Efficient processing of XPath queries with structured overlay networks*

Gleb Skobeltsyn, Manfred Hauswirth, and Karl Aberer

Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{gleb.skobeltsyn, manfred.hauswirth, karl.aberer}@epfl.ch

Abstract. Non-trivial search predicates beyond mere equality are at the current focus of P2P research. Structured queries, as an important type of non-trivial search, have been studied extensively mainly for unstructured P2P systems so far. As unstructured P2P systems do not use indexing, structured queries are very easy to implement since they can be treated equally to any other type of query. However, this comes at the expense of very high bandwidth consumption and limitations in terms of guarantees and expressiveness that can be provided. Structured P2P systems are an efficient alternative as they typically offer logarithmic search complexity in the number of peers. Though the use of a distributed index (typically a distributed hash table) makes the implementation of structured queries more efficient, it also introduces considerable complexity, and thus only a few approaches exist so far. In this paper we present a first solution for efficiently supporting structured queries, more specifically, XPath queries, in structured P2P systems. For the moment we focus on supporting queries with descendant axes ("//") and wildcards ("*") and do not address joins. The results presented in this paper provide foundational basic functionalities to be used by higher-level query engines for more efficient, complex query support.

1 Introduction

P2P systems have been very successful as global-scale file-sharing systems. Typically these systems support simple exact and substring queries which suffice in this application domain. To make P2P systems a viable architectural alternative for more technical and database-oriented applications, support for more powerful and expressive queries is required, though. A couple of approaches have been suggested already on top of unstructured P2P systems and are being applied successfully in practice, for example, Edutella [21]. Unstructured P2P systems do not use indexing, but typically some form of constrained flooding, and thus structured queries are very easy to implement, since each peer receiving the query, which can be arbitrarily complex, can locally evaluate it and return its contribution to the overall result set. However, this comes at the expense of very

* The work presented in this paper was (partly) carried out in the framework of the EPFL Center for Global Computing and supported by the Swiss National Funding Agency OFES as part of the European project BRICKS No 507457.

high bandwidth consumption and some intrinsic limitations. For example, completeness of results cannot be guaranteed, query planning is not possible, and joins are nearly impossible to implement efficiently in large-scale settings.

The efficient alternative are structured P2P systems, as they typically offer logarithmic search complexity in the number of participating nodes. Though the use of a distributed index (typically a distributed hash table) makes the implementation of structured queries more efficient, it also introduces considerable complexity in an environment that is as instable and error-prone as large-scale P2P systems. Thus, so far only a few approaches exist, for example the PIER project [14].

In this paper we present a first solution for the efficient support of structured queries, more specifically, XPath queries, in large-scale structured P2P systems. We assume such a P2P system processing queries expressed in a complex XML query language such as XQuery. XQuery uses XPath expressions to locate data fragments by navigating structure trees of XML documents stored in the network. We refer to this functionality as processing of *structured queries*. In this paper we provide an efficient solution for processing XPath queries in structured P2P networks. We do not address query plans or joins, but focus on a foundational indexing strategy that facilitates efficient answering of structured queries, which we refer to as *structural indexing* in the following. We restrict the supported queries to a subset of the XPath language including node tests, the child axes ("/"), the descendant axes ("//") and wildcards ("*") which we will denote as $XPath_{\{*,//\}}$ in the following. Thus, in this paper we describe an indexing strategy for efficient $XPath_{\{*,//\}}$ query answering in a structured P2P network. Our goal was to provide a basic functional building block which can be exploited by a higher-level query engine to efficiently answer structural parts of complex queries in large-scale structured P2P systems. However, we think that the work presented in this paper provides generally applicable concepts which can be generalized to more complete support of XPath predicates and joins.

The paper is organized as follows: Section 2 gives a brief introduction to our P-Grid structured overlay network which we use to evaluate our approach. Our basic indexing strategy is described in Section 3 whose efficiency is then improved through caching as described in Section 4. The complete approach is then evaluated in Section 5 through simulations. Following that, we position our approach in respect to related work in Section 6 and present our conclusions in Section 7.

2 The P-Grid overlay network

We use the P-Grid overlay network [1, 3] to evaluate the approach presented in this paper. P-Grid is a structured overlay network based on the so-called distributed hash table (DHT) approach. In DHTs peer identifications and resource keys are hashed into one key space. By this mapping responsibilities for partitions of the key space can be assigned to peers, i.e., which peer is responsible for answering queries for what partition. To ensure that each partition of the

key space is reachable from any peer, each peer maintains a routing table. The routing table of a peer is constructed such that it holds peers with exponentially increasing distance in the key space from its own position in the key space. This technique basically builds a small-world graph [16], which enables search in $O(\log N)$ steps. Basically all systems referred to as DHTs are based on variants of this approach and only differ in respect to fixed (e.g., P-Grid, Pastry [25]) vs. variable key space partitioning (e.g., Chord [27]), the topology of the key space (ring, interval, torus, etc.), and how routing information is maintained (redundant entries, dealing with network dynamics and failures, etc.).

Without constraining general applicability we use binary keys in P-Grid. This is not a fundamental limitation as a generalization of the P-Grid system to k-ary structures is natural, and exists. P-Grid peers refer to a common underlying binary trie structure in order to organize their routing tables as opposed to other topologies, such as rings (Chord), multi-dimensional spaces (CAN [24]), or hypercubes (HyperCuP). Tries are a generalization of trees. A trie is a tree for storing strings in which there is one node for every common prefix. The strings are stored in extra leaf nodes. In the following we will use the terms trie and tree conterminously.

