FreeSplit: A Write-Ahead Protocol to Improve Latency in Distributed Prefix Tree Indexing Structures

Rudyar Cortés[1], Xavier Bonnaire[2], Olivier Marin[1], and Pierre Sens[1]

[1]Université Pierre et Marie Curie, CNRS
INRIA - REGAL, Paris, France
E-mail: [rudyar.cortes, olivier.marin, pierre.sens]@lip6.fr
[2]Universidad Técnica Federico Santa María, Valparaíso, Chile
E-mail: xavier.bonnaire@inf.utfsm.cl

***Abstract*—Distributed Prefix Tree indexing structures on top of peer-to-peer overlays provide a scalable solution to support range queries and proximity queries for Big Data applications. However, the latency of current maintenance protocols impacts very negatively on main operations like data insertions. This paper presents a new maintenance protocol that anticipates every data insertion on provisional child nodes. A performance evaluation conducted on the Prefix Hash Tree and FreeSplit shows that FreeSplit improves PHT by reducing the time required to perform maintenance operations. This significantly reduces the impact of the maintenance on queries, and therefore their response time.**

Keywords*—*Trie indexing, Big Data, Maintenance Protocol

I. Introduction

Extracting key information from a huge amount of data is the main challenge of Big Data [1] [2] [3]. In this context, indexing data is a major concern and represents one of the most difficult tasks [4]. Tree-based dynamic indexing over structured peer-to-peer overlays like Prefix Hash Tree (PHT) [5], LIGHT [6], Distributed Segment Tree (DST) [7], or Range Search Tree (RST) [8] supports indexing and range queries on a large scale. These solutions distribute object keys efficiently by using a decentralized tree-based indexing scheme mapped onto a Distributed Hash Table like Pastry [9] or Chord [10].

Big Data applications that provide sensing as a service [11] [4] generate massive and dynamic inputs of small data objects coming from million of users on daily basis. Applications such as digital health and personal sports training [12] [13] generate massive and dynamic inputs of data coming from millions of users on a daily basis. These applications generates a high load of insertions with no deletions.

Distributed Prefix Tree indexing structures relies on a maintenance protocol that dynamically distributes data on new trie nodes through high-latency internet links based on data distribution. Achieving the lowest latency in the maintenance of distributed trie structures is a key task due to it allows to minimise the side effect of maintenance on insertions. An insertion arriving on a leaf node performing the maintenance protocol must retry on a time Δt. It increases the query response time and the number of messages generated in the network. Moreover, concurrent and continuous inputs on a large scale generate a heavy load on trie nodes: it has an impact on the time for a node to dynamically configure to new trie nodes, and therefore a direct consequence on query response times. Therefore, massive incoming flows in trie-based indexing systems significantly deteriorate overall system performance because of the high maintenance cost they induce. Section II gives an overview of such systems, and especially the Prefix Hash Tree.

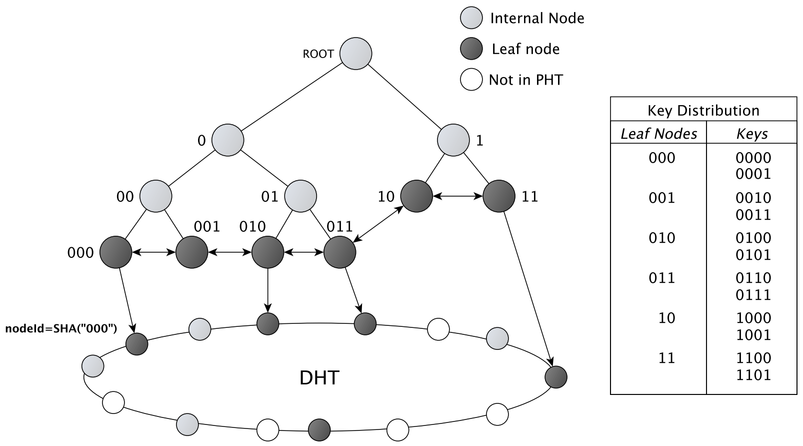
In this paper we present *FreeSplit*, a new split technique for tree indexing structures that drastically improves the performance of prefix trees when indexing massive input flows of small objects. *FreeSplit* anticipates maintenance operations by performing every insertion on the target node and on provisional child nodes. *FreeSplit* reduces the impact of *split* operations, thus improving the response time for data insertions.

The main contributions of this paper are:

- A study of the cost of index maintenance operations on a trie-based indexing system like PHT,
- An evaluation of the impact of the input load on the index maintenance cost and insertion queries,
- a new maintenance operation called *FreeSplit* which drastically reduces the index maintenance cost on distributed Prefix Tree structures.

Section III explains the new mechanisms introduced by *FreeSplit*, and how to implement our solution on top of PHT. Section IV provides a detailed performance evaluation of *FreeSplit*.

The discussion in section V shows the benefits of using *FreeSplit* in the context presented at the beginning of this introduction: massive insertions due to voluminous and continuous incoming flows of small objects. In this context, *FreeSplit* significantly outperforms PHT. Section VI presents a brief state of the art of the Tree-Based Indexing Structures, and section VII concludes and draws perspectives.



