A Tabu-Based Cache to Improve Range Queries on Prefix Trees

Nicolas Hidalgo*, Luciana Arantes*, Pierre Sens* and Xavier Bonnaire†
*Université Pierre et Marie Curie, CNRS - INRIA - REGAL, Paris, France
E-mail: [nicolas.hidalgo, luciana.arantes, pierre.sens]@lip6.fr
†Department of Computer Science, Universidad Técnica Federico Santa María, Valparaíso, Chile
E-mail: xavier.bonnaire@inf.utfsm.cl

***Abstract*—Distributed Hash Tables (DHTs) provide the substrate to build large scale distributed applications over Peer-to-Peer networks. A major limitation of DHTs is that they only support exact-match queries. In order to offer range queries over a DHT it is necessary to build additional indexing structures. Prefix-based indexes, such as Prefix Hash Tree (PHT), are interesting approaches for building distributed indexes on top of DHTs. Nevertheless, the lookup operation of these indexes usually generates a high amount of unnecessary traffic overhead which degrades system performance by increasing response time.**

In this paper, we propose a novel distributed cache system called *Tabu Prefix Table Cache* (TPT-C), aiming at improving the performance of the Prefix-trees. We have implemented our solution over PHT, and the results confirm that our searching approach reduces up to a 70% the search latency and traffic overhead.

***Keywords*-Distributed Cache, Information Retrieval, Complex Queries, Peer-to-Peer, DHT.**

I. Introduction

Peer-to-peer (P2P) networks are now widely used to build distributed information systems. In this context, Distributed Hash Tables (DHTs) have shown to be a very efficient solution to implement large scale distributed applications. DHTs are scalable, fault tolerant, and provide load balancing. Well-known DHT-based overlays used to build distributed applications are, for instance, Pastry [17] and Chord [20].

DHTs are distributed systems based on a numerical space in which every node has a unique identifier ($nodeId$) generated using a cryptographic hash function, such as SHA-1. Each data object is associated with a key and stored in the network by mapping its key to a peer. The $lookup(key)$ operation returns the identifier of the node that stores the key. This operation gives support to hash table operations $put(key, object)$ and $get(key)$ that respectively stores and retrieves an object identifier associated to key. A significant advantage of hashing techniques used in DHTs is that they provide a uniform distribution of the objects within the numerical space. On the other hand, uniform hashing destroys data locality [21]. While this does not hinder lookup operations based on exact match queries, it precludes direct methods for the efficient support of complex queries such as range queries.

Range queries are required in a wide variety of distributed applications like music or movie storage, P2P persistent games, scientific computation, data mining, and many types of large scale distributed databases.

A range query retrieves all the objects with values within a given range. For example: *"find all the computers with memory capacity between* 1*GB and* 3*GB"* or *"find all the movies between years* 2000 *and* 2011*"*.

A solution that improves search latency over such indexes is not trivial: DHT properties should be preserved and there must be a trade-off between the provided improvement and the overhead induced by the solution. Techniques such as data replication in upper levels of the index tree are typical solutions. However, they introduce higher index maintenance costs and bottleneck issues.

In order to support range queries, many P2P data indexing solutions have been proposed. Among them, those that build an index over the DHT allow to preserve the properties of the underlying overlay, which provides the substrate for building scalable applications [12], [14], [16], [21], [22]. We discuss related work in Section II.

In this paper we focus on Prefix Trees (or trie) and present our solution in the context of Prefix Hash Tree (PHT) [16].

PHT is one of the most well-known solutions to support range queries over a DHT. It is a trie-like indexing data structure, fully distributed among the peers of the network that can easily be implemented over any DHT. Section III details PHT: its operations and its data structure.

We claim that it is possible to significantly improve the performance of the search in Prefix Tree-like structures. In this paper, we propose a new cache for Prefix-Trees which is based on the principle of the Tabu search heuristic [9]; we call it Tabu Prefix Table Cache (TPT-C). TPT-C uses a distributed shared tabu list in order to reduce the search space. By avoiding internal nodes of the trie, it provides a lightweight cache solution. These and others features of TPT-C are discussed in Section IV.

Our main contribution is a cache approach which improves the search process over prefix tree indexes. It works independently of data distribution and provides good load balancing. Extensive range query simulation experiments have shown that TPT-C search significantly outperforms the classic PHT approach: it reduces in average traffic overhead and latency by up to 70%. One of the strong features of our cache technique is that its performance does not degrade in dynamic scenarios since it does not store static references. Comparative evaluation performance results based on simulation are presented in Section V. The last section of the paper is dedicated to the conclusion of our work.

II. Related Work

In order to support range queries which preserve data locality, many P2P data indexing solutions have been proposed in the literature. They can be separated into two classes: *over-DHT* indexing class [8], [12], [14], [16], [21], [22], which indexes data over DHTs, and the *overlay-dependent* indexing class which indexes data directly on a customised overlay [2], [3], [5], [7], [10], [11]. The advantage of the first class is that it is entirely built on top of the DHT interface, and it is thus portable to any DHT. However, all the indexing data structure solutions produce overhead since it generates extra traffic due to the data index structure itself.

Overlay-dependent indexing solutions adopt either a DHT-free indexing approach which re-designs its own overlay, or a DHT-modification approach where the DHT is adapted in order to provide data locality. However, they suffers of load balancing and traffic message overhead problems because they do not maintain the DHT properties. *Prefix Hash Tree* (PHT) [16] is a well-known example of *over-DHT* indexing solution and has been widely exploited in [6], [15], [18], [19]. More details about PHT structure and operations are presented in the following section.

