also holds for the other.

other systems designed for range query can also be used for cover query. This is not true: it usually requires substantial modifications and it is very unlikely to support both with exactly the same architecture (more comments on this in the Discussion Section). In fact, it is the range representation capability and computability of segment tree and the structural relation cross segment tree layers make DST an effective means for supporting both range query and cover query.

4.3 Protocols for range query

4.3.1 Key Insertion and Removal

The basic operation of key insertion is to insert the given key to a specific leaf node of DST (as determined by DHT) *and* all its ancestors, because the interval of any ancestor covers also that specific key. In other words, a key should be inserted to all the nodes whose interval covers it. Thanks to the computability of segment tree, a key can be inserted to a leaf node and all its ancestors simultaneously and in parallel.

Removing a key from the DST is quite similar to the insertion process. That is, the key is removed from the leaf node and all its ancestors and can be executed in parallel.

4.3.2 Range Query

Given a range $[s, t]$ under query, the client first splits it into the union of minimum number of intervals than are maintained by element nodes in DST, using the segment splitting algorithm in Algorithm 1. It then uses DHT `get` API to retrieve the keys maintained on the corresponding DST nodes. The final query result is the union of the keys returned. This result merging process is extremely simple since all the keys retrieved from different DST nodes are unique. Moreover, all the DHT `get` operations can be called in parallel to shorten the latency. According to Theorem 2, it is usually affordable since only at most $2\lfloor \log R \rfloor$ threads is required for parallel DHT `get` invoking. Therefore, significantly reduced query latency can be guaranteed for moderate ranges under query, as demonstrated in Section 6.

4.4 Protocols for cover query

4.4.1 Range Insertion and Removal

For cover query, what need to be inserted to or removed from the system are ranges. In DST, to insert a range, a minimum number of subranges that expands the range is first obtained using the segment splitting algorithm in Algorithm 1. The range is then inserted, in parallel, to all the nodes that are responsible for those resulting subranges using DHT `put` API. Similar to the range query case, at most $2\lfloor \log R \rfloor$ threads are needed for parallel DHT `put` invoking which guarantee the short insertion delay. There is no need for a node to propagate the ranges inserted into it to its children unless some load balancing mechanism is in place to balance the load across nodes at different layers.

The process for removing a segment from DST is basically the same as that for insertion. If a parent node delegates the responsibility of maintain a certain range to its children due to the downward load stripping mechanism (see next Section), it needs to inform its children to delete the segment information.

4.4.2 Cover Query

Given a value v under query, it is obvious that the leaf node responsible for v (or segment $[v,v]$) should be queried. Since intervals maintained by all the ancestors of that leaf node also contains the value v (from property 3 of segment tree), therefore, those ancestors should be queried as well. Thanks to the computability of segment tree, those ancestors can be easily identified prior to query. Hence, cover query can also be performed by querying the responsible leaf node and all its ancestors in parallel. For a DST with height H, if H parallel threads can be executed concurrently for the query, then the query latency is just one DHT `get` operation.

In a short summary, from above protocol descriptions for range query and cover query, it is evident that DST fully manifests the duality between range query and cover query and can support both of them in a uniform and equally efficient way. These merits attribute to the regularity of segment tree and computability of the optimal representation of ranges via segment splitting algorithm (Algorithm 1).

5 Load Balancing

From the properties of the segment tree, we know that the higher level a node is, the larger range it is responsible for. As an extreme case, the root node is responsible for the whole segment tree range. While it is arguable that the storage requirement of DST is not a big concern since what being stored is only hash values, the query traffic is of more practical concern. In this section, we will present various load balancing techniques with a focus on the query traffic.

5.1 Storage balancing

The proposed key insertion protocol for range query implies that the storage requirement of nodes increases exponentially from higher layers (close to leaf) to lower layers (close to root), which is not quite acceptable. Fortunately, the third property of the segment tree indicates that the keys maintained by a parent node is redundant as they are also completely maintained by its children. The reason that a parent node maintains the keys is solely to improve the query efficiency. Therefore, to balance the load, we design a *downward load stripping* (DLS) mechanism which imposes a constraint (via a system parameter, threshold γ) on the number of keys that a non-leaf node needs to maintain. The downward load stripping mechanism works as follows: each node maintains two counters, left counter and right counter. The left counter is increased by 1 if a key put to this node can also be covered by its left child. Otherwise, the right counter is increased by 1. If a counter reaches the threshold, it triggers a saturation event. If the insertion of a key triggers either left saturation or right saturation, the key will be simply discarded.