Key Distribution	
Leaf Nodes	*Keys*
000	0000 0001
001	0010 0011
010	0100 0101
011	0110 0111
10	1000 1001
11	1100 1101

Fig. 1: PHT Structure

```
Data: A key k
Result: leaf(K)
lo = 0;
hi = D;
while $lo \leq hi$ do
    $mid = (lo + hi)/2$;
    $node = DHT - lookup(P_{mid}(K))$;
    if node is a leaf node then
        return node;
    else
        if node is an internal node then
            $lo = mid + 1$;
        else
            $hi = mid - 1$;
        end
    end
end
return $failure$;
```
Algorithm 1: PHT-LOOKUP-BINARY

II. Dynamic Trie-Based Indexing Structures

Prefix trees, such as the Prefix Hash Tree (PHT) [5] and LIGHT [6], allow distributed indexing by means of recursive space partitioning. The trie dynamically evolves upon index updates: an overloaded leaf node splits the data it stores onto two new child nodes following a defined space partitioning rule.

This paper presents an enhancement that improves any trie-based indexing structure. For the sake of clarity, we will explain our solution using the Prefix Hash Tree (PHT).

PHT is a distributed indexing data structure built over a DHT [9] [10]. This structure creates data locality by using a prefix rule to map trie nodes over a DHT ring. This allows to support complex queries such as range queries, proximity queries and MIN/MAX queries.

PHT comprises three node states: *leaf nodes* store data, *internal nodes* maintain the trie structure, and *external nodes* belong to the DHT ring but not to the trie.

PHT also associates a recursively defined label with every node. The left child node inherits the label of its parent node concatenated with 0, and the right child node with 1. Data storage follows the same prefix rule: looking up an object with key k consists in finding a leaf node which label is a prefix of k. This strategy recursively divides the domain space $\{0,1\}^D$, where D is the amount of bits used to represent keys, and delivers an implicit knowledge about the location of every object in the system. This strategy recursively divides the domain space $\{0,1\}^D$, where D is the amount of bits used to represent keys, and delivers an implicit knowledge about the location of every object in the system.

PHT also maintains a *doubly-linked list* of all leaf nodes in order to provide a sequential access and therefore range query support.

The logical map between PHT nodes and DHT nodes is generated by computing the Secure Hash Function SHA-1 [14] over the PHT node label. Thus, the DHT $nodeId$ associated to a given PHT node labeled l is $nodeId = SHA(l)$.

Figure 1 shows an example of a PHT that indexes the data domain $\{0,1\}^4$ and its respective DHT overlay.

When an overloaded leaf node X reaches its maximum storage capacity $B+1$, where B is a system parameter, it carries out a *split* operation. The overloaded node configures two new child nodes and distributes B objects following the prefix rule. In order to stop data insertions the split node changes its state from *leaf node* to *internal node*, and relies on the DHT to route a *split* message to the nodes that will become its children according to the prefix rule. Every split message is sent in $O(log(N))$ hops, where N is the DHT size. The *split* message contains the number of keys to be transferred before the child can change its state from *external* to *leaf*, and the IP address of the neighbour node that is relevant to the recipient: left neighbor for the left-hand child and right neighbor for the right-hand child.

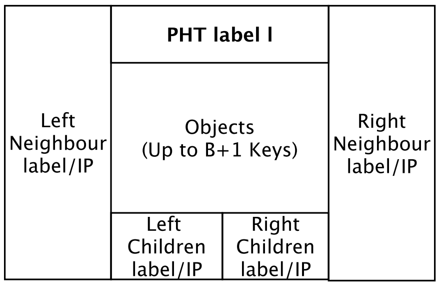
Upon reception of the *split* message, the left-hand child node returns its IP address to its new parent node and sends a message to its new sibling node in order to update the doubly-linked list of *leaf* nodes. Note that the update message to the right-hand sibling is routed via the DHT. The right-hand child does the same, but waits for the message of its left sibling to avoid DHT routing.

Once it has acquired the IP address of a child, the split node transfers all the local data blocks that match the label of the child. When this transfer is over, the child becomes a *leaf* node.

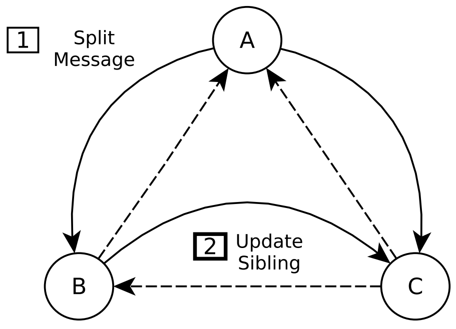
Figure 2a shows the local information on a PHT node when it carries out a split operation: its own label, the stored objects, the label and IP addresses of its neighbor leaves in the doubly-linked list until the split is effective, and the label and IP address of its children. Figure 2b shows the messages exchanged during a split operation. Dotted lines represent direct messages and solid lines represent DHT-routed messages.