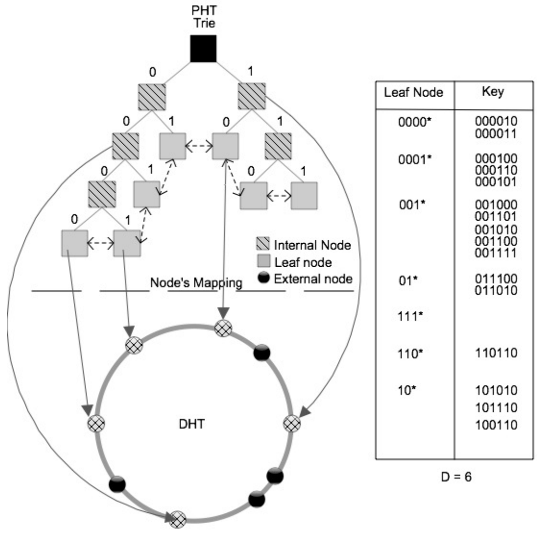
PHT main drawback relies in the top down traversal of the structure when search is performed. This index structure only store data at leaf nodes, as consequence traffic overhead and latency are increased compared with tree-linked solutions like [8], [12], [22] which replicate information at internal levels. Replication on internal level is a typical solution to improve latency and traffic overhead, however the maintenance cost of the index and load balancing still a concern for these approaches.

To improve the PHT search, Chawathe et al. have introduced in [6] a client-side cache system which stores information about leaf nodes. When a new lookup takes place, the client node cache is checked. If it contains the leaf node that stores the searched data, a DHT lookup operation is performed so as to verify that the tree structure has not changed and that the node is still a leaf. Therefore, the leaf node is located with a single lookup. Nevertheless, if the trie structure changes or the cache entry is out of date, the traditional search of PHT is carried out. In this article we denote such an approach as $PHT + Cache$. The $PHT + Cache$ approach improves the PHT performance in static environments. However, this is quite unrealistic since P2P systems are essentially dynamic environments. In this case, static references such as IP addresses may become useless, forcing a reversion to traditional search methods which generate higher latencies and message overhead. In Section V we will show that our approach outperforms this cache solution.

III. PHT: Prefix Hash Tree

Introduced in [16], the Prefix Hash Tree (PHT) is a trie-like indexing data structure over DHT-based P2P networks. It supports range queries by simply exploiting $lookup(key)$, $get(key)$, and $put(key, object)$ operations.

The indexes built over a DHT preserve the good properties of the latter such as scalability, fault resilience, and load balancing.



Leaf Node	Key
0000*	000010 000011
0001*	000100 000110 000101
001*	001000 001101 001010 001100 001111
01*	011100 011010
111*	
110*	110110
10*	101010 101110 100110

D = 6

Fig. 1. PHT structure: Mapping trie nodes over the DHT.

Data Structure

The PHT structure is a binary trie built over the data set where the left branch of a node is labeled 0 and the right branch is labeled 1. Each node n of the trie is identified with a chain of D bits or a prefix produced by the concatenation of the labels of all branches in the path from the root to n. Under the assumption that all objects can be represented by a binary key k, PHT builds a prefix tree in which objects are stored at leaf nodes. Hence, an object with key k is stored at a leaf node with a label that is a prefix of k. Each leaf node stores at most B keys.

The trie structure of PHT is completely distributed among the peers in the network. This is achieved by *hashing* the prefix labels of the PHT nodes over the underlying DHT identifier space. As a consequence, each node of the trie will have an assigned node in the DHT. Fig. 1 illustrates an example of PHT node mapping.

In PHT there are three types of peers: *leaf*, *internal*, and *external*. The first two belong to the PHT trie-structure, while the last one does not, however it participates to the DHT overlay.

In order to improve the performance of range queries, PHT maintains a double list which links all leaf nodes (dashed lines in Fig. 1).

Search: Lookup Operations

Considering a key k with a length of D bits, the *PHT-lookup* returns a unique leaf node $leaf(k)$, that stores k. The label of this node is a prefix of k. Since there are $D + 1$ different prefixes of k, there are $D + 1$ potential candidates [16].

Two different search methods can be used to locate a leaf node: *linear search* and *binary search*.

In the *linear search* method, the *PHT-lookup* starts by invoking the *DHT-lookup* operation with the shortest prefix of k. If no leaf node is reached, a new lookup operation with a one-bit longer prefix is performed. This step gets repeated until a leaf node is reached. In the worst case, the linear search executes $D+1$ lookup operations to reach the leaf node. Note that the linear search can also be performed in $parallel$, i.e., $D+1$ lookup operations are executed in parallel, one for each of the $D+1$ prefixes. However, even if this approach provides low-latency lookups, it generates a high overhead in terms of the number of messages.

A more efficient approach in terms of the number of messages is to perform a *binary search*: Binary search is a half-interval process that starts by querying a middle prefix of D. If the prefix corresponds to an internal node of the PHT, the search discards the lower half of the interval and continues querying a new middle prefix of the remaining interval. If the prefix corresponds to an external node, the search discards the upper half of the interval. This search method produces a number of lookup in the order of $\log(D)$. The drawback of the binary search is that if an internal node fails or leaves the system, it might be impossible to locate a given leaf node. Additionally, binary search cannot be performed in parallel.

