Distributed Segment Tree: Support of Range Query and Cover Query over DHT

Changxi Zheng[1], Guobin Shen[1], Shipeng Li[1], Scott Shenker[2]
[1] Microsoft Research Asia, Beijing, 100080, P.R.China
[2] International Computer Science Institude, University of California, Berkeley

ABSTRACT

Range query, which is defined as to find all the keys in a certain range over the underlying P2P network, has received a lot of research attentions recently. However, *cover query*, which is to find all the ranges currently in the system that cover a given key, is rarely touched. In this paper, we first identify that cover query is a highly desired functionality by some popular P2P applications, and then propose distributed segment tree (DST), a layered DHT structure that incorporates the concept of *segment tree*. Due to the intrinsic capability of segment tree in maintaining the sturcture of ranges, DST is shown to be very efficient for supporting both range query and cover query in a uniform way. It also possesses excellent parallelizability in query operations and can achieve $O(1)$ complexity for moderate query ranges. To balance the load among DHT nodes, we design a downward load stripping mechanism that controls tradeoffs between load and performance. We implemented DST on publicly available OpenDHT service and performed extensive real experiments. All the results and comparisons demonstrate the effectiveness of DST for several important metrics.

1. INTRODUCTION

Distributed Hash Table (DHT) has drawn immense attentions in P2P research field [1] [2] [3] [4]. This is mainly due to its inherent characteristics such as scalability, self-healing and self-organizing capabilities, which are also convincingly demonstrated by the recent deployments of P2P applications where DHT is employed as an underlying infrastructure [5] [6].

As a primary design goal, most DHT-based P2P systems have achieved efficient key lookup, typically at $O(logN)$ complexity. However, the inherent *exact matching* in DHT lookup circumscribes its functionality from a panacea. For instance, *range query*, which is defined as to find all the keys in a certain range over the underlying P2P network, is difficult to achieve via DHT lookup directly, because the cryptographic hash function (such as SHA hash) strips the structural properties on keys. On the other hand, range query is highly desired in many distributed applications such as P2P database, distributed computing, and location-aware computing, as so on [7] [8]. Realizing the challenge of range query, the research community has proposed a variety of solutions that address the problem from different angles, as will be discussed in detail in Section 2.

However, to the best of our knowledge, another highly desired functionality, *cover query*, which is to find all the ranges currently in the system that cover a given key, is rarely touched. Cover query arises from a number of existing popular P2P applications:

- In P2P file swarming applications such as BitTorrent [9] and Avalanche [10], a file is divided into a large number of slices. Different slices are exchanged among peers to accelerate the downloading process. Clearly, given a slice or a range of slices to download, a peer needs to lookup some other peers who has that slice or range of slices. Note that a slice is typically represented by a range between a starting position and an ending position.
- In P2P streaming applications such as CoolStreaming [11] and oStream [12], peers typically cache the recently played portion of the bitstream in a sliding-window manner. New comers or the peers who performed random seeks to new positions need to firstly lookup some (or all) other peers that can potentially serve them, i.e., whose sliding caching window covers the desired playing position.

Cover query is actually the *dual problem* of range query: the keys inserted in range query correspond to the keys that may be queried in cover query. And in turn, the ranges inserted in cover query correspond to the ranges that may be queried in range query. Recognizing the duality, we hope to design a single structure that can support both kinds of query in a uniform way. Moreover, we want to leverage the inherent advantages of DHT to consolidate the efficiency and robustness, as many other works did.

The main contribution of this paper is the design of a distributed structure over DHT to gracefully support both range query and cover query in a uniform way. The basic idea is to distribute a *segment tree* over DHT (hence the name distributed segment tree, DST) so that the *structural information* can be retained and exploited for efficient query. More importantly, the underlying DHT lookup operations can be invoked in parallel. Therefore, both range query and cover query can be achieved at close to $O(1)$ complexity (in underlying DHT lookup operations) for moderate query ranges.

