Locating Data Sources in Large Distributed Systems

Leonidas Galanis Yuan Wang Shawn R. Jeffery David J. DeWitt

Computer Sciences Department, University of Wisconsin - Madison
1210 W Dayton St
Madison, WI 53706
USA
{lgalanis, yuanwang, jeffery, dewitt}@cs.wisc.edu

Abstract

Querying large numbers of data sources is gaining importance due to increasing numbers of independent data providers. One of the key challenges is executing queries on all relevant information sources in a scalable fashion and retrieving fresh results. The key to scalability is to send queries only to the relevant servers and avoid wasting resources on data sources which will not provide any results. Thus, a catalog service, which would determine the relevant data sources given a query, is an essential component in efficiently processing queries in a distributed environment. This paper proposes a catalog framework which is distributed across the data sources themselves and does not require any central infrastructure. As new data sources become available, they automatically become part of the catalog service infrastructure, which allows scalability to large numbers of nodes. Furthermore, we propose techniques for workload adaptability. Using simulation and real-world data we show that our approach is valid and can scale to thousands of data sources.

1. Introduction

Our vision is demonstrated by the following scenario: At some computer terminal of a large distributed system a user issues a query. Based on the query, the system determines where to look for answers and contacts each node containing relevant data. Upon completion of the query, regardless of the number of results or how they are ranked and presented, the system guarantees that all the relevant data sources known at query submission time have been contacted. The naïve way to implement our vision would be to send a query to each of the participating nodes in the network. While this approach would work for a small number of data providers it certainly does not scale. Hence, when a system incorporates thousands of nodes, a facility is needed that allows the selection of the subset of nodes that will produce results, leaving out nodes that will definitely not produce results. Such a facility implies the deployment of catalog-like functionality.

A catalog service in a large distributed system can be used to determine which nodes should receive queries based on query content. Additionally it can be used to perform other tasks such as query optimization in a distributed environment. There are three basic designs for building a catalog service for a distributed system: 1) A central catalog service, 2) a fully-replicated catalog on each participating node, or 3) a fully distributed catalog service. A centralized design implies a resource exclusively dedicated to servicing catalog requests. Existing technology allows the construction of such servers that could sufficiently handle thousands of nodes. Such a solution, however, requires a central infrastructure and a scheme to share expenses among the participating peers. To avoid this each node in the system can take over the burden of catalog maintenance. To this end, one simple design is the use of a fully replicated catalog on each peer (as practiced in distributed database systems [18]). When a new peer joins the system it downloads the catalog from any existing peer and it can immediately query the entire community. Nevertheless, maintenance of the catalogs requires $O(n^2)$ number of messages for the formation of a network of n nodes. Clearly, this is not scalable to thousands of nodes.

We focus on a fully distributed architecture motivated by recent advances in peer-to-peer computing (P2P). P2P systems research has proposed a number of new distributed architectures with desirable traits, including no central infrastructure, better utilization of distributed resources, and fault tolerance. Particular attention has been paid into making these systems scalable to large numbers

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment
Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003

of nodes, avoiding shortcomings of the early P2P pioneers such as file sharing systems like Gnutella [9] and Napster [17]. Representatives of scalable location and routing protocols are CAN [21], Pastry [22], Chord [25] and Tapestry [32], henceforth referred to as Distributed Hash Tables or DHTs. Each of these protocols, however, allows only simple key based lookup queries.

This paper studies the feasibility of using existing P2P technology as the basis for efficiently facilitating complex queries over an arbitrary large number of data repositories. Given an arbitrary query *q* and a large number of data repositories, our goal is to send *q* only to the repositories that have data relevant to *q* without relying on a centralized catalog infrastructure. Additionally, data repositories must be able to join the P2P system and make their data available for queries. Our design builds on current P2P technologies. The contributions of this work are:

- A catalog framework for locating data sources
- A fully decentralized design of a distributed catalog service that allows data providers to join and make their data query-able by all existing peers.
- Techniques to adapt to the query workload and distribute the catalog service load fairly across the participating nodes.
- An experimental evaluation of a distributed catalog for locating data sources in large distributed XML repositories.

The rest of this paper is organized as follows: The system model of our envisioned catalog service is described in Section 2 where a simple example demonstrates its application. Section 3 discusses the desired features of the data summaries for our distributed catalog and proposes two designs. In Section 4 we show how our system evolves as new nodes join. Section 5 describes how the catalog service is used in order to direct queries to the relevant data sources. Section 6 points out load balancing issues and proposes effective solutions. Section 7 presents our experiments. Related work and Conclusions (Sections 8 and 9) follow at the end.

2. System Model

Conceptually the system allows an arbitrary number of data providers or nodes to join and make their data available. Let N_i ($1 < i \leq n$) denote the n nodes, each of which publishes a set D_i of data objects. When a node N_i wants to join the system it creates catalog information which is the set $\mathbf{C_i} = \{(k_j, S_{ij}) \mid S_{ij}$ is a summary of k_j on node $N_i\}$. The items k_j are present in the data objects D_i. In an XML repository if D_i is a set of documents, the k_j's will be a subset of the attribute and element names in D_i. Each S_{ij} is summary catalog information (or *data summary*) corresponding to k_j and depends on the data on node N_i. For example, a data summary for the element *price* on node N_i might contain all the unique paths that lead to *price* as well as a histogram of *price*'s values.

The catalog service determines which nodes a query Q should execute on using the functions *query_parts*() and *map*(). The function *query_parts* extracts a set of k_j's from a query. Given a query Q. The function *map*: $\{Q\}\times\{\{C_i \mid 1 < i \leq n\}\} \rightarrow \{N_i \mid 1 < i \leq n\}$, uses *query_parts*() to examine the relevant sets of data summaries S_{ij} in order to determine the nodes storing data relevant to Q. Of course the catalog service may contain additional information but this paper focuses on the implementation of *map* when the number of nodes in the system becomes very large.

One possible *map* function would be the constant function $map(Q, \{\mathbf{C_i}\}) = \{N_i \mid 1< i \leq n\}$. However, such an implementation would not scale for large values of n since it would require contacting every node for every query. The study in [10] demonstrates on a real system that the key to scalability is minimizing the number of messages in the distributed system. Hence, our goal is to implement $map(Q, \{\mathbf{C_i}\}) = \{N \mid P_1 \vee P_2\}$ where P_1: *Executing* Q *on* N *yields a non-empty results set* and P_2: *Executing part of* Q *on* N *is required to produce the final result set for* Q. Proposition P_2 covers the case in which Q requires a join or an intersection of data across different nodes. To achieve our goal, our implementation of map employs a fully distributed catalog design based on DHTs.

2.1. DHT Background

The DHTs have very desirable characteristics. Their goal is to provide the efficient location of data items in a very large and dynamic distributed system without relying on any centralized infrastructure. Thus, given a key, the corresponding data item can be efficiently located using only *O*(logn) network messages ([22], [25]) where n is the total number of nodes in the system. Moreover, the distributed system evolves gracefully and can scale to very large numbers of nodes. Hence, current DHT designs provide a means to create large fully distributed dynamic networks of nodes for storage and efficient retrieval of objects. Our work leverages this functionality to provide a scalable fully distributed catalog service. Chord, which serves as the experimental substrate of our work, is publicly available and has been successfully used in other projects such as CFS [7]. Nevertheless, our design does not depend on the specific DHT implementation and can work with any of the aforementioned DHT protocols.