With the DLS mechanism in place, the storage consumption can be well controlled. The drawback is in its negative impact on the query performance because queries to those saturated nodes need to be relayed to their children. The good thing is that the correctness of the query results is still ensured. The probability of saturated nodes depends on the system parameter γ. We examine different settings for γ in the experiments. We find when the γ is proportional to the interval l that a node is responsible for, i.e., $\gamma = c + k \cdot \log l$ with c, k being some constants, a good tradeoff between query performance and the storage requirement can be achieved. In this case, the unbalance between nodes at different layers is limited to a logarithmic scale.

Storage requirement is less a problem for cover query because a range is only inserted into a minimum number of nodes that are responsible for its subranges. Nevertheless, the DLS mechanism can also be applied. The system parameter, threshold γ, is set to constrain the maximum number of segments that a node can take. Different from the range query case, a single counter is enough. Whenever a range is stored onto a node, its counter will increase by 1. Once the counter reaches the threshold, it triggers a saturation event which will cause the segment to be pushed down to its children. The model we adopted is $\gamma = c + k \cdot (H - \log l)$, where H is the height of DST, and c, k and l have the same meaning as for range query case. Note but, unlike the range query case where DLS mechanism has negative impacts on the range query performance, it has no impact on (and practically helpful to) the cover query performance and only affects the segment insertion/removal process.

5.2 Query traffic balancing

The query traffic for cover query increases exponentially from higher layer nodes (closer to leaf) to lower layer ones (closer to root), because we always query the leaf node responsible for the given key and all its ancestors in parallel. This implies that the root node will be queried by all the cover queries issued to the system, which is obviously unacceptable. This is akin to the storage requirement for range query case (due to duality between cover query and range query). Also, the reward to query the root is limited when DLS mechanism in place because the root stores only $c + k$ entries of range information.

We observe that in many practical P2P applications, it is often not necessary to retrieve all the nodes whose interval cover a specific value. Instead, only a limited amount of them is enough. For example, in BitTorrent, a client typically maintains up to four active connections and a pool of few tens of backup connections to other peers. Therefore, it suffices to return only few tens records from some nodes that can fulfill the issued query. In other words, we can visit only a subset of nodes along the path from the leaf node to the root and retrieve a desirable portion of answers. This implies less traffic towards nodes in lower layers (closer to root). The key problem now becomes how to select the nodes on the path and how to balance the query traffic cross layers of DST.

In our design, each parent node periodically informs all of its children about the number of records it maintains and those of its ancestors by either through dedicated messages or piggybacking on other messages (such as heartbeat). In our experiment, we use dedicated messages whose payload is a length H (i.e., the height of the segment tree) vector, where the i-th element of the vector contains the number of records an inner i-th layer node maintains.[4] The message is periodically sent out by the root. When a non-leaf node receives the message, it fills in the number of records (i.e., intervals) it maintains and forward the updated message to its children. With this periodical advertising mechanism, the leaf nodes have the updated knowledge about the number of records stored in all the nodes along the path from itself to the root.

With the knowledge at the left nodes on the number of records currently stored in their ancestors, we are able to design a new tunable query interface – `cq(key, num)`, where a new field `num` is added to indicate the desired number of replies. The client first send the query to the leaf node. If the leaf node has enough number of records, it will fulfill the query directly. Otherwise, the leaf node has two strategies to play with, i.e., the recursive strategy and the iterative strategy. If the iterative strategy is chosen, the leaf node will return all the records it maintains and also the knowledge about its ancestors. The client will then *randomly* select some of the ancestors of the leaf node and conduct a second round query. If the recursive strategy is chosen, the leaf node itself will selected some of its ancestors and forward the cover query to them. There is no penalty in time for the cover query to be finished for the two strategies, however, the recursive strategy implies more load on the leaf nodes but can ensure a better balance among the traffic sent to its ancestors.

Finally, the query traffic is not a major problem for range query because the range under query will be split into a minimum number of subranges and only the nodes responsible for those subranges are involved in the query.