2. RELATED WORK

Due to space limitation, we only discuss some most related works that support range query over DHT and refer readers to the references therein for other range query solutions that rely on specially designed underlying structure.

Mercury [8] adopts a circular overlay (the design philosophy is similar to that of DHT except not using hash) and stores data continuously in order to support multi-attribute range query. Since it uses specially designed overlay, load

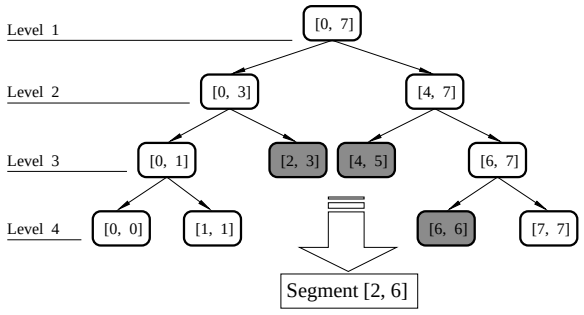


Figure 1: Illustration of a segment tree with a range [0,7] and the optimal representation of range [2,6] via three subranges.

balancing has to be explicitly considered. Skip graphs [13] is a distributed data structure that implements range search, but it requires non-trivial extensions to DHT to maintain the load balance. There is big differences between DST and Mercury and Skip graphs because DST is built on top of generic DHT and adopts highly regular data structure. In this sense, Prefix Hash Tree (PHT) [7] is most similar to our work, because both not only use DHT for traditional key-based lookup, but also impose a new data structure (specifically, trie-based structure for PHT) onto a generic DHT for richer functionalities while retaining other inherent benefits of DHT such as scalability and robustness etc. Therefore, we will mainly perform comparisons against PHT in our experiments.

However, in the trie-based structure of PHT, keys are stored *only* at the *leaf* nodes that share the same prefix and the client has no knowledge about the structure of the whole PHT. As a result, a client has to spend additional DHT `get` to reach the leaf nodes with the longest matched prefix. For a D-bit key, the complexity is $O(logD)$ using binary search and the search is intrinsically *sequential*. On the contrary, in DST, we maintain a highly regular architecture and allow intermediate nodes to store keys as well. The regularity of DST allows each client to easily calculate a union of minimum subranges that matches the query range which is key to simultaneously support both range query and cover query with a uniform structure. The client can find the responsible node of each subrange at $O(1)$ complexity (in underlying DHT `get` operation), and the queries of subranges can be executed in parallel.

As mentioned before, we are not aware any other work that supports cover query over DHT. In our previous work [14], we proposed a scheme for efficient service discovery in asynchronous P2P streaming applications. It is essentially a cover query problem and is solved using a specially designed architecture that combines tree and mesh. Although that work debuted the application of segment tree data structure for efficient range representation, the design of DST, i.e., distributing segment tree over DHT, enjoys all the desired features inherited from underlying DHT such as robustness, efficiency, scalability, etc.

3. DISTRIBUTED SEGMENT TREE

In this section, we first describe the data structure that motivates the DST, its properties that enables efficient range representations which is crucial to both range query and cover query. Then we present how to distribute such a structure over DHT.

The basic data structure of DST, *segment tree* [15], comes from the Computational Geometry and is essentially a full binary tree with the 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 (henceforth the range is called segment tree range) has a height $H = \log L + 1$.

2. Each node on a segment tree represents a *node 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 any $k \in [1, 2^l - 1]$. This property ensures the continuity of the segment tree.

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

An exemplar segment tree representing the range $[0, 7]$ (i.e., $L = 8$) is depicted in Figure 1. We can easily verify all above properties.

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

Since the segment tree is a full binary tree, it is trivial to prove the first half of the theorem. For instance, the segment $[2, 6]$ can be represented by the union of intervals $[2, 3]$, $[4, 5]$ and $[6, 6]$, as shown in Figure 1. The second half of the theorem is also evident from the third property of segment tree.