The Chord protocol supports just one lookup operation: It maps a given key to a node. Depending on the application this node is responsible for associating the key with the corresponding data item (object). Chord uses hashing to map both keys and node identifiers (such as IP address and port) onto the identifier ring (Figure 1). Each key is assigned to its successor node, which is the nearest node traveling the ring clockwise. Nodes and keys may be added or removed at any time, while Chord maintains efficient lookups using just *O*(logn) state on each node in

the system. For a detailed description of Chord and its algorithms refer to [25].

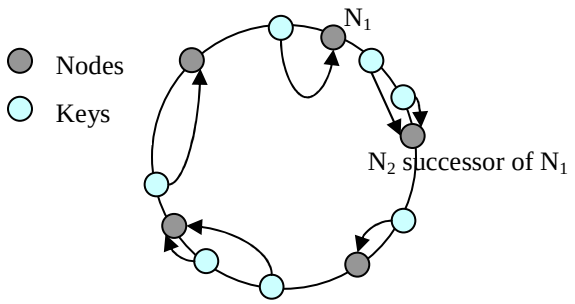


Figure 1: The Chord identifier ring

2.2. Keys and Objects

One of the design challenges is to determine how to map the catalog information to the DHT. Based on our conceptual catalog model the obvious choice for keys are the items k_j, henceforth also referred to as keys. The objects stored are sets of data summaries S_{ij}, and not just single data summaries since mapping from k_j to S_{ij} is not 1 to 1.

An example using a simple XPath [29] query illustrates how the outlined concepts can be put into practice. Consider four XML repositories which contain the data shown in Table 1. Assume that element tags are chosen as keys k_j and that each summary S_{ij} contains a *set* of all the possible paths in the data that lead to k_j. For example $S_{1,\,author}$ ={library/catalogs/book, library/reservation/book} while $S_{2,\,author}$ = {bookstore/book}. Table 2 shows how the DHT assigned the keys to the nodes that joined the network. The summary sets are stored along with the keys.

Paths in XML Data	
N_1	library/catalogs/book/author, library/reservation/book/author
N_2	bookstore/book/price, bookstore/book/author
N_3	bookstore/book/price, bookstore/book/author
N_4	bookstore/book/price, bookstore/book/author

Table 1: Nodes with sample data

Query Q_1=/library/reservation//book illustrates how the data summaries can be used. Q_1, submitted on N_3, asks for all reserved books from all the library nodes in the P2P network (Figure 2). Determining which nodes to send Q_1 to is done as follows: The tag name *book* serves as the DHT lookup key. Q_1 is sent by the DHT layer to N_4 (step 1), which stores the portion of the catalog which contains *book* information. On N_4, the path in Q_1 is matched against $S_{i,book}$, (1<i<4). N_4 replies to N_3 with the node set $\{N_1\}$ (step 2) since only $S_{1,book}$ matches the given query and so N_1 is the only node that stores at least one XML document that contains a *book* element with a *library* ancestor. Finally, N_3 sends Q_1 to N_1 for execution (step 3). The example illustrates one possible way to use DHTs by appropriately defining k_j and S_{ij} for XML repositories.

DHT Index	
N_1	author:{($S_{1,author}$), ($S_{2,author}$), ($S_{3,author}$), ($S_{4,author}$)}
N_2	reservation: {($S_{1,res.}$)}
N_3	--
N_4	book: {($S_{1,book}$), ($S_{2,book}$), ($S_{3,book}$), ($S_{4,book}$)}, price: {($S_{2,price}$), ($S_{3,price}$), ($S_{4,price}$)}

Table 2: Part of the DHT index on each node

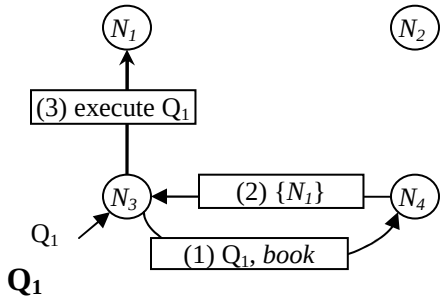


Figure 2: Execution of Q_1

3. Data Summaries

The extraction of the data summaries S_{ij} from the set D_i is the next step in the design process. Considering how the summaries could be used can provide insights into how they should be extracted from the data on each node. Assume that our input queries are XPath expressions and the data on the participating nodes are XML documents. Consider the path expression *//book//price*. Assuming that *book* is chosen as the lookup key for this path, the set of data summaries $\{S_{i,book}\}$ should enable the catalog to find nodes that have documents in which *price* is a descendant of *book*. Consequently data summaries should contain *descendant* information for *book*. Alternatively, if *price* is used as the lookup key, *ancestor* information is required. Besides structural summaries, summarizing values is equally important. Consider the example query *//book*[*category* = "Peer-to-Peer"]. If there are 100,000 nodes that store book information but only 20 of them specialize in "Peer-to-Peer" books, it would be much more efficient to send such a query only to the 20 relevant nodes and not to all nodes that have books.

Flexibility in data summarization is also an essential requirement for our system since there are various data value domains and schemas. There are many different data summarization techniques and naturally none of them is suitable in every case. Consequently, data providers that join the system should enjoy the flexibility of choosing from a variety summarization techniques that fit their data. For example histograms [13] are suitable for numeric values while Bloom filters [3] are more suitable for web addresses. In some cases no summarization is necessary, such as the domain of all states in the USA.

Note a fine distinction between the data summaries provided to the system by a data source N_s and the summaries actually stored on some other node N_c in the sys-

tem. Conceptually, N_c receives a set $\{(k_j, S_{ij})\}$ from the other nodes in the system. However, it is not required to store each S_{ij} as provided by the data source. N_c can store the summary information however it desires, but should not degrade its accuracy or otherwise change its content.

In this paper we evaluate our design on large networks of independent XML data repositories. Two path summarization methods for XML data are presented that are suitable for the purposes of our experiments. For both designs we adopt the choice of keys and summaries made in the initial example (Section 2.2): XML element tags and attribute names are our keys k_j. For each k_j the corresponding data summary S_{ij} stores all possible unique paths to k_j, which, in essence, corresponds to structural ancestor information. During a lookup operation, the last step of an XPath branch is used, which eliminates false positives from the structural summaries. Other configurations are also possible.

The DHT layer distributes the set of keys across the nodes in the system. The catalog $\mathbf{CT_N}$ on node N holds the set of the data summaries for keys assigned to N. In essence $\mathbf{CT_N}$ implements the *map* function for XPath queries for all keys assigned to N. For our study we implemented $\mathbf{CT_N}$ using the methods described below. The performance of these implementations was then measured and the results were used in our simulation experiments.