In P-Grid each peer $p \in P$ is associated with a leaf of the binary tree. Each leaf corresponds to a binary string $\pi \in \Pi$, also called the *key space partition*. Thus each peer p is associated with a path $\pi(p)$. For search, the peer stores for each prefix $\pi(p,l)$ of $\pi(p)$ of length l a set of references $\rho(p,l)$ to peers q with property $\pi(p,l) = \pi(q,l)$, where $\overline{\pi}$ is the binary string π with the last bit inverted. This means that at each level of the tree the peer has references to some other peers that do not pertain to the peer's subtree at that level which enables the implementation of prefix routing for efficient search. The cost for storing the references and the associated maintenance cost scale as they are bounded by the depth of the underlying binary tree.

Each peer stores a set of data items $\delta(p)$. For $d \in \delta(p)$ the binary key $key(d)$ is calculated using an order-preserving hash function, i.e., $\forall s_1, s_2 : s_1 < s_2 \Rightarrow h(s_1) < h(s_2)$, which is pre-requisite for efficient range querying as information is being clustered. $key(d)$ has $\pi(p)$ as prefix but it is not excluded that temporarily also other data items are stored at a peer, that is, the set $\delta(p, \pi(p))$ of data items whose key matches $\pi(p)$ can be a proper subset of $\delta(p)$. Moreover, for fault-tolerance, query load-balancing and hot-spot handling, multiple peers are associated with the same key-space partition (structural replication), and peers additionally also maintain references $\sigma(p)$ to peers with the same path, i.e., their replicas, and use epidemic algorithms to maintain replica consistency. Figure 1 shows a simple example of a P-Grid tree. Note that, while the network uses a tree/trie abstraction, the system is in fact hierarchy-less, and all peers reside at the leaf nodes. This avoids hot-spots and single-points-of-failures.

P-Grid supports a set of basic operations: *Retrieve(key)* for searching a certain key and retrieving the associated data item, *Insert(key, value)* for storing new data items, *Update(key, value)* for updating a data item, and *Delete(key)* for deleting a data item. Since P-Grid uses a binary tree, *Retrieve(key)* is of

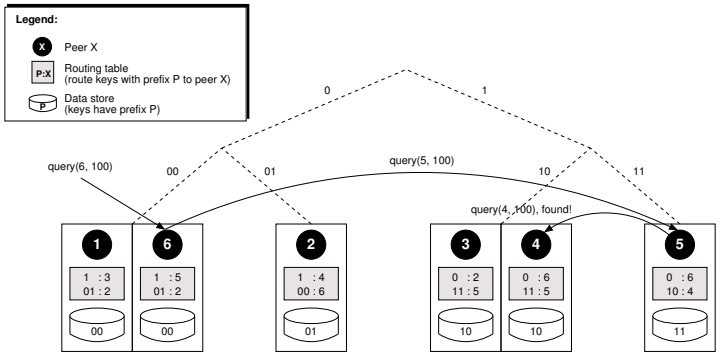

Fig. 1. P-Grid overlay network

complexity $O(\log |\Pi|)$, measured in messages required for resolving a search request, in a balanced tree, i.e., all paths associated with peers are of equal length. Skewed data distributions may imbalance the tree, so that it may seem that search cost may become non-logarithmic in the number of messages. However, in [2] it is shown that due to the randomized choice of routing references from the complimentary subtree, the expected search cost remains logarithmic ($0.5 \log N$), independently of how the P-Grid is structured. The intuition why this works is that in search operations keys are not resolved bit-wise but in larger blocks thus the search costs remain logarithmic in terms of messages. This is important as P-Grid's order-preserving hashing may lead to non-uniform key distributions.

The basic search algorithm is shown in Algorithm 1.

Algorithm 1 Search in P-Grid: $Retrieve(key, p)$

1: **if** $\pi(p) \subseteq key$ **or** $\pi(p) \supset key$ **then**
2: return$(d \in \delta(p) | key(d) = key)$;
3: **else**
4: determine l such that $\pi(key, l) = \overline{\pi(p, l)}$;
5: r = randomly selected element from $\rho(p, l)$;
6: Retrieve(key, r);
7: **end if**

p in the algorithm denotes the peer that currently processes the request. The algorithm always terminates successfully, if the P-Grid is complete (ensured by the construction algorithm) and at least one peer in each partition is reachable (ensured through redundant routing table entries and replication). Due to the definition of ρ and $Retrieve(key, p)$ it will always find the location of a peer at which the search can continue (use of completeness). With each invocation of $Retrieve(key, p)$ the length of the common prefix of $\pi(p)$ and key increases at least by one and therefore the algorithm always terminates.

Insert(key, value) and *Delete(key)* are based on P-Grid's more general update functionality [10], *Update(key, value)*, which provides probabilistic guarantees for consistency and is efficient even in highly unreliable, replicated environments, i.e., $O(\log|\Pi| + replication\ factor)$. An insert operation is executed in two logical phases: First an arbitrary peer responsible for the key-space to which the key belongs is located (*Retrieve(key)*) and then the found peer notifies its replicas about the inserted *key* using a light-weight hybrid push-and-pull gossiping mechanism. Deleting and updating a data item works alike.

3 Basic index

The goal of structural indexing is to provide efficient means to find a peer or a set of peers, that store pointers to XML documents or fragments containing the path(s) matching the queried expression. As we target large-scale distributed XML repositories, we try to minimize the messaging costs, measured in overlay hops, required to answer the query. The intuition of our approach is to use standard database techniques for suffix indexing applied to XML path expressions. Instead of symbols, the set of XML element tags is used as the alphabet.