Range Queries

Given two keys L and U with $U \geq L$, a range query over PHT will return all the keys within the range $[L, U]$. To perform the search, the lower bound key L is used to find the first leaf node in charge of such a key. Once this node is reached, the bidirectional links over the leaves are used to collect all the keys until the leaf node that is in charge of the upper bound key U.

Range queries can also be performed in *parallel* (PHT-Parallel). By performing a DHT lookup operation, it is possible to locate the node whose label corresponds to the greatest-common-prefix that completely covers the specified range. If the latter is an internal node, then it recursively forwards the query to those children which overlap with the specified range. This process continues until the leaf nodes overlapping the query are reached. This approach generates searches with low-latency and higher overhead in terms of the number of messages when the query span is large [16].

IV. TPT-C: Tabu Prefix Table Cache

In this section, we describe our proposal of a new cache technique for Prefix Trees indexing. This cache provides a very efficient and persistent shortcut mechanism that avoids re-visiting internal nodes of the tree. The goal of our solution is to improve search performance. We explain our solution over PHT, but other Prefix Tree solutions like P-grid [1] can also exploit our approach.

Tabu Prefixes Table Cache (TPT-C) is inspired by the *Tabu Search* optimisation method [9]. This heuristic is used to solve combinatorial optimisation problems, such as the traveling salesman problem (TSP). Tabu search maintains a *Tabu list* which stores the solutions already used and thus avoids re-visiting the same solutions. The principle of *Tabu Search* is to reduce the search space in order to speed up the process of finding an optimal solution. Like *Tabu Search*, the idea of TPT-C is to reduce the search space in the trie, thus improving latency and reducing both the message traffic overhead and the load at internal levels of the trie. Note that TPT-C is not a heuristic like Tabu search, however it exploits the principle of the tabu list.

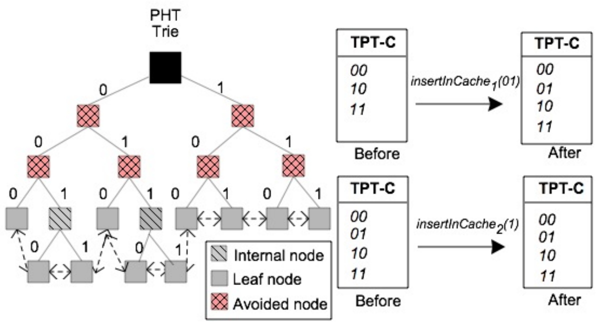


Fig. 2. TPT-C insert operation

TPT-C is a flexible and distributed cache solution which provides knowledge about the trie shape. It stores prefix labels of nodes visited in previous searches. Thus, in a search operation, cache information of every visited node is exploited in a distributed manner. Furthermore, since cached information is related with the logical structure of the index which is independent of the DHT, churn on the overlay hardly affects the performance of TPT-C.

A. *Insert in Cache*

Each entry in TPT-C stores a prefix label p, which corresponds to an internal node of the PHT. The number of entries per node is limited to ϵ. When the number of entries reaches ϵ, traditional strategies can be used for entry replacement: *First in First Out* (FIFO), *Least Recently Used* (LRU) or *Least Frequently Used* (LFU).

Whenever a node wants to insert a prefix in its local cache, it calls the *InsertInCache* function. The prefix will be inserted in the node cache if it is not already stored and it is not *redundant*.

Algorithm 1: Insert Operation in Cache (InsertInCache)

```
input : Entry $e$
$strategy \in$ {FIFO, LRU, LFU};
Boolean $insert\_flag$ = true;
foreach Entry $f$ in TPT-C do
    /* check if $e$ is not redundant */
    if commonPrefixLength $(e, f) \geq e.size$ then
        $insert\_flag$ = false;
if $insert\_flag$ then
    foreach Entry $f$ in TPT-C do
        /* remove redundant entries after the insert */
        if commonPrefixLength $(e, f)$ = $f.size$ then
            removeEntry ($f$);
        insert ($e$);
if TPT-C.size $> \epsilon$ then
    removeEntry ($strategy$);
```

The information about the prefix related to an internal node is *redundant* if it has a child already present in the node cache. The idea is to preserve, whenever possible, the longest prefixes in the cache and remove the unnecessary entries. The *InsertInCache* function is described in Algorithm 1.

Fig. 2 shows two examples of an entry insertion in TPT-C. When the $InsertInCache_1(01)$ is performed, the prefix entry

01 is inserted because it is neither present nor redundant in cache. On the other hand, in $InsertInCache_2(1)$ call, an entry with label 1 is not included in the cache because it is *redundant*.

B. TPT-C Search Algorithm

In original PHT, when a PHT-lookup of a key k is performed, the querying node Q_n iteratively generates a lookup operation whose key $P_i(k)$, at each step i, is a prefix of length i according to the selected search method (linear or binary).

Algorithm 2: Check in Cache (CheckInCache)

```
input : Key $k$, int $prefixLength$
output: Entry $hit$
int $gcp$ = $prefixLength$;
/* gcp denotes the greatest common prefix length */
Entry $hit$ = $null$;
foreach Entry $e$ in TPT-C do
    if commonPrefixLength $(e, k) > gcp$ then
        $gcp$ = commonPrefixLength $(e, k)$;
        $hit = P_{gcp}(k)$;
        /* $P_i(k)$ denotes the prefix label of length $i$ of
           the key $k$ */
```

In TPT-C, when Q_n starts the PHT-lookup, it first checks its own cache in order to find an entry which can reduce the search space.