3.1. Generic B+-Tree

This method for implementing $\mathbf{CT_N}$ is the most generic in the sense that it is both simple and independent of the structure of the data summaries. The keys of the B+-Tree are the pairs (k, N) where k corresponds to the element tag or the attribute name and N is the node that contributed a summary S for k. Only a prefix lookup on k is required but having N in the key helps speed up the algorithm in Section 5. S contains two parts: The structural summary SS and the value summary VS, both of which are optional.

The simplest way to implement SS is to store all unique paths that lead to k in the data of N. The space required for this will be usually small. Let *c* denote the total number of paths that lead to k; let *l* denote the average depth of each path and finally, let *t* denote the size in bytes allocated for storing each tag on the path. The size of SS is then approximately $s = c \cdot l \cdot t$. To make *s* independent of the length of the element names, tags are hashed to 32-bit integers. Using reasonable values of 10, 4, and 4 ([5]), for *c*, *l*, and *t*, respectively, the size of SS will be about 160 bytes. If the size of SS becomes a performance limiting issue in pathological cases (very large *c* or *l*) more advanced summarization techniques ([1], [19]) can be applied. VS depends on the value domain of k. If k has numeric values simple ranges suffice. Bloom filters [3] can be used when we are interested only in equality queries on the value domain of k.

Given a simple XPath query Q: $/a_1/a_2/.../a_n/k$ *op* x the nodes that must receive Q are determined as follows: Each S_i that corresponds to k and node N_i is examined. Q will be sent to N_i only if SS_i contains $/a_1/a_2/.../a_n$ and VS_i *op* x =TRUE (*op* is an operator such as '=' or '<'). As expected, the set $\{N_i\}$ returned by this procedure will contain some false positives, mainly because of the accuracy of the value summary. The structural summary has a very low probability of producing false positives, when a good hash function is used, because hash collisions must occur in every element of a path to produce a false positive. Nevertheless, false negatives can always be avoided, as long as the summaries are updated regularly. Note that SS can also handle XPath queries with wildcards and "//".

One problem arises when the number of B+-tree entries for k becomes large, in which case the system may need to match a path against a large number of potentially very similar structural summaries. The solution is to replace all keys (k, N_i) with keys (k, C_i) that correspond to data items CS_i. C_i corresponds to a cluster of nodes with similar structure and CS_i is a new compound summary constructed by merging the structural summaries of the nodes in C_i. The paths contained in all S_is are merged and to each path *p*, a subset RN is assigned. RN contains the nodes that either contain or don't contain *p*, depending on which choice yields a smaller set. The other alternative for handling large numbers of nodes is to use an index keyed by the path.

3.2. Path Index

A path index is an alternative which is less flexible than the previous method but more efficient for elements that appear on large numbers of nodes. Flexibility is limited because this design requires that the structural summaries contain paths while the previous method works for any structural summary. Processing one summary per node is, however, no longer necessary. Consider again the query Q: $/a_1/a_2/\ldots/a_n/k$. The DHT will relay Q based on k to N_c for the purposes of catalog lookup. On N_c, a B+-Tree keyed on inverse paths will guide the lookup. The data items in the tree are lists of nodes and each node is specially annotated if it has provided a value summary for the specific path. Thus, if the B+-tree contains the key $k/a_n/\ldots/a_2/a_1$, retrieving the corresponding data item will yield the nodes in the system that contain $/a_1/a_2/\ldots/a_n/k$. In order to make the key sizes of the index less variable the tags are replaced with constant size integers using hashing. Processing queries with wildcards is also possible. For example, Q: $//a_1//*/a_n/k$ can be processed by initiating a range scan using the B+-Tree key k/a_n, and evaluating each subsequent B+-Tree key for at least one element of any tag above a_n and an element a_1 somewhere in the path above all the others.

With this structure a node must be associated with each path it contributes and consequently with each corresponding key in the B+-Tree. For example if node N con-

tributes 10,000 different paths for tag k, a reference to N has to appear in 10,000 data items in the B+-Tree. Similarly, if a path is present on 10,000 nodes its associated data item must contain references to all these nodes. Size concerns stemming from these cases can be easily addressed by either clustering nodes with similar documents into groups or by simply compressing the node lists. Our study assumes that a scalable, efficient and reasonably sized index is available on each participating node.

4. System Evolution

System evolution, such as bootstrapping and node arrival and departures, is defined by the Chord protocol. The specification, however, refers only to the keys and not to the objects corresponding to those keys. It is up to the application to manage storage and retrieval of the objects corresponding to the Chord keys. This section describes how the distributed catalog service evolves when nodes join, leave and update their data, and how objects, which are the sets of data summaries, are stored and accessed.

4.1. Arriving Nodes

Section 2 outlines a model according to which nodes create the data they provide to the P2P network. Each new node N_n that joins the system creates the set $\mathbf{C_n}$ which makes the data of N_n query-able by the nodes already in the system. First N_n contacts any node N_c in the system (Step 1, Figure 3). Chord finds N_n's successor N_s in the identifier ring. The new node N_n, now part of the identifier ring, injects each $(k_{nj}, S_{nj}) \in \mathbf{C_n}$ into the system and the Chord protocol decides which nodes should receive the new catalog information (Step 2). Since uploading N_n's catalog information may take some time, not all the data on $\mathbf{C_n}$ becomes immediately query-able by existing nodes in the system. Only after all the summaries in $\mathbf{C_n}$ have found their way to their hosts is N_n's entire data visible to other nodes in the system.

Additionally, N_n becomes part of the catalog infrastructure and is available to share the load of the catalog service by hosting parts of the summary sets already in the network. The Chord protocol will assign to N_n keys k_j for which N_n is the successor in the identifier ring. These keys are located on N_s (k_1 and k_2, Figure 3). Our design co-locates keys and summary sets in the interest of avoiding one extra network message during lookup. Therefore, it follows that when N_n joins the system both keys and data summaries need to be moved from N_s to N_n (Step 3).

Moving the keys from one node to another happens much faster than moving the summaries, because the former are smaller. Thus, each k_j for which N_n becomes responsible initially points to N_s until the corresponding data summary S_{ij} is fully transferred to N_n (Step 2). During this transitional period lookups will be relayed from N_n to N_s. Figure 3 shows the time line of N_n joining the system.

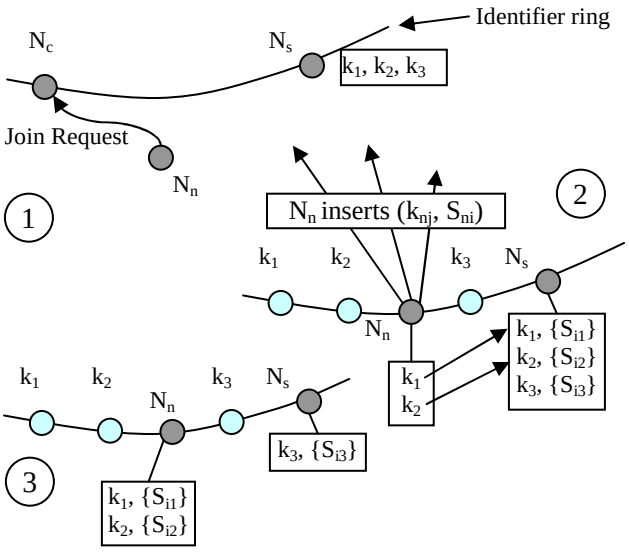


Figure 3: Node N_n joining the network

4.2. Updates and Departures

Catalog information stored in the system may need to be updated as the data on individual nodes changes. Update requests are handled in a similar way as insertion of information during join (Step 2, Figure 3). Updates to summaries follow the "single writer/multiple readers" model; only a node that has created a data summary (the *owner*) is allowed to change its content. Nodes that store the data summaries do not alter their content, although they may alter the way they are stored.