Although there are multiple possibilities to represent a larger range with unions of smaller subranges, the following theorem ensures the existence of the optimal representation.

THEOREM 2. *Any segment with a range $\mathcal{R}$, ($\mathcal{R} \leq L$), can be expanded by a union of no more than $2 \log L$ node intervals.*

`Proof`: Due to the space limitation, we only give a short intuitive proof. For a given segment S, suppose the longest part on S represented by a single node is P, then the left part to P should always be represented by the right children on segment tree, and the right part should be represented by the left children. There are at most $\log L$ consecutive left children on the tree and at most $\log L$ consecutive right children. So a segment can be represented at most $2 \log L$ nodes on the tree.

```
// Parameters:
// s,t: bounds of input segment
// lower,upper: bounds of current node interval
// ret: resulting union of node intervals

SplitSegment(s, t, lower, upper, ret)
  if s≤lower AND upper≤t then
    ret.add(interval(lower, upper));
    return;
  mid←(lower + upper) / 2;
  if s≤mid then
    SplitSegment(s, t, lower, mid, ret);
  if t>mid then
    SplitSegment(s, t, mid+1, upper, ret);
```

Table 1: Range splitting algorithm.

The code snippet shown in Table 1 shows the *range splitting algorithm* that constructs the union of minimum node intervals that expand the range $[s, t]$.

In DST, the segment tree structure is distributed onto DHT in a way similar to that adopted in PHT: the node interval $[s, t]$ is assigned to the DHT node (i.e., a peer) associated with the key $Hash([s, t])$ using the underlying DHT logic. In other words, information about any node of the segment tree can be efficiently located via a DHT lookup operation. This assignment implicitly reestablishes a connection between the structural information (node intervals) of the segment tree and the underlying structureless routing (due to the hash operation) substrate, DHT. Consequently, both range query and cover query can be achieved efficiently over DST as will be elaborated in subsequent sections.

Note that the segment tree structure and the range splitting algorithm described above can be extended to the multi-dimensional cases in a straightforward way. We use 2^N-branch segment tree to maintain the N-D structural information, instead of using space-filling curve to convert the N-D space into 1-D space as in many other works. Due to space limit, we omit the details here.

4. RANGE QUERY

Given a range $[s, t]$, range query returns all the keys that belong to that range and are currently stored on the P2P overlay. In this section, we first describe the key maintenance mechanism of DST that facilitates efficient range query, and then discuss the range query procedure over DST.

4.1 Insert and Remove

The basic operation of insertion is to insert the given key to a specific leaf node (as determined by DHT) *and* all the ancestors of that leaf node, because the node interval of any ancestor covers that specific key. In other words, a key should be inserted to all the nodes whose interval covers it.

Since the segment tree is a full binary tree, every peer on the network can reconstruct the segment tree *locally* as long as it knows the segment tree range.[1] As a result, a key can be inserted to a leaf node and all its ancestors simultaneously and in parallel. According to Theorem 2, if up to $2 \log L$ parallel threads can be executed concurrently for the insertion, then $O(1)$ complexity can be achieved by exploring the parallelism.

As we know, the node interval on segment tree increase exponentially against levels. In the extreme case, the root node is responsible for the whole segment tree range. So, with above key insertion scheme, the load on nodes across levels are quite unbalanced. On the other hand, the keys maintained by a parent node is redundant and is purely for improving the query efficiency. Therefore, to balance the load, we impose a constraint (via a system parameter, threshold γ) to limit the number of keys that a non-leaf node needs to maintain and design a *downward load stripping* mechanism to achieve this.

The downward load stripping mechanism works as follows: each node maintains two counters, *left counter* and *right counter*. The left counter is increased by one if a key put to this node can also be covered by its left child. Otherwise, the right counter is increased by one. 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 discarded. It is safe doing this way because, as aforementioned, the key maintained on the parent is redundant and is for improved query efficiency. The negative effect is that a query over the parent's interval will have to be split into two queries over the two children's intervals, which, fortunately, can be executed in parallel. Clearly, by reducing redundancy embedded in the segment tree, the downward load stripping mechanism achieves a better tradeoff between load and performance, but does not affect the correctness of the query.