Algorithm 3: New Query at Q_n

```
input : Key $k$
output: Message m
$prefixLength = 0$;
Entry $e$ = checkInCache $(k, prefixLength)$;
if $e$ = $null$ then
    $m$ = lookup (nextPrefix $(k, prefixLength)$);
else
    $m$ = lookup (nextPrefix $(k, e.prefixLength)$);
```

The $CheckInCache$ function is presented in Algorithm 2. It iteratively searches the cache in order to find the most useful cached information, i.e., the entry which has the *greatest common prefix length* (gcp) with k. Notice that in order to go on with the search, the size of the entry prefix must be longer than the prefix length of the current lookup operation. If an entry is found (cache hit), it can be used to continue the search, otherwise, the function returns *null* and the search reverts to a normal PHT search.

Algorithm 3 shows how a query with a key k is managed by a node Q_n. When a local cache entry hit takes place, the node performs a new $lookup$ operation starting from one step further (`nextPrefix`) than the prefix corresponding to this cache hit. This shortcut allows to go directly to lower levels of the index. If no entry is found, the lookup operation is performed as usual. The next prefix used by the lookup (`nextPrefix`) is based on the linear search method, i.e., a one bit longer prefix.

Algorithm 4 presents the reception of the Q_n lookup operation by node R_n. The reply sent by R_n includes the type of the node (leaf, internal, or external). Furthermore, if R_n is an internal node, it checks its local cache to find if there is an entry which can reduce the search space. If an entry e is found, R_n also includes e in the reply message to Q_n.

Algorithm 4: Lookup Processing at R_n

```
input : Key $k$, int $prefixLength$
output: Message m
if $myNodeType$ = $'leaf'$ then
    lookupReply ('leaf', $null$);
else
    Entry $c$ = checkInCache $(k, prefixLength)$;
    if $c$ != $null$ then
        $m$ =lookupReply $(myNodeType, c)$;
    else
        $m$ = lookupReply $(myNodeType, null)$;
```

Algorithm 5: Lookup-Reply Processing at Q_n

```
input : Key $k$, int $prefixLength$, NodeType $type$, CacheEntry $c$
Output: Message m
if $type$ = $'leaf'$ then
    $m$ = getData ();
else
    if $c$ != $null$ then
        insertInCache $(c)$;
        $m$ = lookup (nextPrefix $(k, c.prefixLength)$);
    else
        $m$ = lookup (nextPrefix $(k, prefixLength)$);
        if $type$ = $'internal'$ then
            insertInCache $(P_{prefixLength}(k))$;
```

The reception of a lookup-reply message by node Q_n is described by Algorithm 5. Upon reception, Q_n executes one of the following actions: if the reply is from a leaf node, Q_n knows that the search has ended and gets the searched data by contacting node R_n; if the lookup-reply message includes a cache entry, the latter is inserted in the cache of Q_n and the lookup continues one step further; otherwise, the search continues as usual. Note that if the reply is from an internal node, the searched prefix is also stored in the cache of Q_n.

Algorithm 6: TPT-C Parallel Search

```
input: Key $prefix$, Query-Range $r$
List $newTarget$ = $null$;
/* add both children to $newTarget$ */
$newTarget \leftarrow prefix.append(0)$;
$newTarget \leftarrow prefix.append(1)$;
/* optimise search with cache entries */
foreach cache-entry $c \in r$ do
    for $i = 0$ to $i = c.lenght$ do
        foreach Entry $p \in newTarget$ do
            if $p = c.subPrefix(0, i)$ then
                /* $c.subPrefix(0, i)$ delivers a sub-string
                   of size $i$ of $c$ */
                $newTarget.remove(p)$;
                $newTarget \leftarrow p.append(0)$;
                $newTarget \leftarrow p.append(1)$;
/* performing parallel lookups */
foreach Key $k \in newTarget$ do
    if $k \in r$ then
        lookup $(k)$;
```

C. TPT-C Parallel Range Query Search

In order to provide low-latency lookups TPT-C can be performed in *parallel*. Given a range query r, TPT-C Parallel will reach all the leaf nodes within r. Its principle is similar to that of the TPT-C key search algorithm presented above: it avoids unnecessary $lookup$ operations over internal node prefixes. TPT-C Parallel exploits the information kept in the node cache to identify all those prefixes related to internal nodes that cover r. Contrarily to PHT Parallel which visits all the internal and leaf nodes contained in the specified range r, TPT-C Parallel exploits cache information in order to avoid contacting internal nodes unnecessarily. Such an approach reduces response latency and message traffic compared to original PHT Parallel. Algorithm 6 describes the TPT-C Parallel search executed at each visited node. The algorithm receives the prefix of the current node and the range of the query. At the end of the algorithm, $newTarget$ contains a list of node prefixes to be contacted. The internal nodes that are avoided correspond to those entries in cache which are within range. The corresponding lookup operations are therefore performed in parallel.

V. Performance Evaluation

A. Experiments Setup

To conduct our experiments, we used an event-driven simulation over Peersim [13] with Pastry as overlay. We implemented a PHT layer on top of Pastry as well as both cache approaches: TPT-C and *PHT+Cache* (see section II).