When a node N decides to leave the system, it must hand over the catalog information to its successor according to the Chord protocol. Furthermore, it notifies the nodes that hold N's catalog data. To achieve this, N uses the keys it inserted into the system to find the nodes that currently hold N's catalog information. Another valid approach is one where N does not do anything upon leaving the system and lets the remaining nodes detect its absence over time.

4.3. Expected System Volatility

The *system volatility*, the rate at which nodes join and leave the system, is a key factor to consider when designing such a system. If this rate is expected to be high the design should avoid moving large amounts of data across nodes as part of system maintenance. DHTs generally try to minimize the amount of state stored on each node in order to be able to facilitate any level of volatility. Catalog information, which is moved across nodes, is not large, but its size would probably be an issue in an environment similar to those of other file-sharing systems like Gnutella and Napster, in which there is a high turnover of nodes. Our target, however, is a community of independent data providers that are interested in making their data

widely query-able and therefore are highly unlikely to have the same behavior as individuals sharing music files.

We expect that nodes, having joined, will leave only for scheduled maintenance and then rejoin. In this case they do not need to reinsert their catalog information into the system as they do during the initial join phase. Therefore, the goal is to achieve high throughput of catalog lookup requests and not to mitigate the impact of system volatility by minimizing the amount of catalog data stored on each node. Nevertheless, the dynamic maintenance algorithms of DHT designs are important even in a low volatility environment, since they allow new nodes to join in a scalable way.

5. Processing Catalog Queries

Given an XPath query, a catalog lookup determines which nodes in the system should receive the query. The system can handle general branching XPath queries. Consider the following example which is based on the sample network introduced in Section 2.2 and the query Q_2: //book/[author="J. Smith"]/price. Q_2 retrieves the prices of books by author "J. Smith" and has two branches:

Q_{21}: //book/price

Q_{22}: //book/author ="J. Smith"

Both branches must be satisfied and thus the query will be sent only to those nodes that have both paths in their repositories. Figure 4 shows the steps for each query branch. Q_2 is sent to N_4 (step1), where, based on *price* and Q_{21}, the set $\mathbf{N_{21}}=\{N_2, N_3, N_4\}$ is produced. Q_2 and $\mathbf{N_{21}}$ are then forwarded to N_1, which is responsible for *author* elements (step 2). Q_{22} contains a value predicate on *author*. Hence, both structural and value summaries are utilized to select relevant nodes. Assuming only N_2 has authors named "J. Smith", the lookup using Q_{22} yields the set $\mathbf{N_{22}}=\{N_2\}$ (step 3). Finally, N_3 sends Q_2 to N_2 for execution (step 4), since it is the only node appearing in $\mathbf{N_{21}} \cap \mathbf{N_{22}}$.

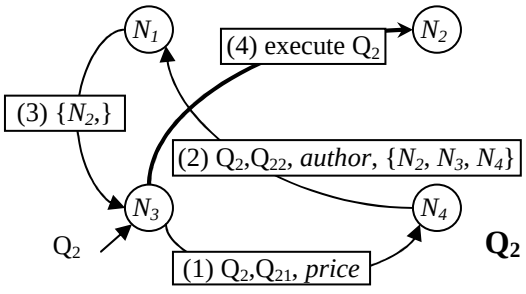


Figure 4: Processing strategy for Q_2

The class of XPath queries that the system can handle are of the form $p = /a_1[b_1]/a_2[b_2]/\ldots/a_n[b_n]$ *op value*. Each b_i is in turn a path. The path steps can include wildcards and the '//' navigation operator. The structural part of the XPath query is handled using the structural catalog information. The value predicate uses the value summaries. The implementation of *query_parts* selects the target tag name of each branch of the query. The algorithm for resolving this general query is as follows:

1: Extract all the N simple paths sp_i from p. Simple paths have the form $sp_i=/a_{i1}/a_{i2}/\ldots/a_{i,Mi}$ *op value* ($1<i<N$). Let the set of candidate nodes be $\mathbf{N} = \varnothing$.

2: Pick the next $a_{i,Mi}$ in the set **T** of targets of the simple paths.

3: Visit the node N_c responsible for $a_{i,Mi}$ and retrieve the set of candidate nodes $\mathbf{N_i}$ for sp_i using the catalog information on N_c.

4: Set $\mathbf{N} = \mathbf{N} \cap \mathbf{N_i}$, or $\mathbf{N} = \mathbf{N_i}$ if $\mathbf{N} = \varnothing$. $\mathbf{T} = \mathbf{T} - \{a_{i,Mi}\}$

5: If $\mathbf{T} \neq \varnothing$ go to 2.

6: **N** contains the nodes on which p should be submitted.

Caching catalog query results or even summaries on nodes that submit queries is feasible. Of course this would increase performance by offloading frequently contacted nodes, but also would raise cache maintenance and invalidation issues. Our study does not deal with caching catalog query results or data summaries since accessing the primary copy of a data summary is guaranteed to fetch the most up to date results. Incorporating caching as an additional layer over the present design is left as future work.

6. Catalog Lookup Scalability Issues

Current implementations of DHTs balance the load across participating nodes by distributing keys uniformly under the assumption that the keys are accessed uniformly. Our system cannot be load balanced by relying solely on the provisions of the DHT layer because some elements will be used in queries more frequently than others. The result will be a higher processing load for catalog queries on the nodes that are responsible for frequently accessed elements, which invalidates the uniformity assumption of key accesses present in DHT designs. Furthermore, it can lead to scalability limits unless the processing load for popular elements is distributed across more nodes in the system. Even if the processing load of catalog lookups on nodes is negligible compared to the processing of the actual queries on each node, such lookups still consume resources such as available connections. This section describes techniques that allow the redistribution of the catalog query load dynamically based on the global query workload. It is assumed that nodes are willing assist with load balancing.

6.1. Structure Based Splitting

The last step in a branch of an XPath query is used as the lookup key for locating the corresponding summary set. When the request rate for a key exceeds a specific threshold, the affected node initiates a key split which forms new keys. This has the effect of transferring part of the load to other nodes. The following example illustrates how this technique works (Figure 5). Consider the tag *price* and for illustration purposes, assume *bike*, *car*, *boat*

and *house* are its possible parent elements. Let N hold the data summaries for *price*. Once N detects that requests for *price* exceed its capacity for catalog requests it initiates a split of *price*. Thus, N creates four new keys: k_1 = *bike*/*price*, k_2 = *car*/*price*, k_3 = *boat*/*price* and k_4 = *house*/*price*. Then N creates a new set of data summaries by appropriately splitting the existing data summaries of *price*. The new keys k_1, k_2, k_3, k_4 and the corresponding partial summaries are handed to the DHT layer and eventually end up on nodes N_1, N_2, N_3 and N_4 respectively. Care is taken that N does not receive any of the new keys. Finally, N has two choices for what to do with the initial data summaries for *price*: it keeps them (*split-replicate*) or it discards them (*split-toss*). Both approaches are valid since, combined, the new summaries contain the same information as the old summaries. Updates to the *price* summary requested by *price*'s owners have to be propagated to the affected new summaries in either case.