Insertions are performed by using a binary lookup algorithm [5]. This algorithm receives as input a key k and returns the leaf node which label is prefix of k by sending $O(log(D))$ node state requests routed by the DHT ring, where D is the amount of bits used to represent keys. Algorithm 1 presents the binary lookup algorithm used by PHT [5]. While an internal node with label l is configuring two new leaf nodes, there are not an available leaf node associated to $l0$ and $l1$, and therefore all lookups over a key k prefix of l will return failure until the split operation finishes and child nodes become available to process requests. In this case, the client node must wait a time Δt and retry the insertion. If after n insertion failures it is not possible to index the data item, the insertion finishes with an error status, and the client must retry later. It increases the response time and generates an additional message overhead in the network.

A range query consists in contacting all leaf nodes whose label fits within the given data range. A *Sequential range query* over a data interval $I = [L, M]$ starts by locating the leaf node that indexes the lower bound L and crawls through the doubly-linked list until it reaches the leaf node that indexes the upper bound M. The crawling of the doubly-linked list can be avoided by performing a *parallel search*. A parallel range query starts by locating the node whose label shares the largest common prefix with the lower and the upper bound. This node recursively forwards the query until all leaf nodes that cover the range are reached.



(a) PHT node data structure



(b) Split protocol

Fig. 2: PHT Index Maintenance

III. FreeSplit: a Write-Ahead Protocol

This section details *FreeSplit*, our *write-ahead* protocol for distributed prefix tree structures. The goal of *FreeSplit* is to decrease the latency of split operations by using a write-ahead technique on provisional nodes upon each insertion, regardless of the split condition. Thus when the split condition occurs, the latency of the split operation is reduced to the transfer of a single object from the parent node to one of its children.

Our write-ahead protocol follows the recursive space partitioning rule and introduces a new node state called *next leaf*. Let A be a client node that indexes a given data item with key k, and let X be the leaf node whose label l is a prefix of k. The insertion of k on X is written-ahead to a provisional child node whose label starts with l and shares one additional bit with k.

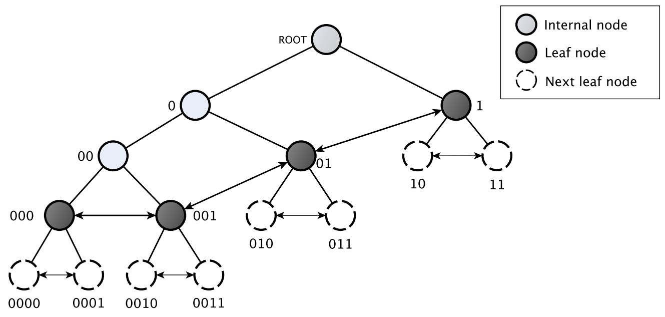


Fig. 3: Representation of the write-ahead protocol on top of the PHT structure.

Figure 3 represents the structure produced by our protocol on top of PHT. A leaf node with label l is parent to two *next leaf nodes*: the left one corresponds to label $l0$ while the right one corresponds to $l1$. A *next leaf node* stores the static and dynamic reference to its direct sibling. For example, the *next leaf node* labeled $l = 010$ stores the reference to its direct sibling node labeled $l = 011$ and vice versa.

At start-up all DHT nodes are external, they do not belong to the trie. The initialization of the *FreeSplit* trie consists in setting up its foundation: the root node and all the children associated with a single digit label (these constitute the leaf nodes at startup), as well as all next leaf nodes.

```
Data: message
if message==initPhtMessage then
    label = “root” ;
    state = “leaf” ;
    setPhtNode(label,state,null) ;
    NodeIdLeftChildren = SHA(“0”);
    NodeIdRightChildren = SHA(“1”);
    route(NodeIdLeftChildren,setNextLeafNode);
    route( NodeIdRightChildren,setNextLeafNode);
end
else if message==setNextLeafNode then
    label = setNextLeafNode.getLabel();
    parentHandle = setNextLeafNode.getAddr();
    state = “nextLeafNode”;
    setPhtNode(label,state);
    route(parentHandle,initACK);
end
else if message==updateSiblingNode then
    siblingHandle = updateSiblingNode.getAddr();
    setSiblingNode(siblingHandle);
    if message.isReply!=1 then
        message.isReply = 1;
        route(siblingHandle,updateSiblingNode);
    end
end
```

Algorithm 2: Write-Ahead PHT initialization

```
Data: message
if message.type()==DataMessage then
    data = message.getData();
    key = message.getKey();
    store(Data);
    if node.state==leaf and myLabel.length() $<$ D then
        // Extract the prefix of key of
           size myLabel.length()+1
        nextLeafLabel = extractPrefix(myLabel,key);
        nextLeafHandle = extractAddr(nextLeafLabel);
        forward(nextLeafHandle,message);
        if storage.size() == B+1 then
            // Send two direct split
               messages to two next leaf
               nodes
            route(leftChildHandle,splitMessage);
            route(rightChildHandle,splitMessage);
            setState(“internal”);
        end
    end
    else if message==splitMessage then
        setState(“leaf”);
    end
end
```

Algorithm 3: Write-Ahead indexing

Algorithm 2 shows the pseudocode of the FreeSplit init routine on top of PHT.