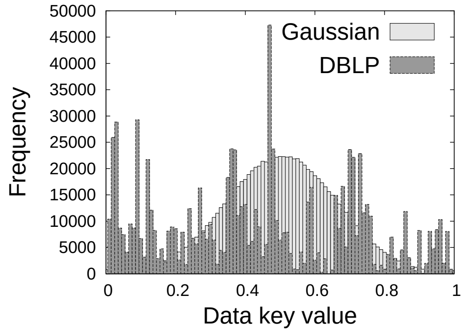


Fig. 3. Gaussian and DBLP data distribution

We considered a network of $10,000$ peers, where a PHT node stores at most 1,000 objects ($B = 1,000$). $500,000$ objects were inserted in the system. Queries are generated using both synthetic and real data sets: *Uniform*, *Gaussian*, and the DBLP Computer Science Bibliography [4] dataset. The indexed attribute in the DBLP dataset is the author name which has been converted to a floating number in the range $[0,1]$. For the Uniform dataset, keys were generated so as to be uniformly distributed in the range $[0,1]$. In the case of the Gaussian dataset, keys follow a gaussian distribution with a mean $\mu = \frac{1}{2}$ and standard deviation $\sigma = \frac{1}{6}$. Note that in the *Gaussian* distribution almost 98% of the data keys are within the range $[0,1]$. Fig. 3 presents the *Gaussian* and the DBLP data key distribution. The number of cache entries was fixed to 100 entries and the *Least Recently Used* (LRU) replacement strategy is used as cache policy.

All our experiments comprise a total of $1,000,000$ queries generated using the three above-mentioned distributions. A snapshot of the system is taken every $100,000$ queries.

In the following section, we present our evaluation performance results related to query latency and message traffic overhead as well as load balancing among the trie nodes. We also discuss the impact of the number of cache entries, data distribution, range query size, and churn on the performance of the different solutions.

B. Message Traffic Overhead and Query Latency

In order to evaluate the performance of TPT-C compared to the original PHT and *PHT+Cache*, we measured the query latency and message traffic overhead in the system. Latency is measured by the number of lookup operations necessary to reach the corresponding leaf node. Each simulation starts by inserting $500,000$ objects in the network.

Table I compares the message traffic generated by PHT Linear, PHT Binary, PHT+Cache Linear, and TPT-C Linear searches. The percentage of improvent was calculated in relation to PHT Linear: for example *% PHT+Cache Gain* reflects the percentage of traffic reduction obtained by using *PHT+Cache* compared to PHT Linear. We can observe that TPT-C strongly reduces the total message traffic. Note that TPT-C induces a reduction of more than 67% of the total message traffic in the case of a Uniform distribution. In distributions similar to that of DBLP, the reduction exceeds 73%. On the other hand, *PHT+Cache* reduces the total message traffic by only 15% and 20% when the data keys follow a Uniform and DBLP-like distribution respectively.

A second interesting result can be deduced from Table I. Our approach "artificially decreases" the height of the trie by avoiding internal nodes. The practical consequence is that the height of the trie no longer has an impact on the message traffic, similarly to PHT and PHT+Cache. Our solution thus produces approximately the same amount of messages independently of the height of the trie.

TABLE I
Number of Traffic Messages

	Uniform	Gaussian	DBLP
PHT Linear	45,119,200	49,641,018	58,929,556
PHT + Cache	38,211,964	43,519,742	46,728,556
PHT Binary	12,856,326	20,853,420	21,985,418
TPT-C	14,563,486	16,455,180	15,452,542
% PHT + Cache Gain	15.30	12.33	20.70
% PHT Binary Gain	71.50	57.99	62.69
% TPT-C Gain	67.72	66.85	73.77

As mentioned in Section III, binary search is an effective search method but, in dynamic scenarios, it can present some drawbacks. In PHT, *binary search* may fail to locate leaf nodes. Since both PHT insert and delete operations produce trie-shape modifications, wrong answers about the state of the nodes in the trie can compromise the search. As binary search does not go backwards, the search might finish without result. Such a behaviour introduces extra traffic overhead and latency, because a new search must then be performed, using the same method or the linear search. Table I also shows the total message traffic when a binary search is used. Compared to PHT Linear, the total traffic of Uniform and DBLP datasets are respectively reduced by 71% and 63%. However, binary search performance is not constant as we can see in Fig. 4(l). It can produce different performance results, depending on the indexed data distribution and the maximum key prefix.

In order to evaluate latency, we measured the number of lookup operations necessary to reach a leaf node. We compared the results of TPT-C with PHT Linear, *PHT+Cache*, and PHT Binary. Fig. 4(a), 4(b), and 4(c) show the numbers of lookup operations performed for all three distributions when the cumulative number of queries grows until $1,000,000$. As can be observed, TPT-C effectively reduces latency, offering better performance than the search techniques we compared it to. In Fig. 4(a), the performance of TPT-C and PHT Binary searches are very similar. In the case of Gaussian and DBLP distributions (Fig. 4(b) and 4(c)), TPT-C search performance is 25% better than binary search due to the irregular performance of the latter.