Given an XML path P consisting of m element tags, $P = l_1/l_2/l_3/\ldots/l_m$, we store m data items in the P-Grid network using the following subpaths (suffixes) as application keys:

- $sp_1 = l_1/l_2/\ldots/l_m$
- $sp_2 = l_2/\ldots/l_m$
  $\vdots$
- $sp_m = l_m$

The key of each data item is generated using P-Grid's prefix-preserving hash function: $key_i = h(sp_i)$. The insertion of the m data items requires $O(m \log N)$ overlay hops. Each data item stores the original XML path to enable local processing and a URI to the XML source document/fragment. We refer to this index as *basic index* in the following.

For example, for the path "*store/book/title*", the following data items (we represent them in a form of {key,data} pairs) will be created:

- $\{h(\text{“}store/book/title\text{”}), (\text{“}store/book/title/\text{”}, URI)\}$
- $\{h(\text{“}book/title\text{”}), (\text{“}store/book/title/\text{”}, URI)\}$
- $\{h(\text{“}title\text{”}), (\text{“}store/book/title/\text{”}, URI)\}$

Any peer in the overlay network can submit an $XPath_{\{*,//\}}$ query. To support wildcards ("*") we consider them as a particular case of descendant axes ("//"). They are converted into "//" and are used only at the local lookup stage as a filtering condition. I.e., our strategy is to preprocess a query replacing all "*" by "//", for example, "$A/*/B$" → "$A//B$", answer the transformed query using our distributed index and filter the result set by applying the original query

to it, thus we obtain the intended semantics of "*". In this paper we concentrate on general indexing strategy and do not address possible optimizations on this issue.

Let q_B denote the longest sequence of element tags divided by child axes ("/") only, which we will call the *longest subpath of a query* in the following. For example, for the query "$A//C/D//F$", q_B = "C/D".

When a query is submitted to a peer, the peer generates a query message that contains the path expression and the address of the originating peer and starts the basic structural querying algorithm as shown in Algorithm 2.

Algorithm 2 Querying using basic index: $AnswerQuery(query, p)$

```
1: compute q_B of query;
2: key = h(q_B)
3: if π(p) ⊆ key then
4:    return(d ∈ δ(p) | isAnswer(d, query) = true);
5: else if π(p) ⊃ key then
6:    ShowerBroadcast(query, length(key), p);
7: else
8:    determine l such that π(key, l) = π(p, l)‾;
9:    r = randomly selected element from ρ(p, l);
10:   AnswerQuery(query, r);
11: end if
```

The function $AnswerQuery(query, p)$ extends $Retrieve(key, p)$ described in Algorithm 1 for answering the $XPath_{\{*,//\}}$ $query$ using the basic index. First the search key is computed by hashing the query's longest subpath q_B. Then we check whether the currently processing peer is the only one responsible for key. If yes, the routing is finished and the result set is returned (line 4). Function $isAnswer(d, query)$ examines if the data item d is a correct answer for $query$. Alternatively, if routing is finished at one of the peers from the sub-trie defined by key (line 5) [1], all peers from this sub-trie could store relevant data items and have to be queried. To do this, we use a variant of the broadcasting algorithm (line 6) for answering range queries described in [11] as shown in Algorithm 3, where the range is defined by key prefix. I.e., we query all peers for which $key \subset \pi(p)$.

The algorithm starts at an arbitrary peer from the sub-trie, and the query is forwarded to the other partitions in the trie using this peer's routing table. The process is recursive, and since the query is split in multiple queries which appear to trickle down to all the key-space partitions in the range, we call it the *shower algorithm*.

[1] I.e., the key is a proper substring of the peer's path ($\pi(p) \supset key$), which means that all bits of the key have been resolved and the query has reached a sub-trie, in which several peers may store data belonging to the query's answer set, and all have to be checked for possible answers (this is ensured by P-Grid's clustering property)

Algorithm 3 $ShowerBroadcast(query, l_{current}, p)$

1: **for** $l = l_{current}$ to $length(\pi(p))$ **do**
2: r = randomly selected element from $\rho(p, l)$;
3: $ShowerBroadcast(query, l + 1, r)$;
4: **end for**
5: return$(d \in \delta(p) \mid isAnswer(d, query) = true)$;

With basic indexing the expected cost (in terms of messages) of answering a single query is $O(L) + O(S) - 1$, where L is the cost of locating any peer in the sub-trie and S is the shower algorithm's messaging cost. The expected value of L is a length of the sub-trie's prefix. The intuition for this value is that it is analogous to the search cost in a tree-structured overlay of size 2^L. The expected value of L is $N/2^L$, which refers to the number of peers in the sub-trie. The latency remains $O(\log N)$ because the shower algorithm works in a parallel fashion.

To illustrate how a query is answered using the basic index, assume the $query$ = "$A//C/D//E$" is submitted at some peer p. Following Algorithm 2 the peer responsible for h("C/D") is located. Assume there is a sub-trie defined by the prefix h("C/D") as it is depicted in Figure 2. The shower broadcast is executed and every peer in the sub-trie performs a local lookup for $query$ and sends the result to the originating peer p.