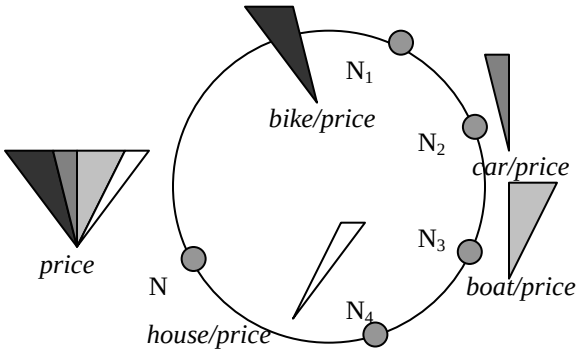


Figure 5: Splitting of price

Once the tag *price* has been split, query processing changes slightly. When some node N_q submits a query q_1=//*car*/*price* < $10,000, q_1 will initially arrive on N which is responsible for *price* summaries. N will then respond to N_q that *price* has been split and that new keys of *level* 2 should be used when submitting catalog lookups. N_q will cache this information and will resubmit the catalog lookup request using the new key *car*/*price*. Eventually all nodes that request *price* catalog information will find out about the split and replace the key *price* with its corresponding *level* 2 key for subsequent queries. The *level* of a key indicates the number of path steps contained in it. Initially all keys are *level* 1.

There is, however, a class of queries that will not be able to use a *level* 2 key. Consider the query q_2=//*store*[*name*="A"]//*price* < $1000 submitted by node N_q in which the parent of *price* is not defined. If N followed the *split-replicate* policy during the splitting of *price*, it will process q_2 itself. If N followed the *split-toss* policy, N_q being aware of this fact will have to submit four catalog queries using the four possible keys and then merge the results from nodes N_1, N_2, N_3 and N_4.

Query q_2 makes one trade-off between *split-toss* and *split-replicate* apparent: data replication versus more messages per query. Another arises in when the split is no longer necessary because of changes in the global workload. Using *split-replicate*, any of the nodes N_1, N_2, N_3 and N_4 can safely discard its *price* summaries when it notices that it does not receive any significant traffic. Using, *split-toss*, however, all nodes involved in the split need to coordinate their actions.

In general, the structure-based splitting (SBS) algorithm takes as input the key k to split and the set K_k = {(p, v, N)} where p is a path that leads to k, v is the value summary associated with k and N is the owner node of the pair (p, v). Note that k itself can be the product of a split. In this case the owner node is the node of the original unsplit key. The output of SBS is the set {(k_i, K_i)} and the mapping (k, {k_i}). If a split is necessary but the key cannot be split, the solution is replication described next.

6.2. Replication

Another method for balancing the catalog query load is to replicate sets of summaries on other nodes. When a node N detects that queries of one of its keys k exceeds a specific rate, it contacts one or more nodes in the system and requests that they replicate the summary data for k. N also creates a mapping for k's new catalog data locations which it hands over to nodes that request lookups on k. A node N_q queries the specified new locations in a round-robin fashion, after it is notified that key k has been replicated.

6.3. Splitting vs. Replication

There are several important differences between the two methods of load balancing. Replication is oblivious to the content of the data summaries and works with any summarization method. SBS exploits the structure summaries to create as many partitions as there are level 2 keys. Thus if there are *a* level 2 keys which contain the key k being split, the initial request *r* rate is roughly divided by (*a* + 1) in the case of *split-replicate*. Furthermore, if a new partition on a node is not receiving a significant rate of requests it can be safely discarded, if *split-replicate* is used. On the contrary, plain replication cannot, a priori, decide the optimal number of new replicas. One choice is to create only one replica on one other node in the system and let the new node replicate further if the rate of requests is still too high. The other choice is to create many replicas at once, which will cause the system to react to the increased load more rapidly. Another difference is that *split-replicate* replicates the information only once per split. Also, a single update in the case of SBS needs to be propagated to only one other summary per split, whereas in the case of replication it must go to all replicas. Problems such as those described in [11] are not an issue since only the owner of a data summary set requests updates to catalog information. In any case, replication is the last resort for load balancing if no further splitting is possible.

7. Experimental Results

The goal of our experiments is to demonstrate the viability of our approach. Thus, certain traits of the DHT designs are taken for granted. For instance we do not evaluate how our system works under various degrees of volatility, or how different parameters of the underlying DHT affect the number of messages in the network. The focus is to facilitate scalable catalog lookups during query processing. A simulator is used to verify the scalability of catalog lookups in very large networks.

7.1. Experimental Data

Our goal is to use data that will be very similar to the data one could expect in a real world deployment of our catalog service. This can be achieved using data from potential data providers that would find the catalog service useful. Additionally, we opt for using XML data since this is the de facto standard for platform independent data representation and information exchange. The website www.xml.org contains a registry in which organizations that use XML data can register their schemas. We believe that this data is very close to reality and therefore we use it in our experiments. We found 30 usable schemas that draw from various domains such as Banking, Transportation, Arts and many others. The data collection contains 3500 unique element and attribute names and 16,000 different paths from root elements to leaves.

7.1.1. Data Generation

The purpose of data generation is to assign the available schemas to the nodes in the system. Each node N is assigned a number of schemas NS which is drawn from a normal distribution with mean MNS and standard deviations SNS. Each schema is chosen with probability proportional to a weight WS. Changing WS allows regulation of the popularity of each schema. For each chosen schema we create a node-specific schema as follows: All the paths that are required in the schema are also included in the node specific schema. The paths that are not required are included with probability PP, in order to create a more realistic situation.

Each node must create data summaries to insert into the catalog service. To this end each possible path that leads to *leaf* is enumerated and, for each element and attribute found in the node's assigned schemas, a structural summary is built. Note that commonly used element names that appear in multiple schemas will contain paths from different schemas in their summaries. In our study we ignored namespaces.

7.1.2. Query Load Generation

The query load is generated concurrently with data generation and is done in such a way that makes the generated queries follow the distribution of the generated data across all nodes. Each schema is assigned *query credits* QC which is proportional to WS. While paths are enumerated, each path is picked with some probability QP to contribute to the query pool. For each chosen path p the size of the XPath query it yields is between two and four 80% of the time (a similar assumption is made in [1] and [19]). The other 20% of the time the size of the query is uniformly distributed between one and the size of the path p. The target of the XPath query is chosen to be one element or attribute of path p as follows: The target is the eth item from the leaf with e exponentially distributed with mean 2. This gives preference to tags closer to the leaf nodes, which is more realistic. Once a query is created the QC corresponding to the schema is decreased by one. Thus, popular schemas contribute more queries to the query pool than less popular ones.