6 Evaluation

We implemented both DST and PHT [6] upon the publicly available OpenDHT service [14] and ran experiments on the PlanetLab. To ensure fair comparison, we always run the same queries simultaneously from the same host server. We also limit the number of concurrent DHT operations to 50 (i.e., at most 50 concurrent threads) for both of DST and PHT implementations to prevent the system resource from exhausting. Due to the vagaries of load on the PlanetLab, the reported results are averaged over more than 300 repeated experiments.

[4]Each element of the vector should be at least $\log \gamma$ long in bits. In our experiment, we simply set it to be a byte.

Some of comparisons have been shown in [7] such as the 1-D insertion/query latency for range query etc.

For sake of space considerations, we will more focus on the essential properties of DST and some new or updated experimental results like 2-D range query, and will echo very few 1-D experimental results here to make full story.

6.1 Range query performance

To measure the performance of range query, and to compare it with PHT, 2^{16} artificially generated keys are pre-loaded onto both DST and PHT. They are uniformly distributed over a 2^{20} key space.

6.1.1 Structural Properties

Recall that we use a threshold γ to constrain the number of keys maintained on each node. In the first experiment, we measure the number of nodes that become saturated as the keys are inserted. We use the model $\gamma = c + k \cdot \log l$, where l is the interval a node is responsible for and c, k are some constants. $k = 0$ implies a constant model.

Figure 3 shows a plot of the percentage of saturated nodes on each level of segment tree during the key insertion process. It can be found that if we make the storage constraint to increase logarithmic with the interval that a node is responsible for, the saturate ratio of nodes will be significantly reduced. This implies that significantly fewer saturated nodes would be encountered while perform range query. Notice that when a range query hits a saturated node, the query needs to be relayed to its children. Therefore, slightly increase (in logarithmic scale) the storage capacity of inner-nodes of DST, the range query performance can be greatly improved.

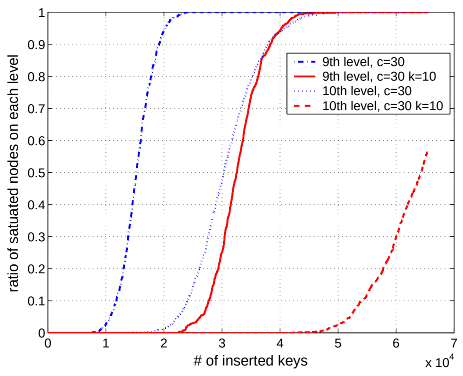

Figure 3: The percentage of nodes that are full of capacity on each level when inserting the keys onto them.

6.1.2 Query Performance

Due to space limit, we only report the experimental results for 2-D range query. Please refer to [7] for 1-D case. In this set of experiments, we generated 1200 2-D range queries. Both dimensions of each 2-D range query are uniformly distributed between $[0, 2^{10}]$. Figure 4 shows the cumulative distribution function (CDF) of the latency of all the queries. Clearly, DST significantly outperforms PHT.

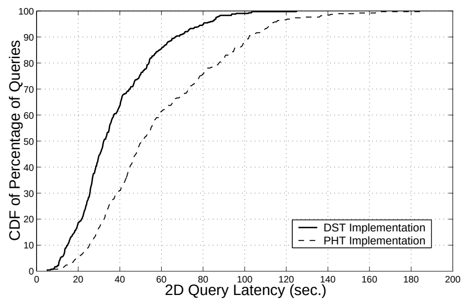

Figure 4: A cumulative distribution function(CDF) of 2D query for 500 items.

PHT performs 2-D range query by converting the 2-D query to 1-D case using space-filling curves such as Hilbert curve or z-curve. While space-filling curves are proven good at retaining the correlation between neighboring element, it is impossible to completely capture the 2-D correlation. As a result, when directly applied to range query, it will lead to many false results. The reason is that the conversion often significantly increased the actual query range, see Figure 10 for an illustration.

Define the noise ratio to be the undesired query range divided by the actual query range that is submitted to the system. Figure 5 shows the average noise ratio of 1000 random experiments, with the maximum range for each direction is set to 1024 and we measure different query areas from 1 to 2^{16}. In general, the noise ratio is very high, exceed 70% all the time. Interestingly, the noise ratio depends on the shape of the query area and square query shape seems most preferable.