4 Caching strategy

The basic index is efficient in finding all documents matching an $XPath_{\{*,//\}}$ query expression based on the longest sequence of element tags (q_B). It performs well with queries containing a relatively long $h(q_B)$, such that the number and the size of shower broadcasts is not excessive. However, the search cost might be substantially higher for queries, which require large broadcasts, i.e., $h(q_B)$ is short. For example, queries like "A//B" are answered by looking up the peer responsible for h("A") and then a relatively expensive broadcast depending on the data in the overlay may have to follow. The search would be more efficient if knowledge about the second element tag "B" would be employed as well. In this section we introduce a caching strategy to address this issue, which allows us to reduce the number of broadcasts, and thus, decrease the average cost of answering a query.

Each peer which receives a query determines if it belongs to one of the following types:

1. Queries that can be answered locally, i.e., $\pi(p) \subseteq h(q_B)$. For example the path "A/B/C//E" at the peer responsible for h("A/B").
2. Queries that require additional broadcasts, i.e., $\pi(p) \supset h(q_B)$, but contain only one subpath, $query = q_B$. For example, the path "A" at the peer responsible for h("A/B"). In this case matching index items are stored on

all the peers responsible for $h(\text{“}A\text{”})$. As queries of this type may be very expensive, for example “//”, they could be disabled in the configuration or only return part of the overall answer set to constrain costs.

3. Queries that require an additional broadcast, $\pi(p) \supset h(q_B)$, but include at least one descendant axis (“//”) or wildcard (“*”), i.e. $q_B \neq q$. For example the query “A//C//E” at the peer responsible for $h(\text{“}A/C\text{”})$. The result set for such queries can be cached locally and accessed later without performing a shower broadcast.

Type 1 queries are inexpensive and thus work well with basic indexing. *Type 2 queries* are so general that they return undesirably large result sets and the system may want to block or constrain them. The most relevant type of queries whose costs should be minimized are thus *type 3 queries* which we will address in the following. For simplifying the presentation we assume that only one peer is responsible for a given query and have resources to cache results. We can assume that storage space is relatively cheap as the “expensive” resource in overlay networks is network bandwidth. However, each peer is entitled to arbitrarily limit the size of its cache at will.

4.1 Answering a query

As a first step Algorithm 2 is modified by changing the routing and adding cache handling. If we sort the subpaths of an $XPath_{\{*,//\}}$ query by their length in descending order, we can “rewrite” the original query as $q_C = concat(P_{l_1}, P_{l_2}, ..., P_{l_k})$, where P_{l_i} is the i-st longest subpath. We will use q_C for routing purposes instead of q_B, which gives us the benefit that we use the whole query for generating the routing key. The modified querying algorithm is shown in Algorithm 4.

Algorithm 4 Querying using basic index extended with cache: $AnswerQueryWithCache(query, p)$

```
1: compute q_C of the query;
2: key_C = h(q_C)
3: compute q_B of the query;
4: key_B = h(q_B)
5: if π(p) ⊆ key_B then
6:    return(d ∈ δ(p) | isAnswer(d, query) = true);
7: else if (π(p) ⊃ key_B) and (ifCached(query) = false) then
8:    ShowerBroadcast(query, length(key_B), p);
9: else if π(p) ⊆ key_C then
10:   return(d ∈ cache(p) | isAnswer(d, query) = true);
11: else
12:   determine l such that π(key_C, l) = overline(π(p, l));
13:   r = randomly selected element from ρ(p, l);
14:   AnswerQueryWithCache(query, r);
15: end if
```

In line 1 we compute q_C which is used for routing (line 12) to the peer (probably) storing a cached result set. Since P-Grid uses a prefix-preserving hash function and $q_B \subseteq q_C$ (q_B is always the first subpath of q_C), this peer is located in the $key_B = h(q_B)$ sub-trie.

Similarly to the basic index's search algorithm we check whether the currently processing peer is the only one responsible for key_B (line 5). If yes, the result set is returned (line 6). If the routing reached one of the peers from the sub-trie defined by key_B, we execute the shower broadcast (line 8) to answer the query as introduced in the previous section, but only if the query has not already been cached (line 7). Section 4.2 explains how the function $ifCached(query)$ works. If the query is cached, the routing proceeds until the peer responsible for key_C is reached. This peer answers the query by looking up a cached result set (line 10).

4.2 Cache maintenance

Each peer runs a cache manager, which is responsible for cache maintenance. Two functions $createCache(query)$ and $deleteCache(query)$ are available, where $query$ is any query the peer is responsible for. In the following we explain how these functions work. How the cache manager decides if a query is worth caching or not will be described in 4.3.

To cache a query a peer determines a sub-trie's prefix by hashing q_B and collects a result set for the query by executing a special version of the shower broadcast algorithm. The only difference with regard to the $ShowerBroadcast$ listed in Algorithm 3 is that for cache consistency reasons all the peers in the broadcast sub-trie add the query expression to their *lists of cached queries* L_{CQ}. Thus, in case the P-Grid is updated, i.e. data items are inserted, modified or deleted, any peer from the sub-trie can contact the peer(s) that cache relevant queries, to inform them of the change so they can keep their cache consistent. This operation needs $O(\log N)$ messages per cache entry. The function $ifCached(query)$ (line 7, Algorithm 4) looks up the (locally maintained) L_{CQ} list to determine if the query is cached. This solution requires additional storage space which can be significantly decreased by the use of Bloom filters. Similarly, the cache deletion operation requires updates of all L_{CQ} lists.