7.2. Simulator Architecture

The simulator used for evaluating our catalog service is based on the Chord protocol simulator found on the Chord project website. The DHT substrate is used unmodified and contains some additional optimizations (such as LRU finger tables) not found in [25]. These optimizations further decrease the number of network messages required but do not affect our study. Each message between nodes incurs a network delay exponentially distributed with an average of 50ms. Nodes do not unexpectedly die and so all nodes that join are available for catalog queries. The average local processing time for Chord maintenance tasks is uniformly distributed between 50 ms and 200 ms.

An additional layer translates an XPath query Q to the necessary Chord lookups, based on the algorithms described earlier. Once the query reaches the node with the appropriate summary it is evaluated and the set **N** of candidate nodes that match the structure of the query is produced. To simulate the effect of value summaries a percentage (CT) of the nodes in **N** are discarded before **N** is sent back to the query origin. Upon receiving **N** the origin sends the XPath query to the nodes found in **N.**

Catalog related processing is simulated in more detail than Chord specific tasks. Each node simultaneously serves up to CR catalog requests using the processor-sharing discipline [14] while additional requests are queued in a FCFS queue that can hold at most CQ requests. The time it takes to process each catalog request is 101 ms on average and follows the distribution of the measured times. Measurements were taken on a Pentium III at 800 MHz, running Linux 7.2 with an IDE disk on implementations of the structures presented in Section 3 using the data generated. These numbers represent the time it takes if a request is processed alone and are increased according to the processor-sharing discipline. The time for XPath queries was set to be uniformly distributed with a mean of 500 ms and a standard deviation of 500 ms and represents the background load on the system.

The system is driven by simulated users, who pick a query from the query pool and submit it on a random node

in the system. The number of users U is a multiple of the number of nodes in the system so that the system becomes loaded. Each user submits a new query after all generated XPath queries have finished executing and after a think time of 5 seconds and a typing time of 3 seconds as specified in the TPC benchmark specifications [26]. Note that by setting the CT parameter to 100% no XPath queries are generated and thus the system runs in catalog-only mode. This setting simulates the case where each data provider uses a dedicated catalog processor or assigns a maximum bandwidth to catalog queries (in conjunction with appropriately setting CQ).

The discussion on load-balancing left the triggering of splitting or replication unspecified. Our opinion is that in a real system the rate of requests that will cause a load balancing reaction is a local decision. For our simulations however, the nodes in the system are all equivalent and there is a common policy for all of them. A load balancing action on a node is triggered once the number of requests has reached CR and new requests are put on the FCFS queue. This indicates that the node is receiving more requests than desired. To remedy the situation a key must be either split or replicated. The key chosen is the most frequently occurring key in catalog queries among the CR requests that are served using Processor Sharing. The load incurred by a load balancing action is also simulated.

Number of Nodes	Parameters
All Setups	MNS=5, SNS=5, PP=90%, WS=10 (for all schemas), QP=80%, CT=100%, CR=20, CQ=500
500	U=5,000, NQ=400,000
1000	U=10,000, NQ=800,000
2000	U=20,000, NQ=1,600,000
3000	U=30,000, NQ=2,000,000
5000	U=50,000, NQ=4,000,000

Table 3: Experimental parameters

7.3. Experiments

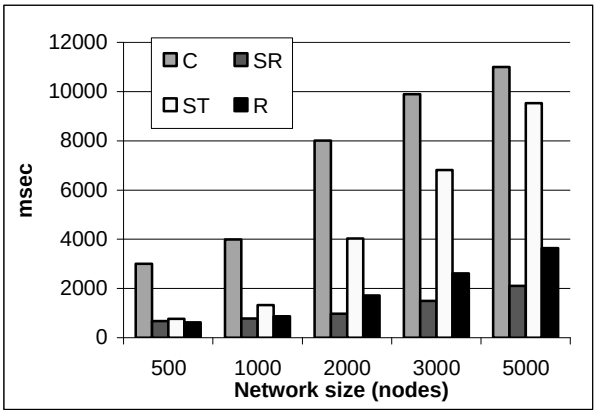
In our experiments we focus on processing catalog queries in a system that has formed, stabilized. Updates are not considered since the update traffic is assumed to be orders of magnitude less intense than the query traffic and so does not affect load balancing. Furthermore, the performance of data summarization techniques is not tested since it is not the focus of this paper. The structural summaries used for the experiments are 100% accurate. The scenario of the experiments is as follows: Users start submitting queries after thinking and typing. They submit a new query after the previous one has finished. The activity stops after NQ number of queries have been submitted and finished. The main metrics examined are the average throughput for catalog queries and the average response time for each catalog query. The distribution of the catalog requests across nodes as well as the number of load balancing actions are also of interest. We test catalog query processing under the following different settings:

1. No load balancing (**C**)
2. Split-replicate load balancing (**SR**)
3. Split-toss load balancing (**ST**)
4. Replication (**R**) one replica at a time

The values for the parameters described previously are taken from the Table 3.

7.3.1. Performance

This section presents the performance of the catalog service for various settings.



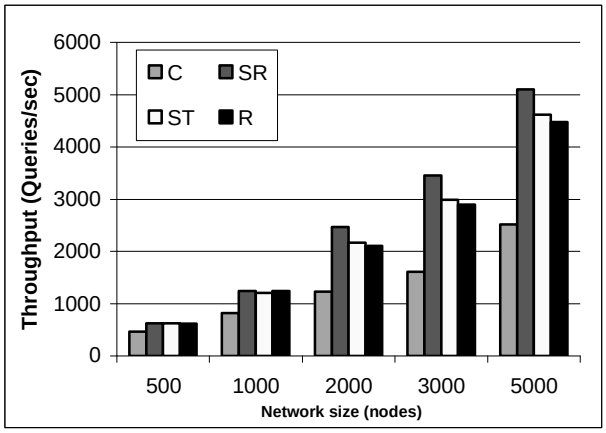
Graph 1: Average response times for catalog queries

The response times measured show that using the DHT alone, without any provisions for adapting to the query workload does not scale (Graph 1). (All measured 95% confidence intervals are within at least 0.5% of the means).

Number of Nodes	500	1000	2000	3000	5000
Average hops per request	3.5	4.1	5.0	5.7	7.0

Table 4: Average number of network hops per request

SR shows the best scalability characteristics when adapting to the query load, holding the average response time below 2.1 sec. The increase in response times for SR can be attributed to an increase in the average hops per catalog request (Table 4) and to the increased load because of insertions caused by splits. ST does not scale as well because it discards the split summary set. This causes the generation of more messages per query and more splits (Graph 6), thus increasing response time. Replication performs better than ST but worse than SR since it cannot adapt quickly to the query load because only one replica is created each time a node reaches the request limit CR. SR splits a key as many times as there are next level keys and so distributes the overload faster. In both SR and ST there were no cases where splitting a key was not possible.