The first step of the trie initialization is to route a *initPhtMessage* to the $nodeId$ associated with the label of the root node: $NodeId_{root} = SHA(\text{“}root\text{”})$. When the root node receives the *initPhtMessage* it routes a *setNextLeafNode* message to both its child nodes labeled $l_0 = 0$ and $l_1 = 1$. The *setNextLeafNode* message contains the label associated to the node and all references to sibling nodes. Note that at startup all these references are null. Every next leaf node initializes its data structure and returns an *initACK* message to the root node with its IP address. Next leaf nodes also initialize partially their double linked list by exchanging an *updateSiblingNode* message with their direct sibling. For instance, the left leaf node labeled $l0$ routes its *updateSiblingNode* message to the right sibling node labeled $l1$. Upon reception the right sibling acquires the IP address of its left sibling and responds directly with its own *updateSiblingNode* message. The initialization process ends when the root node receives an *initACK* message from both its child nodes and registers their IP address.

This strategy anticipates the split operation and reduces its cost in terms of latency. Upon reaching the split condition, the overloaded node changes its state from *leaf* to *internal* and transfers the single latest object to the corresponding next leaf node. The splitting node then sends a *split* message directly to the IP address of its *next leaf* nodes to notify them that a split is under way. The reception of a *split* message on a *next leaf* node triggers a state update: the recipient becomes a *leaf* node. It then updates its doubly-linked list with the IP addresses sent in the *split* message, and routes a *setNextLeafNode* message to initialize its own *next leaf* nodes according to the prefix rule.

Given the write-ahead objective of the FreeSplit structure, indexing a given data item with key k actually involves two data insertions. Algorithm 3 gives the pseudocode of the *FreeSplit* insertion operation. The client carries out the first insertion on the leaf node L_N whose label is a prefix of k, and in turn L_N carries out the second insertion onto its next leaf node whose label is also a prefix of k.

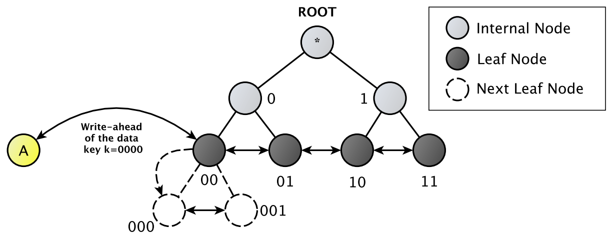


Fig. 4: Write-ahead protocol example: insertion of data key k=0000.

Figure 4 shows an example of the insertion operation. Client node A inserts an object with key $k = 0000$. First, A indexes k onto the leaf node labeled $l = 00$. Then, the leaf node copies k onto its next leaf node labeled $l_1 = 000$.

Figure 5 presents an example of a *FreeSplit* split on top of PHT. The node labeled $l = 00$ changes its state from *leaf* to *internal* and sends a *split* message to both its *next leaf* nodes (labeled $l = 000$ and $l = 001$). In turn, these nodes change their state from *next leaf* to *leaf* and send a *setNextLeafNode* message to their respective *next leaf* nodes.

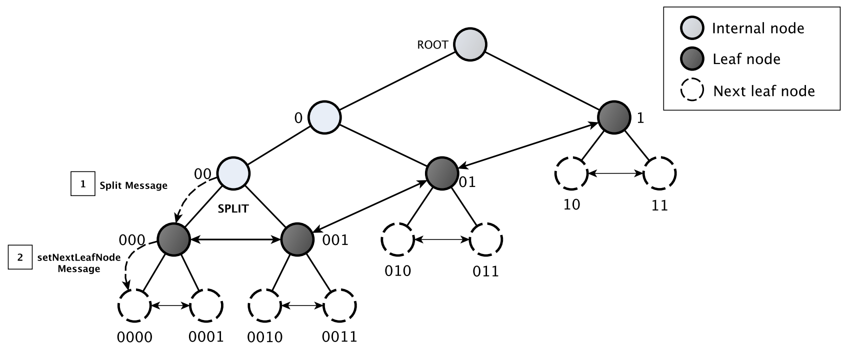


Fig. 5: Write ahead protocol example. Node labeled $l = 00$ performs a split operation

Parameter	Value
Number of nodes (N)	100
Storage capacity (B)	250 keys
Keys length (D)	32 bits

TABLE I: Trie configuration parameters

```
Data: Probability p
Result: key
for $i = 0$ to (D-1) do
    $left = random(0, 1)$;
    if $left \leq p$ then
        key.concat(0);
    else
        key.concat(1);
    end
end
```

Algorithm 4: Keys generation algorithm

IV. Evaluation

In this section we present experimental results obtained by deploying PHT and *FreeSplit* on top of FreePastry, an open-source implementation of Pastry protocol [9].