Removing a key from the DST is quite similar with the insertion process. That is, the key is removed from the leaf node and all its ancestors and can be executed in parallel. The only difference is that it may draw a saturated node back to unsaturation. In the case, the node may recruit an additional key from its children. If no additional key is recruited, it then marks itself as unsaturated. The recruitment mechanism helps to improve the query efficiency but brings in some overhead. As a tradeoff, such recruitment can be performed lazily.

While the robustness of DST mainly depends on that of the underlying DHT, the redundancy of keys on multiple intermediate nodes of DST can further enhance the system robustness. Because the keys maintained on a node can be recovered from the keys on its children, the query on a node can be replaced by two parallel queries on its children if that node failed and not yet recovered.

4.2 Query

Given a range $[s, t]$ under query, the client splits the the range into a union of minimum node intervals of segment tree, using the range splitting algorithm. 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. Again, 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 $2 \log L$ threads is required for parallel `get` invocations. So as long as the span

[1]It is feasible for most applications by taking the whole range of key space (could be very large) as the segment tree range.

of queried range is moderate, $O(1)$ complexity for range query can be achieved, as demonstrated in Section 6.

Due to the downward load stripping mechanism, it may incur additional cost if some of the intermediate nodes are saturated. In this case, the client has to further retrieve the keys from the children of the saturated nodes. In the worst case, it may need up to $logL$ steps. Since the node at higher level of DST is more likely to get saturated, as a result, the longer the query range is, the more expensive the query will be. In practical cases, the query range is much shorter than the whole key space (i.e., the segment tree range). Therefore, the additional cost in practical range query is low, as also demonstrated in Section 6.1.

5. COVER QUERY

In this section, we first describe the key maintenance mechanism of DST that facilitates cover query, and then discuss the cover query over DST.

5.1 Segment Insertion/Remove

Contrary to the range query, here segments need to be inserted into or removed from the system. Simply hashing of segments and putting to DHT would lose the structural information about the segments, and hence embarrass the cover query. In DST, the segment is firstly decomposed into the union of minimum node intervals using the range splitting algorithm. Then the segment is inserted to or removed from the corresponding nodes whose interval belongs to the union. According to Theorem 2, at most $2 \log L$ nodes would get involved for any given segment. Note that, unlike the range query case where the key needs to be propagated to and stored at all the ancestors of the corresponding leaf node, it is not needed at all for cover query. Instead, proper propagation to children nodes may be needed for load-balancing considerations, as will be stated below. Finally, as in the range query case, parallel insertions/removals can be exerted to shorten the latency.

Again we use the downward load stripping mechanism to balance the load between the nodes. A system parameter, threshold γ, is set to constrain the maximum number of segments that a node can take. Different from the range query case, now a node maintains a single counter. Whenever a segment is stored onto it, the counter will increase by one. Once the counter reaches the threshold, it triggers a saturation event. The saturation event will cause the segment to be relayed to its children.

The process of removing a segment from DST is basically the same as that of insertion. However, due to the downward load stripping mechanism, it may need to delete the segment in a recursive way until it succeeds. This can be performed rather lazily since it has no impact on the search result.

5.2 Query

Due to the duality, the query process is very similar to the insertion process in range query case. From root to a leaf, there is a path on which all the nodes cover the given point.[2] That means the segments maintained on these nodes could cover the given point. Therefore, invoking DHT `get` on these nodes (in parallel to shorten the latency) could retrieve the expected segments. From the analysis in Section 3, if the maximum segment span is L, $\log L + 1$ DHT `get` threads is needed for the cover query, which cost only a little system resource in most cases.

[2]DST also supports cover query for any given segment/range as well. We omit it in this paper due to space limit.