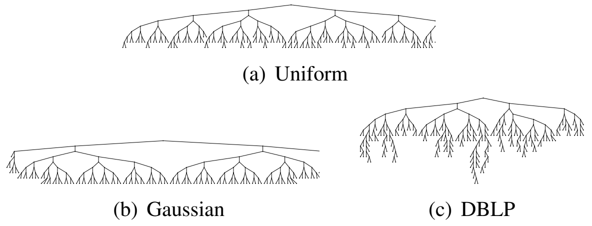
The results obtained are quite promising since TPT-C shows better performances than PHT Binary and *PHT+Cache*. Furthermore, using the real dataset of DBLP, TPT-C has an average of 3.15 lookup operations, reaching 73% better performance than Linear PHT and 67% better performance than *PHT+Cache*. These results are also valid for the two other distributions. It is worth emphasising that the use of TPT-C is more attractive than binary search since the former effectively supports dynamic conditions whereas the latter does not. For instance, binary search may fail because of churn; moreover, insertion and delete operations can also compromise the result of the binary search.

Another interesting property of TPT-C is that it gathers cache information faster than cache techniques which collect information only about the leaf nodes of the trie (e.g. *PHT+Cache*). During every query operation, TPT-C can collect a higher amount of information about internal nodes, while *PHT+Cache* only gets one cache entry per lookup. This explains why better performance is attained faster with TPT-C search than with other cache search techniques.

The difference between Uniform and skewed distributions (e.g. Gaussian, DBLP) is the height of the trie. In the case of Uniform distribution, the trie tends to grow horizontally, while in skewed distributions the growth is vertical. Since data insertions make the trie structure grow, we also compared the impact of data size on the search processes with respect to different distributions. Fig. 4(l) and 4(m) show the results with Uniform and DBLP distributions respectively. In the case of Uniform distribution, the cost of a PHT Linear search increases faster than TPT-C. On the other hand, PHT Binary exhibits a fluctuating but overall good performance. The fluctuation of the curve can be explained since the number of iterations to find a leaf node can vary depending on the size of the searched branch of the trie and the maximum key prefix size.

C. Load Balance

One of the typical problems of tree-like data structures is that a top-down traversal methods is the standard search. Hence, upper levels nodes, close to the root, have to deal with higher traffic than lower levels nodes. Notice that at level e of the trie there are 2^e nodes. The worst case is the first level, where there are only 2^1 nodes. On the other hand, at lower levels the messages are distributed among a higher number of nodes. Linear search performs iteratively by trying a greater prefix at every step until a leaf node is reached. This search overloads the upper level internal nodes, that can thus become a bottleneck.


(a) Uniform

(b) Gaussian (c) DBLP

Fig. 5. PHT generated with Uniform, Gaussian, and DBLP datasets.

TPT-C improves load balancing associated to linear search since it diverts queries to the lower levels of the trie, closer to the leaf nodes. Therefore, traffic is distributed among a higher number of nodes when compared to upper levels ones. The load per level in the trie structure is presented in Fig. 4(d), 4(e) and 4(f), where the horizontal axis represents the levels of the trie (root = level 0, i.e., the highest level).

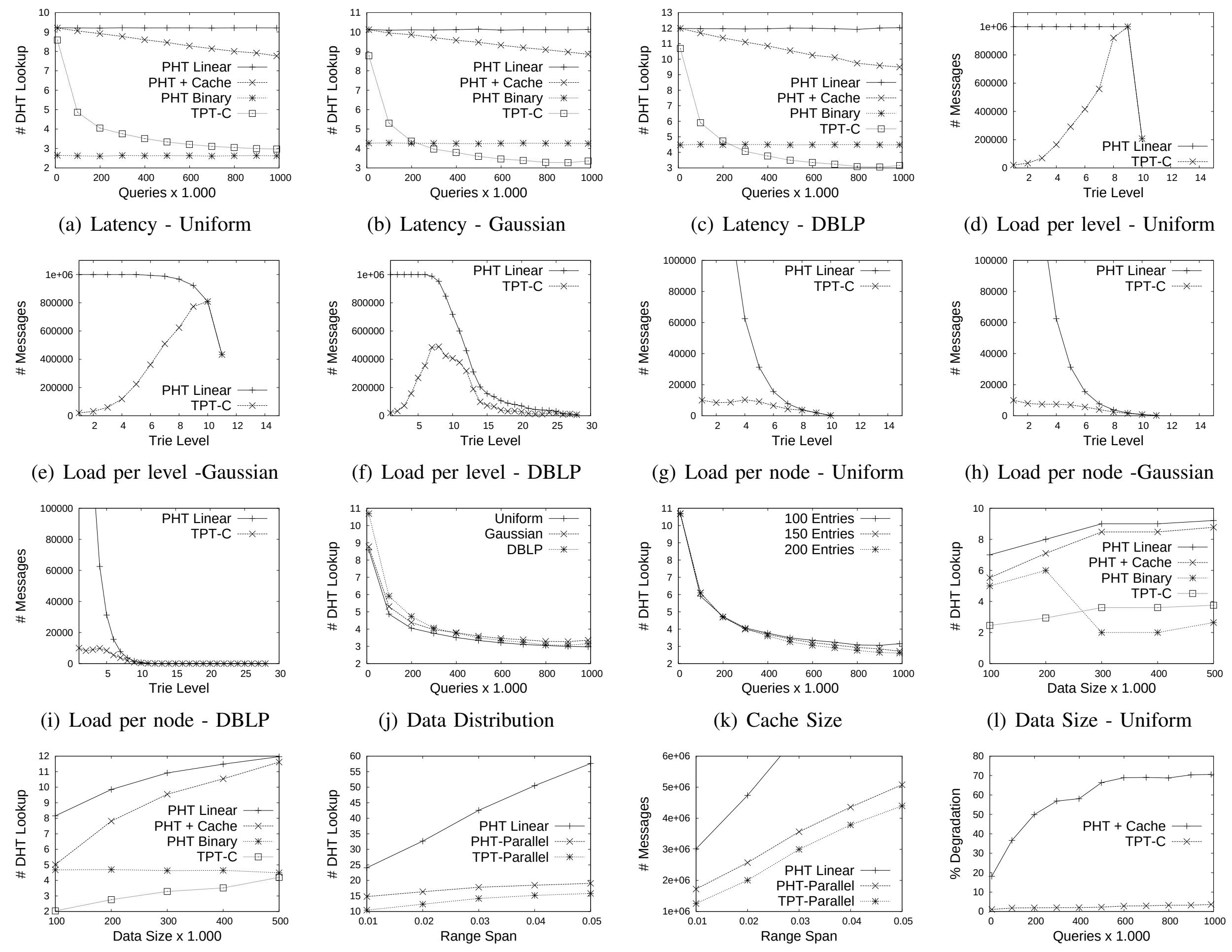
PHT Linear search entails the same load at every internal level of the trie. Since there are less nodes at higher levels than there are at lower ones, their load can become critical. TPT-C avoids bottlenecks by distributing messages at the leaf level. It spreads queries directly at the lower levels of the trie, hence avoiding lookup operations among higher level nodes. Fig. 4(f) shows how the load is distributed among the levels of the trie in the case of the DBLP distribution for a total of $1,000,000$ queries. The lowest levels of the trie have very similar loads, but PHT Linear search introduces much more load on the first 5 levels. This has a significant impact on the average load per node as is presented in Fig. 4(g), 4(h) and 4(i). We can observe that the first levels incur a huge load of messages in the case of PHT Linear search, with the risk of causing a serious bottleneck jam.