Graph 2: Combined throughput of queries

The throughput results in Graph 2 once again show that Chord alone cannot handle the increasing rate of requests as the system grows because some nodes are overloaded. In the 500 node case SR, ST and R achieve the maximum throughput rate which is 624 queries/sec. To achieve this, only a very small number of splits and replications were required. SR also works very well for all five sizes of networks. Both ST and R do not perform as well as SR in the large networks of 2000, 3000 and 5000 nodes. The performance of R vs. ST is, however, reversed relative to the response time numbers. The explanation is that, on average, ST generates more requests and splits. However, it reacts to the increased rate on the large networks faster than R because more new keys are generated per split. Table 5 shows the number of catalog lookups rejected by nodes in the system because they were overloaded. Some individual nodes drop more than 50% of the requests if no load balancing takes place. In any case, using SR provides double the throughput of C without any significant loss of requests and with at least a five times improvement in response times.

Network sizes	C	SR	ST	R
500	2%	0%	0%	0%
1000	4%	0%	0%	0%
2000	8%	0%	2%	0%
3000	11%	0%	4%	0%
5000	18%	1%	6%	1%

Table 5: Droped requests across all configurations

7.3.2. Data Summary Sizes

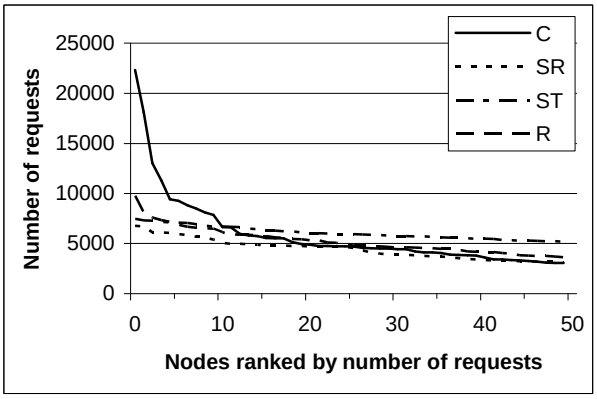
The increased performance observed using load-balancing comes at a very low cost in terms of storage using the path index described in Section 3.2. The average size of the catalog on each node prior to load balancing actions is about 12KB-15KB (for all approaches). The maximum size of the catalog information on any node for various network sizes is shown in the following table

network size	500	1000	2000	3000	5000
Size (KB)	165	247	495	740	1135

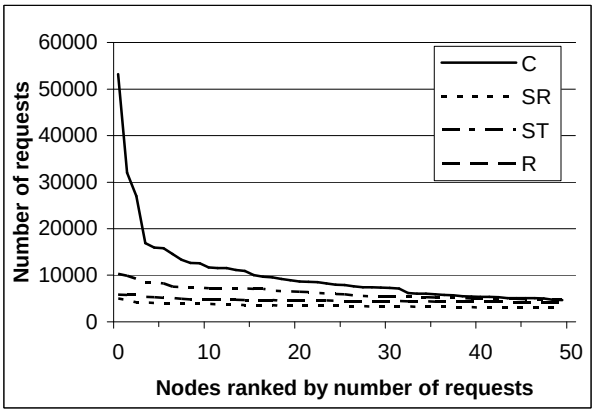
Using SR and R the average catalog size increases by roughly 2KB, while the maximum catalog size on a node does not change. ST naturally does not alter the average catalog size with respect to C. All load-balancing techniques require that each node stores a map M with the new key mappings for keys that were split or replicated. In the case of 5000 nodes the size of M is 35 KB and 6 KB for SR and R, respectively.

7.3.3. Request Load Distribution

Graphs 3, 4, and 5 show the distribution of the requests across the nodes in the system for configurations of 500, 2000 and 5000 nodes (1000 and 3000 networks are omitted for brevity). The nodes in the system, ranked by the number of requests received, are on the X axis. Each graph contains the first 50 nodes which receive roughly 34% of all requests in all configurations when no load balancing is used.



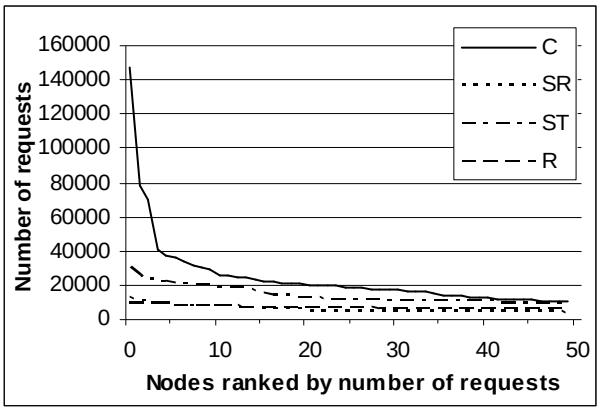
Graph 3: Request distribution (500 Nodes)



Graph 4: Request distribution (2000 Nodes)

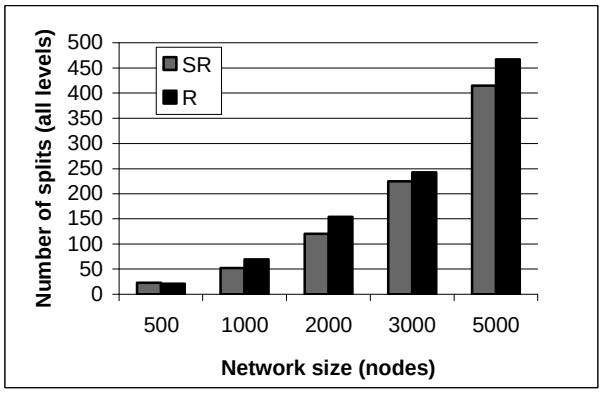
Obviously, if no actions are taken, the distribution of requests across nodes is not balanced. Therefore load balancing is necessary for scalability. Using splitting and replication the system achieves a fairer distribution of

requests which reflects in better throughput and response times (Section 7.3.1).



Graph 5: Request distribution (5000 Nodes)

Using load balancing on the 500 node network leaves the first 50 nodes handling about 27% of the query load. While not ideal, the redistribution seems adequate to handle the overall rate of requests, which is 624 queries/sec. If the rate were higher more load balancing actions would distribute the load more evenly. This is evident in the larger network configurations. The higher request rate causes more splits and replications and thus the first 50 nodes receive only about 11% and 7% of the users' requests in the 2000 and 5000 node networks respectively. It is important to note that strategy ST generates about 1.5-2.5 times more catalog requests than SR because it discards the split summary.



Graph 6: Number of load balancing actions for SR (all levels) and R

As expected the network of 500 nodes required the fewest splits since the nodes in it are subjected to the lowest request rate of all networks. Furthermore, as stated, the ST strategy incurs more splits than SR, since in the former, the initial summary is discarded and all the requests that could be answered using the initial summary go to the split replicas. This causes higher level splitting. In the 5000 node case ST causes almost twice as many splits as SR. Graph 6 shows the increasing number of load balancing actions for SR and R as the networks grow larger. In order to keep up with the request rate the system adapts to the query load.