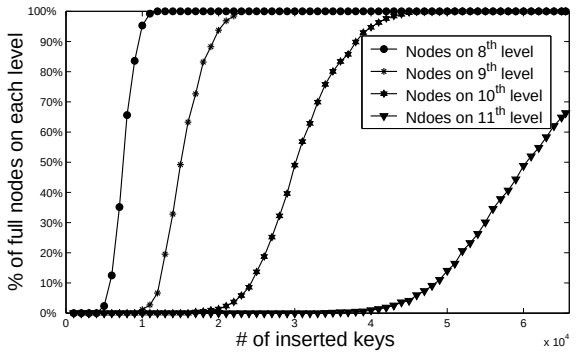


Figure 2: Percentage of nodes get saturated on different levels of DST.

6. EVALUATION

We implemented DST, in Java, upon the publicly available OpenDHT service [16] on the PlanetLab. We report some experimental results on range query and cover query in this section, together with comparisons against that of PHT when appropriate. Considering the limited space, we only report some important metrics. And due to the vagaries of load on PlanetLab upon which OpenDHT is built, the performance of a single experiment may be elusive. So we repeated each experiment more than 300 times and calculated the average results. And for fair comparison against PHT, we run the same queries using DST and PHT simultaneously from two co-located computers with the same configurations. Moreover, since both PHT and DST can utilize parallel DHT operations to shorten the latency, we limit the number of concurrent DHT operations to 50 (i.e. at most 50 concurrent threads) for both of DST and DHT implementations to prevent the system resource from exhausting.

6.1 Range Query Performance

To measure the performance of range query, and to compare it with PHT, 2^{16} artificially generated keys are preloaded 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 set $\gamma = 30$ and measure the number of nodes that become saturated as the keys are inserted.

Figure 2 shows a plot of the percentage of saturated nodes on each level of segment tree during the key insertion process. All the nodes on the 8^{th} level, where the length of node interval is 2^{13}, become saturated after $10k$ keys are inserted. However, no nodes on 12^{th} level are saturated even all 2^{16} keys are inserted. That implies that as long as the span of the query range is moderate, only a few saturated nodes would

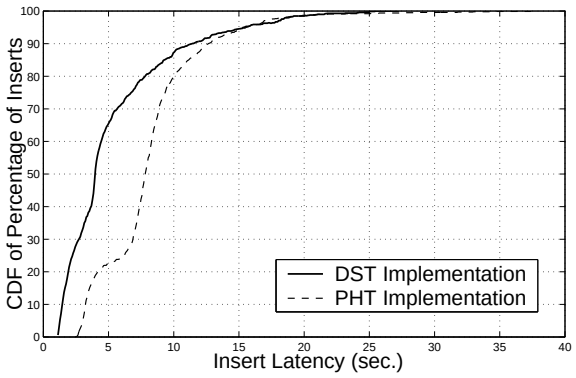


Figure 3: Cumulative distribution function (CDF) of latency for insertion of 1000 items.

be encountered while querying.

6.1.2 *Insertion performance*

Figure 3 shows the cumulative distribution function (CDF) of latency of key insertion. The initial leaf node lookup (which has to be sequential) in PHT prolongs the insertion latency, which makes PHT spend longer time than DST, which can lookup in parallel.

6.1.3 *Query performance*

In this experiment, we generated 500 range queries whose length is randomly distributed between 2^7 and 2^{16}, and calculate the average query span. We still set $\gamma = 30$ and the block size of PHT is set to 60, where the block size of PHT limits the maximum number of keys that can be maintained by a single PHT node. Clearly, a node can maintain at most 60 keys for both DST and PHT.