We define the *split time* as the time interval between the split condition is reached and a new leaf node is configured. We assess the *split time* and its impact over the lookup operation of our write-ahead protocol compared to PHT. We show that *FreeSplit* significantly reduces the *split time* compared with PHT.

We start comparing *split time* and then we measure the number of lookup failures induced by this maintenance cost.

The configuration parameters used to set-up PHT and FreeSplit are showed in table I. A leaf node performs a split operation when the number of objects stored is equal to $B = 250$ objects. All experiments reported in this paper were executed three times. All nodes run on a single Java VM on a processor intel core i7 2.6Ghz with 8GB of main memory, running OS X 10.9.1. PHT and *FreeSplit* were executed using java 6 SDK, version 1.6.0-65.

We measured the impact of the insertion rate and of the key distribution over the *split time*. In every experiment, one single node is chosen randomly in order to index $50,000$ keys at insertion rates of R=10, 100, 250, and 500 keys/second. We generated three key distributions with different levels of skewness following algorithm 4.

- **Uniform distribution (p=0.5)**. Distributes roughly $0.5B$ keys to each child node at every split operation.
- **Skewed distribution (p=0.3)**. Distributes roughly $0.3B$ keys to the left child node and $0.7B$ keys to the right child node.
- **Very skew distribution (p=0.1)**. Distributes roughly $0.1B$ keys to the left child node and $0.9B$ to the right child node.

A. Split time

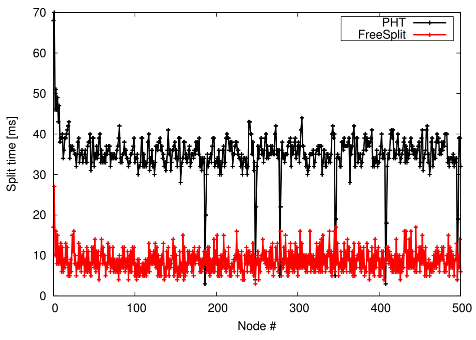
In this subsection we assess the *split time* induced by an index update in PHT compared to *FreeSplit*. We measured the impact of the insertion rate and data distribution on this time.

1) Impact of the insertion rate: Table II shows the average *split time* of PHT and *FreeSplit* (FS) respect to the data distribution and insertion rate. When the insertion rate is R = 10 keys/second *FreeSplit* presents an average *split time* 3.55 times faster than PHT (8.53 ms and 30.36 ms respectively). The *split time* presented by *FreeSplit* represents the minimal time required to configure two new child nodes in advance.

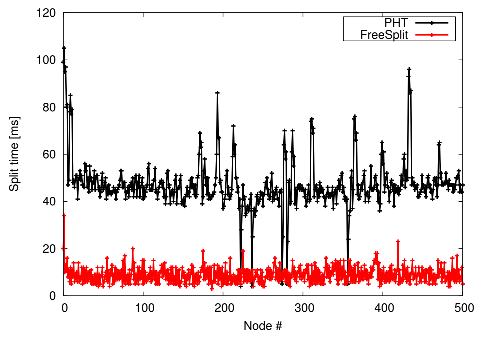
When the insertion rate increases until the storage capacity of a single leaf node (R=B=250) *FreeSplit* presents a near constant *split time* whereas it time increases faster in PHT. For instance, under this upper bound, the average *split time* presented by *FreeSplit* increases only 1.06 times compared to 1.94 times for PHT.

At insertion rates greater than the storage capacity of a single node, *FreeSplit* presents a *split time* 7.1 times faster than PHT. However, at this insertion rate leaf nodes have less time to configure *next leaf nodes* in advance. Moreover, the high frequency insertions produces a high load of lookups requests which have an impact on the *split time*. Figure 6 presents the *split time* measured by every node in a single test for different insertion rates.

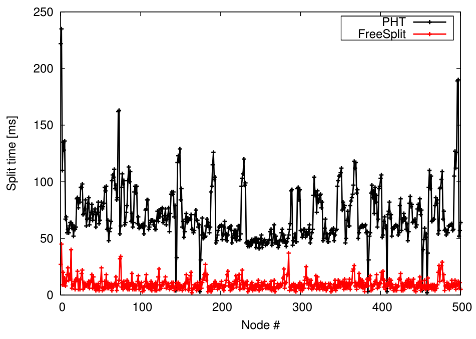
When the insertion rate increases lookups have an impact over the *split time*. It phenomena can be clearly observed in



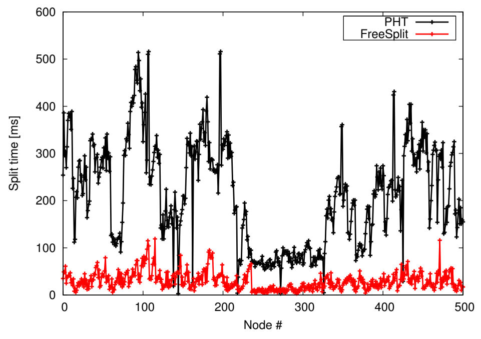
(a) R=10 keys/second



(b) R=100 keys/second



(c) R=250 keys/second



(d) R=500 keys/second

Fig. 6: Split time when indexing an uniform distribution for different insertion rates