D. Data Distribution

PHT search effectiveness is very sensitive to data distributions. Since the latter modifies the trie shape, it has an impact over the lookup performance. Fig. 5 illustrates the influence of data distribution over trie shape. For instance, in clustered datasets (a high density of objects within a given range of keys), data is placed at lower levels of the trie. Therefore, querying such data will require a higher number of lookup operations.

Uniform distributions of keys generate balanced tries in which leaf nodes are located on a similar level. The number of lookup operations required to find a leaf node is almost constant. On the other hand, skewed distributions generate unbalanced tries. Consequently, in order to locate the clustered data, a higher number of lookup operations is necessary.

We evaluated the impact of *Uniform*, *Gaussian* and *DBLP* dataset distributions in the performance of TPT-C. Fig. 4(j)



(a) Latency - Uniform (b) Latency - Gaussian (c) Latency - DBLP (d) Load per level - Uniform

(e) Load per level -Gaussian (f) Load per level - DBLP (g) Load per node - Uniform (h) Load per node -Gaussian

(i) Load per node - DBLP (j) Data Distribution (k) Cache Size (l) Data Size - Uniform

(m) Data Size - DBLP (n) Range - Latency (o) Range - Traffic (p) Churn

Fig. 4. Performance evaluation

shows how the number of lookups evolves during the simulation. We can observe that TPT-C efficiently reduces query latencies independently of data distribution and that there is almost no difference between the three distributions. Notice that query latency quickly decreases as caches are being filled. After $200,000$ queries, the caches are full and the query latency decreases slowly.

E. Cache Size

Since the storage capacity of every node is limited, it is not realistic to consider infinite node caches. TPT-C requires very little storage (σ) per node; therefore it is possible to maintain a high number of entries (ϵ) in cache. Each cache entry of TPT-C consists of a key prefix which in the worst case is the number of bits of the indexed space (D bits). In [6] an 80 bit key was used for a real application. In TPT-C, for the same key size, 100 entries would only require $100 \times 80 = 8000$ bits, approximately 1Kb.

We evaluated the impact of the number of cache entries on the performance of TPT-C in terms of lookup operations.

The results are presented in Fig. 4(k). As shown, cache size has a direct impact on the performance of TPT-C. When dropping from 200 entries to 100, latency is degraded by up to 13%. On the other hand, by tripling the cache size from 100 entries to 300, the latency is improved by 17%.

F. Range Query

We compared different strategies for obtaining the results from range queries. Both PHT Linear and PHT Binary searches traverse the double list of leaf nodes. PHT-Parallel search performs a linear search until it reaches the prefix that completely covers the range size and then – if this is an internal node – it forwards the request concurrently to the nodes that overlap the range.

Fig. 4(o) shows the number of lookup operations for PHT Linear, PHT Parallel, and TPT-C Parallel searches for different range sizes (spans). The range span varies from 1% to 5% of the data space for the DBLP distribution and is generated randomly. As expected, PHT Linear search has the highest latency since it accesses all nodes sequentially: latency increases linearly with the range size. PHT Parallel search, however, has to perform several linear searches. Since it is done concurrently, there is no overhead for traversing nodes that store the data: the number of hops will be equal to the longest branch that overlaps the range span. Nevertheless, the traffic in terms of messages grows linearly with the range span:

this can strongly degrade the performance of the system.