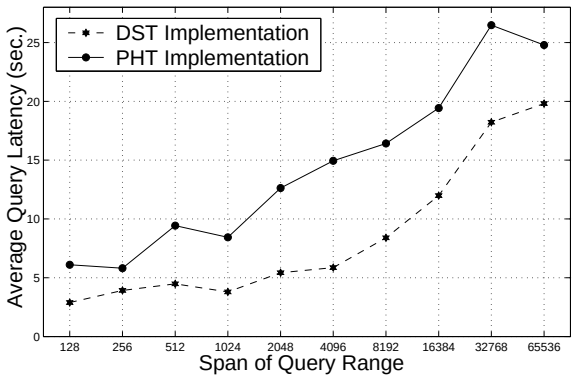
Figure 4 shows the comparison results between DST and PHT, using latency and number of DHT `get` as metrics, respectively. We can see that, when the query range is small, there are almost no saturated nodes for queries over DST. This is because the queried nodes are all on the low levels of DST, which is hard to get saturated. So, for DST, almost all the DHT `get` are called in parallel and result in almost constant latency. This is indeed the case, as can be seen from Figure 4(a). In the figure, the query latency of DST is very close to a constant when the query range is between [128, 1024]. On the contrary, PHT needs several sequential steps to lookup the leaf key. As a result, PHT spends more time for the small-span range query.

As the query span becomes larger, there are more and more saturated nodes encountered in both of DST and PHT queries. So both schemes need to spend some additional `get` to propagate the queries to the children of saturated nodes. This can be seen clearly from Figure 4(a) for the query range from 1024 up. The divergence between DST and PHT curves in Figure 4(a) also implies that more saturated nodes are encountered in PHT than in DST which caused PHT to spend even longer time in large-span range query.

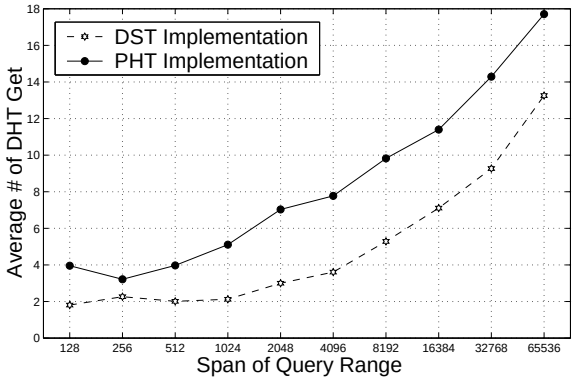
6.2 Cover Query Performance

6.2.1 *Load on DST node*

In this experiment, we generate $10k$ segments randomly



(a) Average Query Latency



(b) Average # of DHT `get`

Figure 4: Comparison of DST against PHT on query latency on different spans of query ranges: (a) The average query latency. (b) The average number of DHT `get`.

distributed in a 2^{14} key space. The span of these segments are distributed uniformly from 100 to 5000. Figure 5 shows the average number of nodes maintained on the DST nodes on each level. It also demonstrates the effectiveness of the downward load stripping mechanism on load balancing: without it, the average number of segments on the 5^{th} level of segment tree is much higher than that on the other levels. With downward load stripping, the load is significantly smoothed over across levels of segment tree.

6.2.2 *Latency*

Figure 6 shows the CDF of latency (averaged over 1000 segments) for segment insertions and cover queries. 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 querying all the H DST nodes along a path from root to a leaf. But the insertion process is elusive. Some additional cost may be imported by saturated nodes.

7. CONCLUSION

In this paper, we proposed distributed segment tree (DST), a layered DHT structure that incorporates the segment tree concept, for the purpose of efficient support of range query

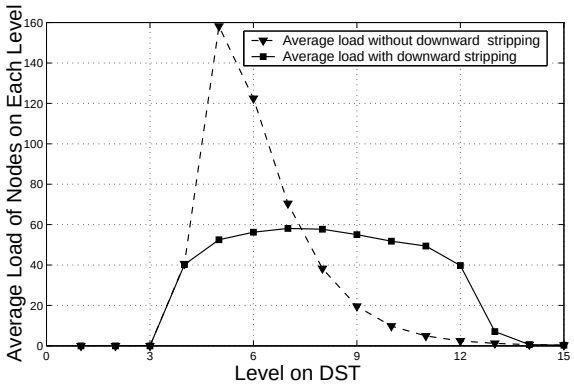


Figure 5: The average number of segments maintained on the nodes on each level of DST.