Very Skewed (p=0.1)			**Skewed (p=0.3)**		**Uniform (p=0.5)**				
Rate keys/s	PHT [ms]	FS [ms]	PHT [ms]	FS [ms]	PHT [ms]	FS [ms]	PHT Average	FS Average	Average Speedup
10	25.49	8.22	30.41	8.59	35.18	8.78	30.36	8.53	3.55
100	34.07	8.28	40.32	8.62	45.86	8.83	40.08	8.57	4.67
250	50.59	8.72	56.35	8.78	70.48	9.84	59.14	9.11	6.48
500	163.11	26.10	139.39	16.10	221.32	31.47	174.60	24.55	7.11

TABLE II: Average split time when a node indexes $50,000$ keys

figure 6 when an uniform data distribution is indexed at different insertion rates from R = 10 to R = 500 keys/second. We chose to show this distribution because is easy to understand the effect of *FreeSplit* on the *split time* as nodes gets roughly the same number of keys, and therefore they must measure similar *split times*.

Figure 6a shows the *split time* measured by trie nodes with a low insertion load. In this case, *FreeSplit* decreases the *split time* at every split operation. Nodes at the highest trie levels present highest times because they receive more insertion load that nodes in lowest levels. When the insertion load increases to R = 500 keys/second as in figure 6d trie nodes become saturated of lookups requests and trie nodes exposes unstable *split times*. *FreeSplit* reduces the impact of this load on the *split time* because the write-ahead protocol allow nodes to copy data before nodes become overloaded.

2) Impact of the skewness: In order to assess the impact of the skewness on the *split time* we indexed three key distributions from a very skewed to uniform. Table II presents the average split time of PHT and *FreeSplit* for these distributions.

Very Skewed (p=0.1)			**Skewed (p=0.3)**		**Uniform (p=0.5)**	
Rate keys/s	PHT	FS	PHT	FS	PHT	FS
10	0	0	0	0	0	0
100	88	0	39	0	0	0
200	240	5	113	2	98	3
250	363	60	190	39	151	28
500	2586	1125	1138	690	1166	543

TABLE III: Average number of insertion fails when a node indexes $50,000$ keys

Note that *FreeSplit* gets similar average *split times* for different levels of key distribution skewness. This result shows that *FreeSplit* gets a lower *split time* than PHT under different key distributions.

B. Insertion Failure

In this subsection we analyse the impact of the *split time* in the insert operation. An insertion over a key k fails when the lookup algorithm is not able to reach an available leaf node with a label that is a prefix of k. This happens when a node is splitting into two new leaf nodes and they are not yet available to process requests (see section II for further details). We measured the number of insertion failures when a single node indexes $50,000$ keys with no retries, respect to the insertion rate and the data distribution.

Table III shows the average number of insertion failures with respect to the insertion rate for different key distributions. *FreeSplit* overcomes PHT by reducing the number of insertion failures in all cases.

1) Impact of the insertion rate: When the insertion rate increases the average number of insertion failures increases as direct consequence of the increasing *split time*. When the insertion rate is lower than the storage capacity of a single node $(R < B)$ the number of insertion failures presented by *FreeSplit* is significantly lower than PHT. For instance, when the insertion load is lower than R=100 keys/second *FreeSplit* does not presents insertion failures. Then, increasing the insertion rate to R=200 keys/second produces a maximum number of insertion failures 5 by *FreeSplit* compared to 240 in PHT.

When the insertion rate increases to $R \geq B$ *FreeSplit* the number of insertion failures increases. The high insertion rate compared to the storage capacity of trie nodes produces a high split frequency in the whole trie that increases the number of insertion failures. In this scenario, *FreeSplit* presents less insertion failures than PHT. For instance, for R=250 keys/second *FreeSplit* performs 5.39 times less insertion failures when the data distribution is uniform and 6.09 times when the data distribution is very skewed.

2) Impact of the skewness: Results presented in table III shows that the number of insertion failures increases with the data skewness. As more skewed is the data distribution, more insertions are concentrated in a single trie branch. *FreeSplit* presents less insertion failures than PHT for different levels of data skewness.

When the insertion rate is lower than the storage capacity of a single node $(R < B)$ the impact of the data skewness is minimum for *FreeSplit*. For instance, when the insertion rate is R = 200 keys/second and the data distribution is very skewed *FreeSplit* presents 5 insertion failures compared to 240 in PHT.

Then, when the insertion rate is $R \geq B$ the number of insertion failures increases in both cases. In these cases, *FreeSplit* overcomes PHT, but the its performance decreases. For instance, when a very skew distribution is indexed *FreeSplit* performs 6.05 times less insertion failures for R = 250 and 2.29 times when R increases to 500 keys/second.