The number of new keys created by the load balancing methods is an indicator of how fast each policy reacts to the query load characteristics. Consider the case of 2000 nodes: R causes 149 replications. This corresponds to 149 new keys. SR causes just 111 splits but creates 1793 new keys, which balances the query load faster. The small number of splits relative to the total number of elements and attributes clearly shows that dynamic splitting is preferred to splitting keys in advance. Note that in the 5000 node configuration, SR creates 5300 new keys with only 415 splits. Cascading effects were also observed when nodes became overloaded as a result of accepting new keys. Cascading, however, stopped once requests were dispersed across a sufficient number of nodes.

To conclude, our experiments indicate that SR is the most suitable strategy for load balancing a distributed catalog based on our framework. It achieves good scalability, fast reaction to the query load characteristics while incurring a small overhead for creating new keys.

7.3.4. Alternate Query Load

We experimented with a different query workload in which elements at all levels are the targets of paths with equal probability. The results are similar to those obtained with the previous workload which favors deeper paths. As expected the distribution of the load is slightly less skewed leading to a smaller number of load balancing actions. The relative performance of C, SR, ST and R remains the same.

8. Related Work

Distributed databases ([18], [28]) use catalogs to store fragmentation information. Each site has its own replica of the catalog and determines where to execute a query or parts of a query. The fact that all the sites of a DDBMS are under the control of a single authority makes this design feasible since the number of nodes in the system is not large. The problem our design tries to address is different in that there is no central authority, no controlled data fragmentation and a much larger number of data providers. More recent distributed query processor designs such as the one in [4] recognize the need for scalable catalog services and advocate distributed catalogs. A distributed catalog proposal based on multiple hierarchies can be found in [20]. This approach is not automatic since it relies on manual extraction of categories from the data.

Our work builds upon recent peer-to-peer DHT protocols that were inspired by earlier pioneers such as FreeNet and Gnutella ([8], [9]). They guarantee a definite answer to a lookup query within a bounded number of network hops, while Gnutella and FreeNet and other peer-to-peer studies opt for returning the first 10 or 100 matches to a query, if any, are found without any guarantees. The difference of the earlier proposed LH* [16] from current

DHT designs is that it allows a hash table to grow by expanding on servers taken from a large preexisting pool, rather than allowing nodes to join and become part of an existing distributed hash table.

Distributed file systems based on DHT protocols can be found in [7] and [23]. All those studies recognize the need for performing additional load balancing on top of the DHT layer to achieve scalability. They resort to replication of popular files across nodes in the system, assuming that sufficient storage is available.

The study in [12] advocates traditional query processing techniques over data stored in the DHTs. In contrast, our approach uses the DHTs for storing metadata, which it uses to guide queries to the relevant data sources.

Other studies of peer-to-peer systems ([6], [30]) use metadata on each peer to efficiently route searches to other peers or answer searches on behalf of other peers in the network. Their query satisfaction criterion (first 100 results render a query satisfactorily answered) is, however, geared more towards the need of file sharing individuals.

Sun's JXTA Search [27] provides searches of data sources that actively produce data (such as news sites). The system builds indices on the queries a data source can answer and distributes them across JXTA hubs to which data sources connect. Thus, the way catalog information is distributed across the hubs is determined by where the data providers connect. It is anticipated that providers of similar topics will connect to the same hubs. In our case catalog information location is independent of where providers join the system.

DNS [2] is a distributed hierarchical catalog service which is widely used. The naming service it provides is very a similar concept to our mapping of queries to data repositories. The analogy is that our system identifies relevant servers from a query instead of a symbolic name.

9. Conclusions

We presented a design based on established technology that allows the implementation of completely decentralized catalog services for large numbers of nodes. In addition to leveraging existing technology we identified application-specific circumstances that require enhancements in the form of load-balancing in order to achieve scalability. Using simulation we demonstrate that our approach is valid and has good scalability characteristics.

References

[1] A. Aboulnaga, A. R. Alameldeen, J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. VLDB 2001.

[2] P. Albitz, C. Liu. DNS and BIND. (4th Ed.) O'Reilly and Associates, 2001.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors, Communications of the ACM, July 1970.

[4] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. VLDB Journal 10(1): 48-71 (2001).

[5] B. Choi, What are Real DTDs Like. WebDB 2002.

[6] A. Crespo, H. Garcia-Molina. Routing Indices for Peer-to-Peer Systems, ICDCS 2002.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica. Wide-area cooperative storage with CFS. SOSP 2001.

[8] FreeNet: http://www.freenetproject.org

[9] Gnutella Resources. http://gnutella.wego.com/

[10] L. Galanis, Y. Wang, S. R. Jeffery, D. J. DeWitt. Processing Queries in a Large Peer-to-Peer System. CAiSE 2003 (to appear)

[11] J. Gray, P. Helland, P. O' Neill, D. Shasha. The Dangers of Replication and a Solution. Readings In Database Systems, 3rd edition p372.

[12] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. IPTPS '02.

[13] Y. Ioannidis, V. Poosala. Histogram-Based Solutions to Diverse Database Estimation Problems. Data Engineering Bulletin 18(3).

[14] L. Kleinrock. Queueing Systems, Volume 1: Theory, John Wiley & Sons, New York, 1975.

[15] J. Kubiatowicz et al. OceanStore: An Architecture for Global-Scale Persistent Storage. In Proc. ASPLOS 2000.

[16] W. Litwin, M. Neimat, D. A. Schneider: LH* - Linear Hashing for Distributed Files. SIGMOD Conference 1993: 327-336

[17] Napster. http://www.napster.com

[18] M. T. Özsu, P. Valduriez. Principles of Distributed Database Systems, Second Edition. Prentice-Hall 1999.

[19] N. Polyzotis, M. N. Garofalakis. Structure and Value Synopses for XML Data Graphs. VLDB 2002.

[20] V. Papadimos, D. Maier, K. Tufte. Distributed Query Processing and Catalogs for Peer-to-Peer Systems. CIDR 2003

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Schenker. A Scalable Content-Addressable Network. in Proc. of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications.

[22] A. Rowstron, P. Druschel, Pastry. Scalable, distributed object location and routing for large-scale peer-to-peer systems. IFIP/ACM Intl. Conference on Distributed Systems Platforms.

[23] A. Rowstron, P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. SOSP 2001.

[24] S. Saroiu, P. K. Gummadi, S. Gribble. Exploring the Design Space of Distributed and Peer-to-Peer Systems: Comparing the Web, TRIAD, and Chord/CFS. IPTPS '02

[25] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In Proc. SIGCOMM 2001.

[26] TPC-C Benchmark Standard Specification Revision 5.0.

[27] S. Waterhouse. JXTA Search: Distributed Search for Distributed Networks. White Paper http://search.jxta.org

[28] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Linsey, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, R. Yost. R*: An Overview of the Architecture. IBM Research Report RJ3325.

[29] XML Path Language (XPath) 2.0 http://www.w3.org/TR/xpath20/

[30] B. Yang, H. Garcia-Molina. Efficient Search in peer-to-peer networks. In Proc. ICDCS 2002.

[31] B. Yang, H. Garcia-Molina. Designing a Super-Peer Network, In Proc. ICDE 2003.

[32] B. Y. Zhao, J. Kubiatowicz, A. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. UCB Tech. Report UCB/CSD-01-1141.