6.2 Cover query performance

6.2.1 Structural Properties

In this set of experiments, we generate $10k$ segments randomly distributed in a 2^{14} key space. The span of these segments are distributed uniformly from 100 to 5000. Figure 6 shows the average number of intervals maintained by the DST nodes on each level. Indeed, the storage requirement on DST nodes are not very heavy, especially considering what being maintained are only hash values. The most loaded nodes maintains only less than 400 range information. Nevertheless, it also demonstrates the effectiveness of the downward load stripping

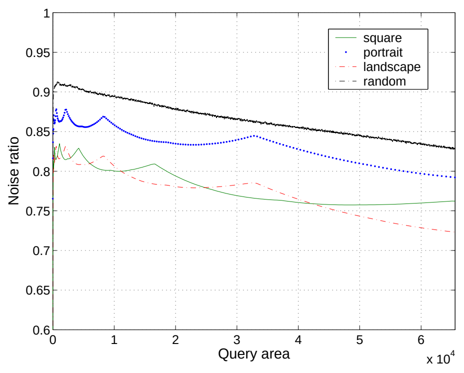

Figure 5: Noise ratio of 2-D range query due to z-curve linearization. Square, portrait and landscape means the ratio of height to width is 1, 2 and 1/2, respectively, and random means no constraint on the shape of area.

mechanism: without it, the average number of intervals maintained on the 4^{th} level nodes of DST is much higher than that on the other levels. With downward load stripping, the load is significantly smoothed over across levels of DST. We adopted the model $\gamma = c + k \cdot (H - \log l)$, where H is the height of DST, and l is the interval a node is responsible for, and c, k are some constants. $k = 0$ implies a constant model.

From the figure, we can see that downward load stripping mechanism has a profound impact on the nodes' storage requirement. If the system parameter is set to constant (80 in experiment), we can see most higher layer nodes (close to the leaf) are saturated, however, about 10% of range information are lost because when the leaf nodes themselves are saturated, they can not accept ranges pushed down from its parent. As will be shown below, this generally has no big impact (even beneficial) for practical settings when applying traffic load balancing. However, if less or no such loss is desired, then slightly increase (in logarithmic scale) of the storage capacity of lower layer nodes will greatly mitigate the loss situation.

6.2.2 Query Performance

Figure 7 shows the CDF of latency (averaged over 1000 segments) for segment insertion and cover query. The average latency are 6.169 seconds and 3.232 seconds, respectively. We notice that query latency is shorter than insertion latency. This is because the query process is almost constantly query H DST nodes along a path from root to a leaf. But the insertion process is elusive, and additional cost may be caused to handle saturated nodes, i.e., push down the inserted ranges to its children.

We designed traffic balancing mechanisms to alleviate lower layer nodes from being overwhelmed by query traffic. We observed from real P2P applications that generally only a desired amount, instead all, of results are expected when a

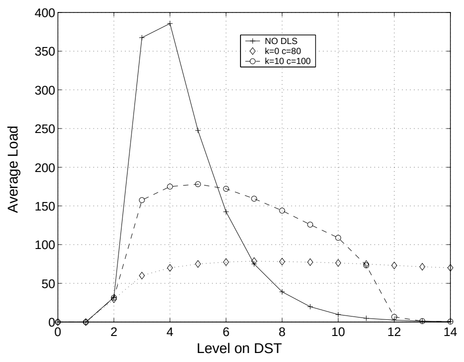

Figure 6: Average number of intervals maintained on DST nodes.

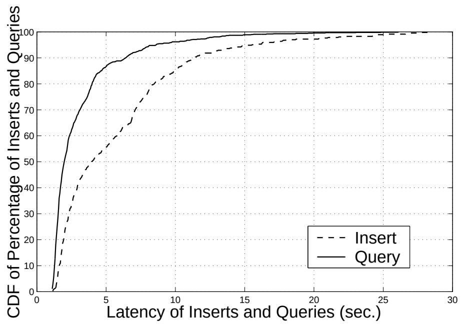

Figure 7: Cumulative distribution function(CDF) of segment insertions and cover queries for 1000 items.

peer issues a cover query. The average number of visited nodes vs the expected portion of resulting records is shown in Figure 8. We see that when the expectation is low, there is significantly savings in the number of nodes visited and, interestingly, more strict storage constraint leads to greater savings. The reason is that due to the downward load stripping, more ranges are pushed down to lower layer nodes. Since a client will contact the corresponding leaf node first, based on the reply, a client (we adopted the iterative strategy) can selectively query a few more nodes directly in a second round.