When a data item is inserted, updated or deleted, all relevant cache entries are updated respectively. A peer looks up the cached queries list and sends update messages to all the peers caching the relevant queries. Each cache update requires a message to be routed with an expected cost of $0.5 \log N$. If we denote as $C(path)$ the number of cached queries that have $path$ as an answer, the update cost can be estimated as $O(\log N) + O(C(path) * 0.5 \log N)$.

4.3 What to cache?

The cache manager analyzes the benefits of caching for each candidate query the peer is responsible for. To do so, it estimates the overall messaging cost for

the query with and without caching. The decision to cache the query result or to delete the existing cache entries is based on comparing these two values.

If the query is cached, each search operation for that query saves a shower broadcast (the shower broadcast requires $s-1$ messages where s is the number of peers in the trie). On the other hand each update operation for any data item related to the query will cost additional $O(\log N)$ messages to update the cache. Knowing the approximate ratio of search/update operations (obtained by local monitoring) the peer can make an adaptive decision on caching of a particular query.

The query is considered to be profitable to cache if:

$$UpdateCost * UpdateRate(subtrie) < SearchCost(subtrie) * SearchRate(query)$$

where

- *subtrie* is the prefix of the q_B sub-trie, i.e., the basic index's shower broadcast sub-trie;
- $UpdateCost$ is the cost of one update, which is equal to the routing cost, i.e., $O(\log N)$;
- $UpdateRate(subtrie)$ is the average update rate in the given sub-trie;
- $SearchCost(subtrie)$ is the number of peers in the sub-trie to be contacted to answer the shower broadcast; and
- $SearchRate(query)$ is the search rate for the given query.

To estimate these values each peer collects statistics. For $SearchRate$ the peer's local knowledge is sufficient, whereas $UpdateCost$ and $SearchCost$ values have to be gathered from the neighbors. To do so, we can periodically flood the network or better employ the much more efficient algorithm described in [4]. This algorithm gossips the information about the tree structure among all the peers in the network. Each peer maintains an approximate number of peers in each sub-trie it belongs to (as many values as the peer's prefix length). The values are exchanged via local interactions between peers and a piggyback mechanism avoids sending additional messages. The same idea is used to gossip the $UpdateRate$ in every sub-trie a peer belongs to.

4.4 Example

An example illustrating the application of caching is shown in Figure 2.

Note, that in Figure 2 each element tag is represented by one capital letter and we omit child axes ("/") to simplify the presentation. The numbers 1–4 written in brackets next to the arrows correspond to the following steps:

1. The cache manager at the peer II decides to cache a result set for the query Q = "$A//C/D//E$". The shower broadcast to the peers responsible for h("C/D") is initiated to fill up the cache with all data items matching the query. It reaches the peers I, III and IV. They add Q to their lists of cached queries.

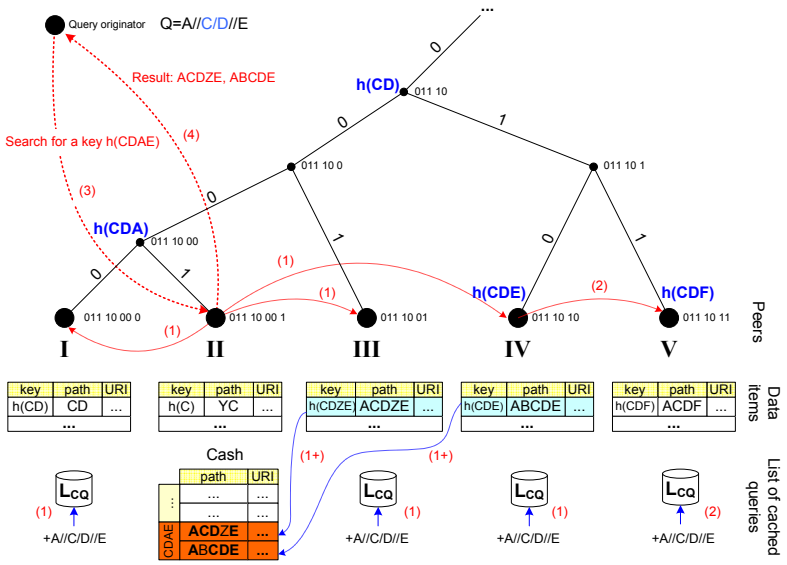

Fig. 2. Caching strategy example

2. Peers III and IV send back the matching items. The shower broadcast reaches peer V, which also adds Q to its list of cached queries. 4 messages were sent to execute the shower broadcast in sub-trie $h(\text{“}C/D\text{”})$.
3. Assume, the query $Q = \text{“}A//C/D//E\text{”}$ is submitted at the originating peer. The search message is routed to the peer II ($O(\log N)$), which can answer a query locally by looking up its cache. The broadcast has to be executed every time to answer the query Q if it is not cached.
4. The answer is sent back to the originating peer.

Assume now, a new path “A/C/D/E” is indexed. One of the four (see Section 3) generated data items with the key $h(\text{“}C/D/E\text{”})$ is added to the peer V. It checks the list of cached queries and finds query $Q = \text{“}A//C/D//E\text{”}$ to be concerned by this change. Peer V sends a cache update message to the peer responsible for Q, i.e., to Peer II, which ensures cache consistency.

5 Simulations

To justify our approach and its efficiency, we implemented a simulator of a distributed XML storage, based on the P-Grid overlay network. The simulator is written in Java and stores all data locally in a relational database. As the simulation results in this section meet our theoretical expectations we will in a next step implement our approach on top of our P-Grid implementation [22] and test it on PlanetLab.