C. Storage cost

FreeSplit replicates every data item twice (one in the current leaf node and another one in the next leaf node). Let n_k the number of keys indexed in a given PHT. The storage cost incurred by FreeSplit is given by.

$$C_{Storage-FreeSplit} = 2 \times n_k = 2 \times C_{Storage-PHT} \tag{1}$$

D. Split message complexity

In PHT, a leaf node reaches its split condition when it receives an insertion request and already stores B objects. The total cost combines the *node coordination* cost and the *object transfer* cost.

Equation 2 computes the message complexity of the *node coordination* for a split operation in PHT: it sums up the total cost incurred by routing the split messages and adds the update cost of the leaf links.

$$C_{coordination} = 3 \times (O(\log(N)) + 1) \tag{2}$$

The *object transfer* cost represents the number of messages required to dispatch $B + 1$ objects from the parent node to its children. Since we assume that each object is small enough to be transferred using a single TCP message, the message complexity of the *object transfer* equals the maximum number B of objects that a node can store.

Therefore, equation 3 computes the total message complexity of a split operation in PHT.

$$C_{PHT-Split} = 3 \times (O(\log(N)) + 1) + (B + 1) \tag{3}$$

Compared to PHT, FreeSplit spend only two extra direct split messages to the IP address of its *next leaf* nodes to notify them that the split operation is under way. Therefore, the number of messages incurred by FreeSplit in a split operation is given by equation 4.

$$C_{FreeSplit} = C_{PHT-Split} + 2. \tag{4}$$

E. Insertion Cost

A single data insertion generates $O(\log(D))$ DHT routing messages. Thus, the number of messages generated by a single insertion is given by equation 5.

$$C_{insertion} = O(log(D)) \times O(log(N)) \tag{5}$$

An insertion that generates a lookup failure must retry n times. Thus, the number of messages generated by n insertion tries is given by equation 6.

$$C_{insertion} = (n + 1) \times (O(log(D)) \times O(log(N))) \tag{6}$$

FreeSplit drastically reduces the number of insertion failures (see table III) and therefore the number of messages generated by insertions.

V. Discussion

A. Message Cost

FreeSplit only sends two extra direct IP messages compared to PHT in order to notify to the next leaf nodes that a new split operation is under way (see equation 4). Therefore, *FreeSplit* exhibits the same scalability as PHT.

B. Insertion rate (R) versus storage capacity (B)

FreeSplit exhibits the best performance when the insertion rate R is less or equal to the storage capacity of a single node B. *FreeSplit*, then copies the maximum number of objects on *next leaf* nodes while maintaining a minimal *split time*. Another benefit of *FreeSplit* compared to PHT is that, if we know the maximum objects insertion rate for a given application, we can choose the appropriate number of objects B per leaf node in order to maintain a minimal *split time*. This allows a proper setup of a tree-based indexing system for a specific application with massive incoming flows of data.

C. Storage space overhead

FreeSplit increases the storage space overhead by a factor of 2 (see equation 1). For applications that need to optimise the storage space, the write-ahead protocol can be initiated at a later point (i.e, when a leaf node is $\alpha\%$ of its storage capacity, where α is a system parameter). The value of α must be chosen as a commitment between the maximum insertion rate R and the storage capacity B as we discussed above. This criteria allows to reduce the storage space overhead below factor 2.

D. High Availability

Like PHT, *FreeSplit* can use replication at the DHT level in order to increase the availability of indexed objects. Replicas are usually stored on numerically close nodes in the DHT to ensure that the overlay can resist to network partitions (in the leafset for Patry or in the L successors for Chord). The same write-ahead operations are carried out within all the replicas of the leaf nodes. Managing replicas induces a higher cost for the maintenance operations. Nevertheless, transferring objects to the replicas of the provisional nodes (next leaves) can also be made in parallel using the same idea of the write-ahead technique. This makes the *FreeSplit* replication faster than that of PHT, where objects must be replicated at the time of the transfer of the $B + 1$ objects to the new leaves upon a split. In *FreeSplit*, when the next leaves become new leaves, most or all of the replicas of the objects already exist.

E. Portability

Any type of dynamic tree-based indexing structure that performs a split operation following a defined domain or data partitioning rule can be improved by implementing our write-ahead approach. For example, LIGHT [6] defines an index maintenance operation named *incremental leaf split* similar to that of PHT. Thus, our write-ahead approach can be implemented in LIGHT in order to anticipate object insertions onto next leaves.

VI. Related Work

The literature comprises many solutions to support distributed indexing and range queries on a large scale over Distributed Hash Tables [9] [10]. We classify them into two main groups: *static* tree-like indexing structures [7] [8] create a static binary tree with n levels by recursive range partitioning, and *dynamic* trie-like indexing structures [5] [6] dynamically divide the domain space.

Static tree indexing solutions suffer from a lack of load balancing and from significant message overheads because they use the higher levels of the tree (internal nodes) to perform insertions and range queries. The Distributed Segment Tree (DST) [7] and the Range Search Tree (RST) [8] are examples of such indexing structures .