When compared to PHT searches, TPT-C Parallel approach is more efficient at reducing latency, which is improved up to 20% in average. It is worth pointing out that the slope is gentler than the ones for PHT Linear search and PHT Parallel search. Such an improvement is the result of parallelising the search while exploiting TPT-C information.

G. Dynamic Scenarios

Traditional cache techniques for prefix trees maintain static references to directly access data. In the case of PHT+Cache, only references to leaf nodes are stored in cache. By exploiting this information, a node can directly contact the leaf node. However, this cache information usually consists of a key and an IP address. Hence, if the node leaves the network, such an entry will be useless for future queries. TPT-C does not use static references; instead, it only exploits logical information about the tree (prefix labels). By gathering information from internal nodes, TPT-C can reduce the search space over the tree even when nodes join or leave the DHT. Therefore, it supports churn.

Since P2P systems are highly dynamic, we evaluated the performance of both *PHT+Cache* and TPT-C when nodes randomly join and leave the DHT under churn conditions.

Simulation was divided into windows of $100,000$ queries. Churn is introduced during a full window followed by another without churn. A 10% churn ratio in relation to the total number of nodes during every simulation window is enough to show the impact of dynamic scenarios on the performance of the two cache approaches.

Fig. 4(p) presents how *PHT+Cache* and TPT-C performance is degraded in a dynamic scenario. Performance degradation percentage corresponds to the relation of latency values between simulations with and without churn.

TPT-C outperforms *PHT+Cache* in dynamic scenarios. It is hardly affected by churn as its performance is degraded by only 3.6%, conversely to *PHT+Cache* whose performance is degraded by 70%. The small difference of TPT-C performance with and without churn is due to new nodes that join the system with empty caches.

VI. Conclusion

In this paper, we propose a new cache for distributed Prefix Trees called Tabu Prefix Table Cache (TPT-C). TPT-C is a solution based on the *Tabu Search*, and it aims at improving the performance of Prefix Trees search. By exploiting the Tabu principle, TPT-C optimises the number of lookup operations performed over internal nodes; therefore it improves both message traffic and query latencies up to 70%. As internal nodes of the trie are free of unnecessary traffic, load balancing among the nodes is also improved.

The obtained results confirm that our solution produces approximately the same amount of messages independently of the height of the trie. Thus TPT-C is efficient independently of the data distribution.

Another interesting feature of TPT-C is that it supports churn. Our results shows that performance is degraded only in 3% in highly dynamic environments, instead traditional PHT cache is degraded up to a 70%.

References

[1] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-grid: a self-organizing structured p2p system. *SIGMOD Rec.*, pages 29–33, 2003.

[2] James Aspnes and Gauri Shah. Skip graphs. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 384–393, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[3] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM*, 34(4):353–366, 2004.

[4] The DBLP Computer Science Bibliography. Dataset of the computer science bibliography. `http://dblp.uni-trier.de/xml/`.

[5] M. Cai, M. Frank, J. Chen, and P. Szekely. Maan: a multi-attribute addressable network for grid information services. In *Grid Computing*, pages 184–191, Nov. 2003.

[6] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A case study in building layered dht applications. In *SIGCOMM*, pages 97–108, NY, USA, 2005. ACM.

[7] A. Chazapis, A. Asiki, G. Tsoukalas, D. Tsoumakos, and N. Koziris. Replica-aware, multi-dimensional range queries in distributed hash tables. *Comput. Commun.*, 33:984–996, May 2010.

[8] Jun Gao and Peter 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*, pages 239–250, Washington, DC, USA, October 2004. IEEE Computer Society.

[9] F. Glover and F. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, 1997.

[10] Z. Hao, J. Hai, and Z. Qin. Yarqs: Yet another range queries schema in dht based p2p network. In *Volume 02*, CIT '09, pages 51–56, Washington, USA, 2009. IEEE CS.

[11] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: a scalable overlay network with practical locality properties. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[12] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: a balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 661–672. VLDB Endowment, 2005.

[13] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[14] Mei Li, Wang-chien Lee, and Anand Sivasubramaniam. Dptree: A balanced tree based indexing framework for peer-to-peer systems. In *Proceedings of the Proceedings of the 2006 IEEE International Conference on Network Protocols*, pages 12–21, Washington, DC, USA, 2006. IEEE Computer Society.

[15] J. Liang and K. Nahrstedt. Randpeer: Membership management for qos sensitive peer-to-peer applications. In *INFOCOM*, 2006.

[16] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. In *ACM PODC*, 2004.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP*, Middleware, pages 329–350, London, 2001.

[18] Z. Runfang and H. Kai. Powertrust: A robust and scalable reputation system for trusted p2p computing. *IEEE Trans. Parallel Distrib. Syst.*, 18:460–473, April 2007.

[19] D. Smith, N-F. Tzeng, and M. M. Ghantous. Fasred: Fast and scalable resource discovery in support of multiple resource range requirements for computational grids. In *NCA*, pages 45–51, Washington, 2008. IEEE CS.

[20] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *ACM Trans. Net.*, 11:17–32, February 2003.

[21] Y. Tang, S. Zhou, and J. Xu. Light: A query-efficient yet low-maintenance indexing scheme over dhts. *IEEE TKDE*, 22(1):59 –75, Jan 2010.

[22] 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 Systems (IPTPS)*, Santa Barbara, CA, USA, February 2006.