As input data for our experiments, we use about 50 XML documents (mainly from [28]) from which we extracted a *path collection* of more than 1000 unique

paths. Based on each path in the collection we generated four additional paths by randomly distorting the element tags. Using the resulted path collection (about 5000 paths) we generate a P-Grid network by inserting a corresponding number of data items per each path (about 20000 data items overall). P-Grid networks of different sizes can be obtained by limiting the maximum number of data items a peer can store.

For our experiments we generated different *query collections* by randomly removing some element tags from the paths in the path collection. A parameter t specifies query construction and ensures percentage of type 3 ("cachable") queries in the collection.

To emulate the querying process we generated a *query load* of 10000 queries by applying different distributions on the query collection. In the following experiments an average search cost value for given parameters is computed by processing all queries in the query load.

In the first experiment we assume that all possibly "cacheable" queries are in fact being cached. We vary the network size and measure the average cost of answering one query. The query load is uniformly distributed and different t parameters are used. In Figure 3 the first four curves show the average search cost for $t = 0$, 0.5, 0.75 and 1 respectively. Obviously, the more queries are being cached, the lower the search cost becomes. The fifth curve shows the cost of locating at least one peer responsible for the query, i.e., the search cost without shower broadcasts. Evidently, the two last curves coincide because if all queries are cached no shower broadcasts are required.

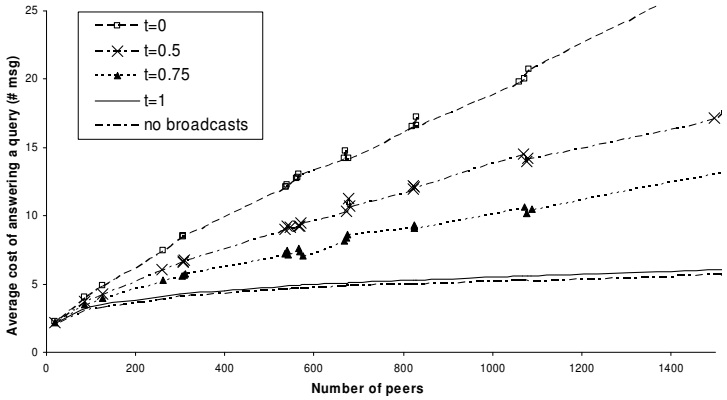

Fig. 3. Average number of messages required to answer a query depending on the network size, t denotes the fraction of "cacheable" queries

However, query load does not necessary follow a uniform distribution. Instead, a Zipfian distribution is more realistic as shown in Figure 4. In the experiment we fixed the network size to 1000 peers, $t = 0.5$ and vary the cache size. The first curve shows the constant search cost if caching is disabled. The other

three curves correspond to the different parameters of the Zipf distribution of the query load and show how our approach performs under these conditions.

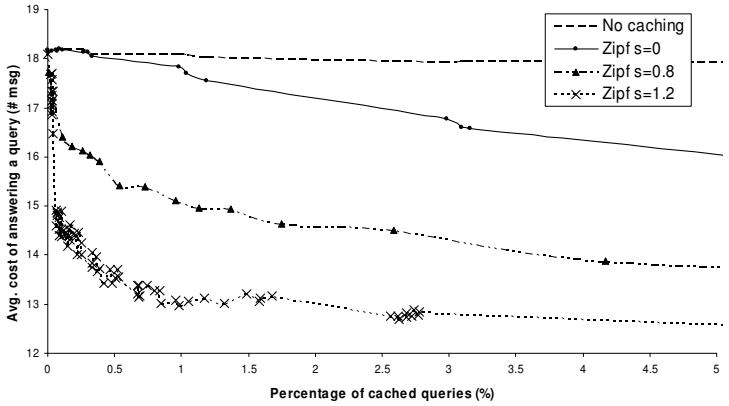

Fig. 4. Average number of messages required to answer a query in the network of 1000 peers depending on the fraction of cached queries

However, the benefits we gain from caching for querying, come at the price of increasing the update costs. To perform one update operation, for example, to insert a new path containing m element tags, we have to contact all the peers responsible for all the subpaths ($O(m \log N)$). We also have to update all relevant cache entries ($O(\log N)$ per cache entry). Figure 5 shows the average update costs depending on size of the network.

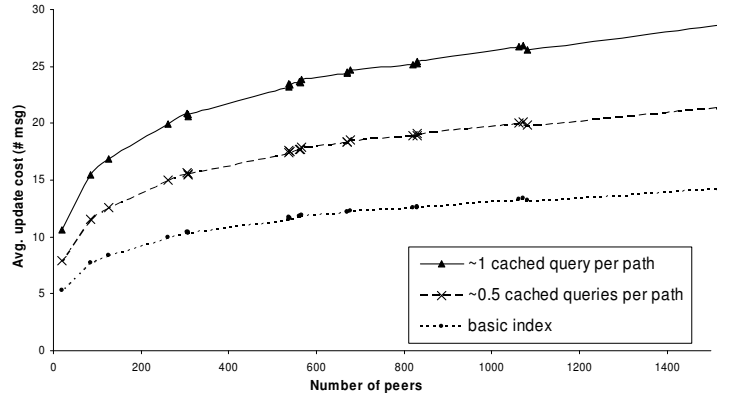

Fig. 5. Average update cost depending on the network size, t denotes the percentage of "cacheable" queries

In Section 4.3 we described the strategy for minimizing the overall messaging costs. In the last experiment we show that for a given state of the system this minimum can be achieved by choosing what queries to cache. In Figure 6 we show that for the given fixed parameters (1000 peers, $t = 0.5$, Zipf $s = 1.2$, average number of element tags in the path = 2.5) the overall messaging cost can be minimized. We show two curves for search/update ratios of 1:2 and 2:1. In these cases the minimal messaging costs are achieved if about 0.5% and 1.0% of the queries are being cached.