The Distributed Segment Tree (DST) is a tree-like structure that indexes the data domain $R = [A, B]$ of size n by dividing it recursively into subintervals of equal size, creating a static segment tree of $L+1 = \lceil(log(n)+1)\rceil$ levels. In this structure all data is replicated onto internal nodes and requires no index maintenance operations. However, this strategy quickly induces internal nodes saturation and high tree levels can easily overload and become bottlenecks.

The Range Search Tree (RST) is an extension of DST. RST proposes a load balancing mechanism based on a *Load Balance Matrix* (LBM) and a dynamic *band* that represents an available tree zone to manage insertions and range queries. LBM performs data partitioning by distributing data onto several nodes in order to avoid node saturation, and the dynamic band distributes the query load according to statistics on insertion and query rates. However, the update of the band induces a high maintenance cost in terms of message overhead and data replication.

Dynamic prefix tree indexing structures create a dynamic trie by splitting overloaded leaf nodes according to data distribution. However, these strategies induce an index maintenance that consists in node coordination and data movement upon every index update. Examples of such structures are the Prefix Hash Tree [5] detailed in section II and LIGHT [6].

LIGHT is a dynamic tree similar to PHT. The main differences reside in the structure and the mapping of nodes onto the DHT ring. LIGHT only maps leaf nodes onto the DHT and does not maintain a doubly-linked list. This is possible because it defines a novel naming function that maps LIGHT nodes onto DHT nodes.

In order to perform index updates LIGHT defines a split operation named *incremental leaf split*. Upon every leaf split the local leaf node calculates the two next leaf labels by using the naming function, updates its label and moves the data only to one new leaf node following the prefix rule. When the index handles uniformly distributed data, LIGHT halves the data movement cost of PHT. However, LIGHT suffers from the same drawback as PHT when handling big data applications. As the number of keys stored onto each leaf node increases, the cost of index updates becomes unacceptable for many applications.

VII. Conclusion

This paper presents *FreeSplit*, a new maintenance protocol for tree-based indexing systems. *FreeSplit* outperforms traditional existing solutions over DHTs like PHT and LIGHT in terms of time efficiency. This approach decreases the response time of queries, and thus significantly impacts the overall latency of the system. *FreeSplit* is particularly well suited for handling massive insertions of small objects in the context of Big Data applications. Knowledge of the object insertion rate for a given application allows adapting the maximum number of objects stored onto a leaf node so that the maintenance operations of *FreeSplit* are always performed in the best case, and therefore remain time efficient. It can easily be ported to various existing tree-based indexing systems, as well as over different structured peer-to-peer overlays.

Future research directions include taking into account the presence of malicious nodes by using a more effective replication scheme combined with trust mechanisms. Another interesting problem is how to handle massive flows of big objects.

References

[1] S. Kaisler, F. Armour, J. Espinosa, and W. Money, “Big data: Issues and challenges moving forward,” in *2013 46th Hawaii International Conference on System Sciences (HICSS)*, pp. 995–1004, 2013.

[2] M. Wigan and R. Clarke, “Big data’s big unintended consequences,” *Computer*, vol. 46, no. 6, pp. 46–53, 2013.

[3] S. Chaudhuri, “How different is big data?,” in *2012 IEEE 28th International Conference on Data Engineering (ICDE)*, pp. 5–5, 2012.

[4] A. Zaslavsky, C. Perera, and D. Georgakopoulos, “Sensing as a service and big data,” in *Proceedings of the International Conference on Advances in Cloud Computing (ACC), Bangalore, India*, July 2012.

[5] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker, “Brief announcement: Prefix hash tree,” in *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC ’04, p. 368368, ACM, 2004.

[6] Y. Tang, S. Zhou, and J. Xu, “LIGHT: a query-efficient yet low-maintenance indexing scheme over DHTs,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 59–75, 2010.

[7] C. Zheng, G. Shen, S. Li, and S. Shenker, “Distributed segment tree: Support of range query and cover query over dht,” in *In Electronic publications of the 5th International Workshop on Peer-to-Peer Systems (IPTPS06*, 2006.

[8] J. Gao and P. Steenkiste, “An adaptive protocol for efficient support of range queries in DHT-based systems,” in *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004*, pp. 239–250, 2004.

[9] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in *Middleware 2001* (R. Guerraoui, ed.), no. 2218 in Lecture Notes in Computer Science, pp. 329–350, Springer Berlin Heidelberg, 2001.

[10] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM ’01, p. 149160, ACM, 2001.

[11] M. Chen, S. Mao, and Y. Liu, “Big data: A survey,” *Mobile Networks and Applications*, pp. 1–39, 2014.

[12] L. Ferrari and M. Mamei, “Identifying and understanding urban sport areas using nokia sports tracker,” *Pervasive and Mobile Computing*, vol. 9, no. 5, pp. 616–628, 2013.

[13] A. Clarke and R. Steele, “How personal fitness data can be re-used by smart cities,” in *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2011 Seventh International Conference on*, pp. 395–400, IEEE, 2011.

[14] D. Eastlake and P. Jones, “Us secure hash algorithm 1 (sha1),” 2001.