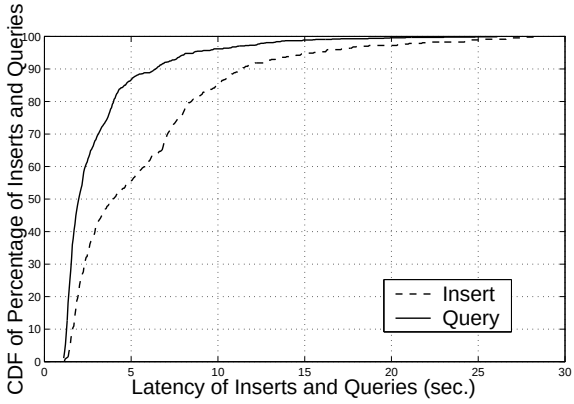


Figure 6: A cumulative distribution function (CDF) of segment insertions and cover queries for 1000 items.

and cover query. We introduced the segment tree concept and its properties as the basis of DST's excellent range query and cover query capabilities. DST essentially tolerates more redundancies to achieve efficiency. We designed a downward load stripping mechanism for load balancing purpose. It possesses excellent parallelizability in query operations and can achieve $O(1)$ complexity for moderate query ranges. Since DST is built on top of DHT, it enjoys all the inherent advantages of DHT such as scalability, robustness etc.

We implemented DST on publicly available OpenDHT service and performed extensive real experiments. All the results and comparisons demonstrate the effectiveness of DST for several important metrics.

REFERENCES

[1] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of the ACM SIGCOMM*, San Diego, CA, Aug. 2001, pp. 161–172.

[2] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A scalable Peer-To-Peer lookup service for internet applications," in *Proceedings of the ACM SIGCOMM*, 2001, pp. 149–160.

[3] P. Druschel and A. Rowstron, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms(Middleware 2001)*, Nov. 2001.

[4] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," University of California at Berkeley, Computer Science Department, Tech. Rep. UCB/CSD-01-1141, 2001.

[5] I. Stoica, D. Adkins, S. Zhaung, S. Shenker, and S. Surana, "Internet indirection infrastructure," in *Proceedings of the ACM SIGCOMM*, Pittsburgh, PA, Aug. 2002, pp. 73–86.

[6] J. Stribling, I. Councill, J. Li, M. F. Kaashoek, D. R. Karger, R. Morris, and S. Shenker, "Overcite: A cooperative digital research library," in *Workshop on Peer-to-Peer System(IPTPS 05)*, Ithaca, New York, Feb. 2005.

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

[8] 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.

[9] B. Cohen, "Incentives build robustness in bittorrent," May 2003.

[10] C. Gkantsidis and P. R. Rodriguez, "Network coding for large scale cotent distribution," in *IEEE Proceedings of the INFOCOM*, Miami, FL, USA, Mar. 2005.

[11] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "Donet/coolstreaming: A data-driven overlay network for live media streaming," Miami, FL, USA, Mar. 2005.

[12] Y. Cui, B. Li, and K. Nahrstedt, "ostream: Asynchronous streaming multicast in application-layer overlay networks," *IEEE Journal on Selected Areas in Communications*, vol. 22(1), Jan. 2004.

[13] J. Aspnes and G. Shah, "Skip graphs," in *Proc. of ACM-SIAM Symposium on Discrete Algorithms(SODA)*, 2003.

[14] C. Zheng, G. Shen, and S. Li, "Segment tree based control plane protocol for peer-to-peer on-demand streaming service discovery," in *Proc. of Visual Communication and Image Processing(VCIP)*, July 2005.

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

[16] 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.