7 Discussions

7.1 To or not to base on DHT

It is argued that DHTs are inherently ill-suited to range queries because the very feature that makes for their good load balancing properties, randomized hash functions, works against range queries [5]. However, as demonstrated in [6],

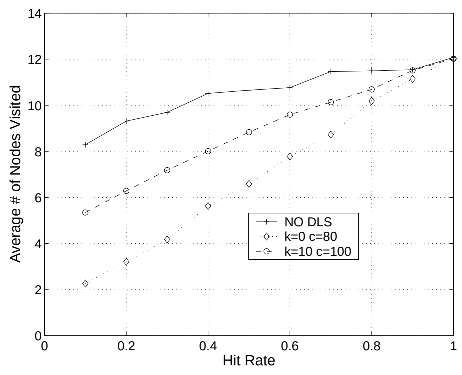

Figure 8: Percentage of retrieval vs number of nodes visited

DHTs are feasible to be used as an application-independent building block to implement other desired functionalities such as range query. Moreover, under the powerful layering design principle, the development of some applications can be significantly simplified by leveraging other already deployed DHT services like OpenDHT [14], because the applications can be essentially alleviated from various tedious overlay maintenance work and, instead, leverage the inherent characteristics such as scalability, robustness etc. of the underlying DHT. Therefore, we also based our work on DHT.

7.2 Direct or indirect support to multi-dimensional range query

There are essentially two classes of approaches to perform multi-dimensional range query, namely direct support and indirect support approaches.

Direct support approaches are to directly support multi-dimensional range query by explicit design of new architectures (e.g., Mercury [5]) or directly extend the 1-D range query solution to multi-dimension case (e.g., DST). Mercury [5]) has the merit of being able to leverage existing 1-D range query solutions, but it requires to incorporate multiple overlays and the member of each overlay needs to maintain extra links to other overlays besides to predecessors and successors in its own overlay. Also, the load balancing needs to be explicitly addressed. DST, with simple extension of 1-D solution, risks that the combinations of ranges on different dimensions may be explosive. But this problem is strongly mitigated by the segment splitting algorithm that decomposes an arbitrary range into the union of minimum number of subranges that are maintained by the segment tree. Figure 9 shows the upper bound (i.e., $2\lfloor \log R \rfloor$) and actual average number (averaged over 1000 random experiments for each range), together with the deviation, of resulting subranges when applying segment splitting algorithm (Algorithm 1) for ranges from 100 to 5000. It can be seen that in most cases, the resulting subrange number are small, which implies

DST is feasible, at least for moderate multi-dimensional range queries. Also, recall that excellent parallelism can be leveraged among the responsible nodes.

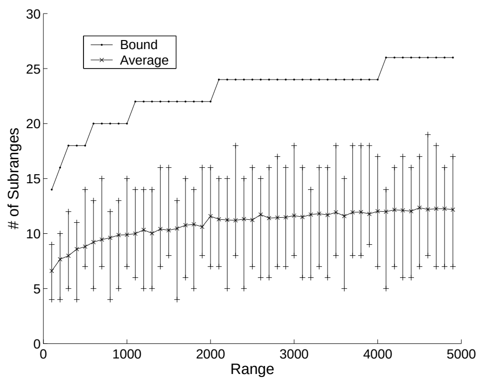

Figure 9: Average number of resulting subranges.

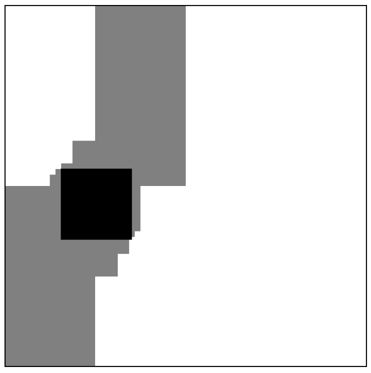

Figure 10: Illustration of noises due to z-curve linearization.