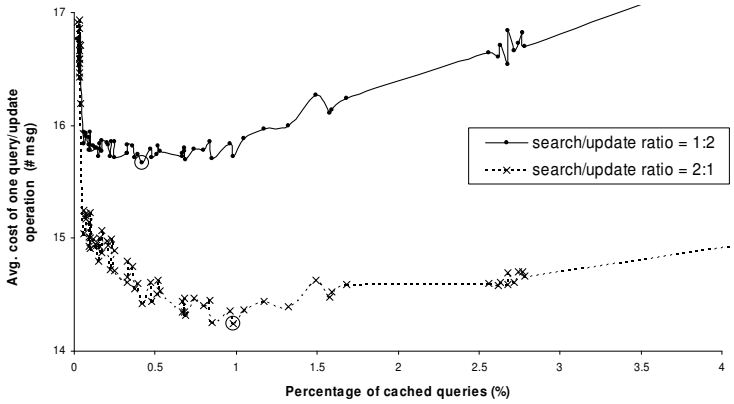

Fig. 6. Average number of messages (query + update) depending on the fraction of cached queries

Evidently, if the search/update ratio is high (more searches than updates) the minimum moves to the right (more queries are to be cached). In contrast, if the update ratio is relatively high, the minimum moves to the left (up to 0, where caching is not profitable anymore). Hence, (1) the higher the search/update ratio is, the more queries should be cached and (2) our solution is adaptive to the current system state and minimizes the overall messaging costs.

The simulations show that the basic index strategy is sufficient for building a P2P XML storage with support for answering structured queries. The introduction of caching decreases the messaging costs. Depending on the characteristics of the query load the benefits from caching vary.

6 Related Work

Many approaches exist that deal with querying of XML data in a local setting. Most of them try to improve the query-answering performance by designing an indexing structure with respect to local data processing. Examples of such index structures include DataGuides [13], T-indexes [20], the Index Fabric [7], the Apex approach [6] and others. However, these approaches are not designed to support a large-scale distributed XML storage.

On the other hand, peer-to-peer networks yield a practical solution for storing huge amounts of data. Thus, a number of approaches exist that try to leverage a P2P network for building a large-scale distributed data warehouse. The important properties of such systems are:

- The flexibility of the querying mechanism (e.g. query language).
- The messaging and maintenance costs.

The use of routing indices [8] facilitates the construction of a P2P network based on content. In such content-based overlay networks peers are linked, if they keep similar data, i.e., each peer maintains summaries of the information stored at its neighbors. While searching, a peer uses the summaries to determine whom to forward a query to. The idea of clustering peers with semantically close content is exploited in [9]. The approach presented in [17] proposes using multi-level bloom filters to summarize hierarchical data, i.e., similarity of peers' content is based on the similarity of their filters. In [23] the authors use histograms as routing indexes. A decentralized procedure for clustering of peers based on their histogram distances is proposed. The content-based approaches could efficiently solve the problem of answering structured queries, though lack of structure affects the result set quality and significantly increases the search cost for large-scale networks.

The Edutella project [21] is a P2P system based on a super-peer architecture, where super-peers are arranged in a hypercube topology. This topology guarantees that each node is queried exactly once for each query, which presumes powerful querying facilities including structured queries, but does not scale well.

Leveraging DHTs to support structured queries decreases the communication costs and improves scalability, but requires more complicated query mechanisms. The approach presented in [12] indexes XML paths in a Chord-based DHT by using tag names as keys. A peer responsible for an XML tag stores and maintains a data summary with all possible unique paths leading to the tag. Thus, only one tag of a query is used to locate the responsible peer. Although ensuring high search speed, the approach introduces considerable overhead for popular tags, when the data summary is large. Our solution for this case is to distribute the processing among the peers in a subtrie. The paper also addresses answering branching XQuery expressions by joining the result sets obtained from different peers. A similar mechanism can be employed for our approach.

[5] also uses a Chord network, but follows a different technique. Path fragments are stored with the information about the super- and child-fragments. Having located a peer responsible for a path fragment, it resolves the query by navigating to the peers responsible for the descendant fragments. Additional information has to be stored and maintained to enable this navigation, which causes additional maintenance costs. For some types of queries the search operation may be rather expensive due to the additional navigation.

Some approaches also employ caching of query results in a P2P network to improve the search efficiency. For example, [18] proposes a new Range Addressable Network architecture that facilitates range query lookups by storing the

query results in a cache. In [26] the authors leverage the CAN P2P network to address a similar problem. In both cases queries are limited to integer intervals. The ranges themselves are hashed, which makes simple key search operation highly inefficient.

The PIER project [14, 15] utilizes a DHT to implement a distributed relational query engine bringing database query processing facilities into a P2P environment. In contrast, our approach solves the particular problem of answering structured XPath queries, which is not addressed by PIER. However, many of query processing mechanisms (join, aggregation, etc.) proposed in PIER can be also employed for building a DHT-based large-scale distributed XML storage with powerful query capabilities. The paper [19] leverages the PIER for building a file-sharing P2P system for answering multi-keyword queries. The authors suggest using flooding mechanisms to answer popular queries, and use DHT's indexing techniques for rare queries.