Indirect support approaches (e.g., PHT) are to apply linearization techniques or space-filling curves, and then resort to existing 1-D range query solutions. However, as shown in previous section, the query results are very noisy. Figure 10 shows one case how the actual query range (gray area) is expanded due to z-curve filling, as compared with the desired query range (black area). Even though the noises can be filtered either inside the service or at the client, an intrinsic overhead on query traffic and processing time is incurred. While it can be argued that PHT can be enhanced by replicating more data to the intermediate tree nodes, it still lacks of the computability feature of DST that decomposes an arbitrary given range into a *minimum* number of element subranges and exert the parallelism among the resulting subranges. If to avoid flooding the whole tree, PHT always need a few sequential prefix-matching steps ($\leq O(\log D)$) to locate the node that most tightly cover the range under query, starting from which parallel queries can be applied.

7.3 Support of cover query

To the best of our knowledge, there is few work that supports cover query. As pointed out in the previous section that the cover query problem is essentially a dual problem of range query. Therefore, it might be natural for one to have the impression that any work designed to support range query could be revised to support cover query fairly easily. Unfortunately, this is usually not the case.

Taking Interval Skip Graph (ISG), the only proposal we are aware of that supports cover query, as example. It is well known that Skip Graph can support range query efficiently, however, it takes nontrivial extensions to design ISG to support cover query, as demonstrated in [8]. As a first step, Skip Graphs is extended to store intervals rather than values. Nodes that storing different intervals are then sorted according to the starting point value of their intervals. Unfortunately, doing this way alone is not adequate since it only tells when to stop a query and one has to start a query always from the beginning of the ISG. To further prune away the non-necessary queries, each node of ISG is also required to store the maximum value among the intervals in all its preceding nodes. Due to such extra effort, the worst case insertion complexity of ISG is $O(n)$, where n is the node number. For queries, the ISG needs to take $O(\log n)$ steps, as inherited from Skip Graphs, to locate the first interval that matches a particular value or range, and conduct *sequential* queries for each successive intervals thereafter.

As pointed out before, the reason DST can support both range query and cover query equally efficient is due to the regularity of segment tree data structure, its efficiency in representing ranges and the computability of the optimal range representation. The other advantage of DST comes from the choice of DHT, where both ranges and values are handled equally with hashing.

7.4 Support of cover query for intervals

So far we have talked the cover query for a specific value. However, in real applications, sometimes it is desired to find cover query for intervals, i.e., to find peers whose intervals can *cover the interval (or range)* under query. Note that this is a completely different from range query. To the best of our knowledge, this problem has not been addressed.

DST can be easily extended to support this feature. Suppose we want to find the intervals currently maintained in the system that cover a certain interval S. We can first calculate an interval S' that is in the segment tree and covers S with minimum expansion. We then directly contact the node responsible for S'. If we can not get enough results from that node, we split S into the union of a minimum number of element subranges of the segment tree. Then we query the nodes that are responsible for those resulting subranges and all their ancestors, and perform a filtering process to filter out the replies that cover only a portion of S.

7.5 Potential applications

There have been enough statement on the applicable scenario of range query. Here, we would describe a few practically appealing application scenarios for cover query. All of them would benefit from DST for more efficient query.

- In the file swarming applications such as BitTorrent, a file is divided into a large number of slices. Different slices are exchanged among all peers to accelerate the downloading process. Clearly, a peer needs first to find some other peers that have that specific slice before she can download from. It is especially helpful when only few blocks need to be downloaded for which current BT protocol failed to handle efficiently and is known as the "last-block problem".

- In real streaming applications, random seek is a frequent phenomenon [15]. How well the random seek is supported is directly related to the user experience and is a highly desired feature. To apply P2P technology, peers who performed random seeks (to new positions) need first to lookup some (or all) other peers that can potentially and immediately serve them, i.e., whose cached segments of bitstream covers the desired access positions.

- Similar requirement also arises from wireless sensor networks where all the intervals of sensed data cover a specific value needs to be effectively identified [8].

While above applications require efficient cover query of a single key, there are also scenarios that require cover query over an interval or a range. For example, in many e-Business platforms and deployed P2P systems, reputation mechanisms are imposed to establish the trust among participants and to ensure the health of the systems in the long run. With a reputation system in place, a user may want to transact only with other users whose reputation is above a certain tolerable value, which corresponds to a cover query of the interval from a certain value towards infinity. In most online games, players are rated according to their skill levels. It is natural for players to find other players with similar or close levels.

8 Conclusion

In this paper, we first identified the importance of cover query and established the duality between range query and cover query. Then, we presented distributed segment tree (DST) - a layered DHT structure that embraces the segment tree concept - which can support range query and cover query in a uniform way and equally efficiently. To the best of our knowledge, this is the first piece of work that can achieve this, while other proposals designed for range query requires substantial extension in order to support cover query.

The computability feature of segment tree allows an arbitrary range to be uniquely decomposed into a union of minimum number of element subranges, which in turn enables DST to exploit the parallelism among resulting subranges. In consequence, the performance of DST is significantly enhanced by exploiting the parallelism and achieves query latency close to a single DHT `get` for moderate query range with a certain small number of parallel network connections. To address the storage and query traffic concerns, we designed effective load balancing techniques. We also extended DST to support multi-dimensional range/cover queries and conducted in-depth discussion on various system design aspects. We systematically studied various properties of DST and evaluate its real performance for 1-D/2-D range/cover queries. All the results and comparisons demonstrate the effectiveness of DST for several important metrics.

References

[1] A. R. Bharambe, C. Herley, and V. N. Padmanabhan, “Analyzing and improving a bittorrent network’s performance mechanisms,” in *IEEE Proceedings of the INFOCOM*, Barcelona, Spain, 2006.

[2] A. Gupta, D. Agrawal, and A. E. Abbadi, “Approximate range selection queries in peer-to-peer systems,” in *in Proceedings of the First Biennial Conference on innovative Data Systems Research (CIDR2003)*, Asilomar, CA, USA, 2003.

[3] X. Li, Y. J. Kim, R. Govindan, and W. Hong, “Multi-dimensional range queries in sensor networks,” in *in Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys’03)*, Los Angeles, USA, Nov. 2003, pp. 63–75.

[4] J. Aspnes and G. Shah, “Skip graphs,” in *Proc. of 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Jan. 2003.

[5] A. R. Bharambe, M. Agrawal, and S. Seshan, “Mercury: Supporting scalable multi-attribute range queries,” in *Proceedings of the ACM SIGCOMM*, Portland, USA, Sept. 2004, pp. 353–366.

[6] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, J. M. Hellerstein, and S. Shenker, “A case study in building layered dht applications,” in *Proceedings of the ACM SIGCOMM*, 2005.

[7] C. Zheng, G. Shen, S. Li, and S. Shenker, “Distributed segment tree: Support of range query and cover query over dht,” in *The 5th International Workshop on Peer-to-Peer System (IPTPS’06)*, Santa Barbara, USA, Feb. 2006.

[8] P. S. P. Desnoyers, D. Ganesan, “Tsar: A two tier sensor storage architecture using interval skip graphs,” in *The 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys’05)*, San Diego, USA, Nov. 2005.

[9] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, “Load balancing in structured p2p systems,” in *The 2nd International Workshop on Peer-to-Peer System (IPTPS’03)*, Feb. 2003.

[10] A. Andrzejak and Z. Xu, “Scalable, efficient range queries for grid information services,” in *in Proceedings of the 2nd International Conference on Peer-to-Peer Computing (P2P’02).* Washington, DC, USA: IEEE Computer Society, 2002.

[11] C. Schmidt and M. Parashar, “Enabling flexible queries with guarantees in p2p systems,” *IEEE Internet Computing*, vol. 8, no. 3, pp. 19–26, 2004.

[12] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, 1997.

[13] T. Hodes, S. Czerwinski, B. Zhao, A. Joseph, and R. Katz, “An architecture for secure wide-area service discovery,” in *Wireless Networks*, ser. 8, Mar. 2002, pp. 213–230.

[14] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, “Opendht: A public dht service and its uses,” in *Proceedings of the ACM SIGCOMM*, 2005.

[15] C. Zheng, G. .Shen, and S. Li, “Distributed prefetching scheme for random seek support in peer-to-peer streaming applications,” in *Workshop on Advances in Peer-to-Peer Multimedia Streaming, ACM Multimedia 2005*, Singapore, Nov. 2005.