7 Conclusions

In this paper we presented the efficient solution for indexing structural information in a structured overlay network used as distributed P2P storage of XML documents. We based the approach on the P-Grid structured overlay network, however, the solution can be ported to similar tree-based DHTs. We demonstrated the efficiency (low search latency and low bandwidth consumption) of our approach via simulations and also showed that our proposed caching strategy chooses the optimal strategy for minimizing messaging costs.

We envision that the presented solution can be used in a P2P XML querying engine for answering structural (sub)queries. Such a system could be an alternative to the solutions based on the unstructured P2P networks (e.g., Edutella [21]), but more scalable due to the considerably reduced messaging costs. As a next step, we plan to extend the system to support more general XPath queries.

References

1. Karl Aberer. P-grid: A self-organizing access structure for p2p information systems. In *CoopIS'01: Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 179–194, London, UK, 2001. Springer-Verlag.
2. Karl Aberer. Scalable Data Access in P2P Systems Using Unbalanced Search Trees. In *WDAS'02: Proceedings of the 4th Workshop on Distributed Data and Structures*, 2002.
3. Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-Grid: A Self-organizing Structured P2P System. *SIGMOD Record*, 32(3), 2003.
4. Keno Albrecht, Ruedi Arnold, Michael Gahwiler, and Roger Wattenhofer. Join and Leave in Peer-to-Peer Systems: The Steady State Statistics Service Approach. Technical Report 411, ETH Zurich, 2003.

5. Angela Bonifati, Ugo Matrangolo, Alfredo Cuzzocrea, and Mayank Jain. Xpath lookup queries in p2p networks. In *WIDM'04: Proceedings of the 6th annual ACM international workshop on Web information and data management*, pages 48–55, New York, NY, USA, 2004. ACM Press.
6. Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. Apex: an adaptive path index for xml data. In *SIGMOD'02: Proceedings of the ACM SIGMOD 2002 International Conference on Management of Data*, pages 121–132, New York, NY, USA, 2002. ACM Press.
7. Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *VLDB'01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 341–350, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
8. Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS'02: Proceedings of the 28th Int. Conference on Distributed Computing Systems*, July 2002.
9. Arturo Crespo and Hector Garcia-Molina. Semantic overlay networks for p2p systems. Technical report, Computer Science Department, Stanford University, 2002.
10. Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *ICDCS'03: Proceedings of the International Conference on Distributed Computing Systems*, 2003.
11. Anwitaman Datta, Manfred Hauswirth, Roman Schmidt, Renault John, and Karl Aberer. Range queries in trie-structured overlays. In *P2P'05: Proceedings of the 5th International Conference on Peer-to-Peer Computing*, August 2005. http://lsirpeople.epfl.ch/rschmidt/papers/Datta05RangeQueries.pdf.
12. Leonidas Galanis, Yuan Wang, Shawn R. Jeffery, and David J. DeWitt. Locating data sources in large distributed systems. In *VLDB'03: Proceedings of the 29th International Conference on Very Large Data Bases*, pages 874–885, 2003.
13. Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97: Proceedings of the 23th International Conference on Very Large Data Bases*, pages 436–445, 1997.
14. M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In *IPTPS'02: Proceedings for the 1st International Workshop on Peer-to-Peer Systems*, 2002.
15. Ryan Huebsch, Brent Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R. Yumerefendi. The architecture of pier: An internet-scale query processor. In *CIDR'05: Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2005.
16. Jon Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *STOC'00: Proceedings of the 32nd ACM Symposium on Theory of Computing*, 2000.
17. Georgia Koloniari and Evaggelia Pitoura. Content-based routing of path queries in peer-to-peer systems. In *EDBT'04: Proceedings of 9th International Conference on Extending Database Technology*, pages 29–47, 2004.
18. Anshul Kothari, Divyakant Agrawal, Abhishek Gupta, and Subhash Suri. Range addressable network: A p2p cache architecture for data ranges. In *P2P'03: Proceedings of the 3rd International Conference on Peer-to-Peer Computing*, pages 14–22, 2003.

19. Boon Thau Loo, Ryan Huebsch, Joseph M. Hellerstein, Scott Shenker, and Ion Stoica. Enhancing p2p file-sharing with an internet-scale query processor. In *VLDB'04: Proceedings of the 30th International Conference on Very Large Data Bases*, August 2004.
20. Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT'99: Proceeding of the 7th International Conference on Database Theory*, pages 277–295, London, UK, 1999. Springer-Verlag.
21. Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. Edutella: a p2p networking infrastructure based on rdf. In *WWW'02: Proceedings of the eleventh international conference on World Wide Web*, pages 604–615, New York, NY, USA, 2002. ACM Press.
22. http://www.p-grid.org.
23. Yannis Petrakis, Georgia Koloniari, and Evaggelia Pitoura. On using histograms as routing indexes in peer-to-peer systems. In *DBISP2P'04: Proceedings of the Second International Workshop on Databases, Information Systems, and Peer-to-Peer Computing*, pages 16–30, 2004.
24. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM'01: Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 161–172, 2001.
25. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM'01: Proceedings of the 18th International Conference on Distributed Systems Platforms*, pages 329–350, 2001.
26. Ozgur D. Sahin, Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. A peer-to-peer framework for caching range queries. In *ICDE'04: Proceedings of the 20th International Conference on Data Engineering*, pages 165–176, 2004.
27. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM'01: Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 149–160. ACM Press, 2001.
28. http://www.cs.washington.edu/research/xmldatasets/.