The Semantic Music Store: Managing Distributed Semantic Overlay Networks

Alexander Löser, Kai Schubert, Frederik Zimmer

Technische Universität Berlin, CIS, Einsteinufer 17, 10587 Berlin, Germany
[aloeser, schubi, tristan]@cs.tu-berlin.de

Abstract. Widespread access to the Internet resulted in the explosion of digital music. People continue to download large collections of music files often stored in directory structures classified by artist or genre. A distributed catalog, scalable to a large number of users, storing classification information of each user would prove very useful. Users may search the catalog to identify other users sharing a particular music genre, decade or artist. We provide a distributed catalog storing taxonomy-based summaries of music files each user is storing. The catalog is based on distributed hash tables and will scale for large music classification taxonomies as well as a large number of user profiles.

1 Introduction

No one questions that music piracy has eaten into profits of the music industry. However, domain experts claim, all the attention paid to the moral and legal issues surrounding copyright protection has allowed the music labels to avoid taking responsibility for violating a cardinal principle of business: Give the customers what they want [10]. Users should be able to buy music in a form that suits their modern lives, e.g. to download single tracks of music on demand from their computer at home or their internet connected MP3 player. They should be able to identify other users sharing the same music flavor and profit from their recommendations. An advanced music classification could allow users to browse the available genres and discover new music as they find genres that describe their own musical tastes. Scalable peer-to-peer networks in connection with actual Digital Right Management technologies could help to build such a large community of world wide distributed music aficionados. Unfortunately current peer-to-peer systems are not able to provide capabilities for rich semantic queries based on existing music classification hierarchies, such as *Give me all symphonies from Beethoven* or *Give me music classified, as Latin/Brazil/Bossa Nova from the 1990ies* or *Which users share which Bossa Nova and Jazz Music*. However, current music semantics, stored in IDv2/IDv3 tags in MP3 files, are rich enough to allow different classification hierarchies. Crespo et.al [3] proposes, that query routing in music sharing networks should be influenced by content. In his approach each user provides a summary of its MP3 files in a semantic description. This description is based on existing music classification taxonomies[1] and allows a classification of the users music for *style*, *artist*, *composer* and *decade*. For the classification of music files into existing taxonomies they utilize

[1] E.g. *allmusic.com* or *musicmoz.org*.

tools[2] that classify the documents, given their keywords, into hierarchical structured categories. In the example in Figure 1 six nodes provide a description of their content, e.g. node A, B and F provide the following music: $A \longrightarrow$ */Style/Latin/Brazil/Bossa Nova*, $B \longrightarrow$ */Style/Latin/Brazil/Bossa Nova* $\wedge$ */Decade/1990* and $F \longrightarrow$ */Composers/Beethoven* $\wedge$ */Style/Classic/Symphonies*. Thus, semantically related nodes form a Semantic Over-

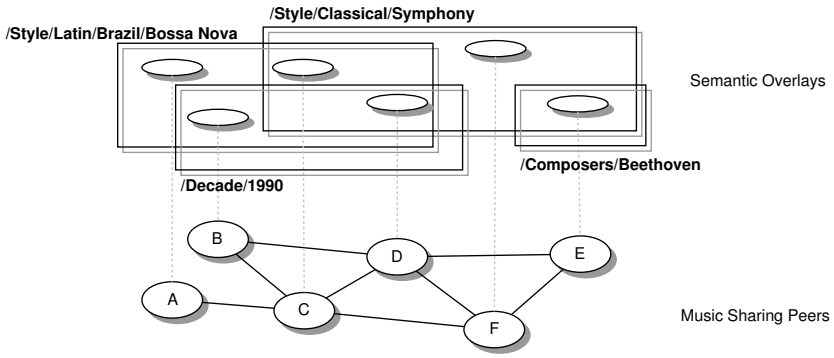

Fig. 1. Semantic Overlays of Music Sharing Peers

lay Network (SON) over the physical network structure. To illustrate, in Figure 1 node A,B,C,D form the */Style/Latin/Brazil/Bossa Nova* overlay, node B,C,D form the */Sytle/Classics/Symphonies* overlay and so on.

Query routing in Semantic Overlay Networks is a three fold process: First a subset of peers providing the right content classified with the right taxonomy path is identified. Then the query is forwarded to this subset of peers. Third, locally each peers music files matching the query are returned. Almost as important, nodes outside the SON (and therefore unlikely to have answers) are not bothered with that query, freeing resources that can be used to improve the performance of other queries. Semantic Overlay Networks provide two main main advantages:

- Queries are only routed to peers providing possible results
- Queries could be narrowed or broadened along classification hierarchy

However the authors of Semantic Overlay Networks gave no detailed information about the efficient realization and computation of such overlay structures. Obviously, nodes must be able to submit their description to an index, so users could query the index and find suitable nodes. Since in music sharing P2P networks no form of centralization is desired, the index should be completely distributed over a set of interconnected nodes in a symmetric network. Another advantages of a *distributed index* is to share costs for bandwidth and computing power for processing among the indexing nodes in the network maintenance of the index. We identified the following major challenges when constructing such an index:

[2] Such as *Autonomy.com*, *Semio.com*

– Hierarchical semantic relations need to be stored and queried in the index.
– The index should be scalable for a large number of nodes.
– Storage and query load balancing strategies are needed, thus nodes may register and queries may be posted on an arbitrary node of the index.
– Since nodes of the index may fail the index should provide replication mechanisms.

Motivated by the recent advantages of scalable location and routing protocols based on distributed hash tables (DHT), we study how to implement a distributed index for SONs on top of the CHORD [17] protocol. However, our findings and algorithms are not CHORD specific and could be used with any other DHT based system, such as CAN, P-Grid, Pastry or Tapastry. Our main contributions are:

– Introducing the basic architecture for storing and querying SONs and define node models and user queries (section 3)
– Presenting algorithms for storing taxonomies and node Ids in DHTs (section 4)
– Presenting distributed matching and query strategies (section 5)
– We present our load balancing strategies (section 6)
– We simulate our approach and discuss the results (section 7)

2 Related Work

In the database community catalogs to select relevant data sources for a given query have been investigated recently. However, most of the approaches assume a central broker architecture [13, 6], which does not scale well or an asymmetric broker topology [9, 4], which is costly and hard to maintain. *www.ondemanddistribution.com* sells and promote music through a diverse set of on-line retailers to find new channels for music sales. However, their content management approach is completely centralized, what may result in a possible bandwidth bottleneck and high communication costs.

To our best knowledge, distributed catalogs for data source selection based on a symmetric topology have been investigated only by *Galanis* et.al [7]. The approach stores attributes of query schemas in a DHT. The idea is interesting, however a search is limited to exact matches of schema attributes. Content specific attributes, such as taxonomy classifications, as well as tree-based searches are not considered.

More closely to our approach comes *INS/ Twine*[1], a resource discovery system, extracting prefix subsequences of attributes and values, called 'strands' from resource descriptions and indexes the hash values for each of these strands in a peer based resolver. In contrast to our approach the system has the disadvantage that it assumes all resources are equally requested and stores descriptions of resources redundantly in the network and resource and device information are stored redundantly on all peer resolvers that correspond to the numeric keys.

Recent proposes [19] a first step towards supporting complex queries over DHTs. They proposed more efficient methods to process multi-attribute queries and proposed the notion of range guards, which can significantly improve the performance of range and join queries.

3 System Architecture

In this section we describe the architecture of the catalog based on a Super-Peer network and give example user queries and examples for results. Figure 2 shows the basic architecture of our approach. It consists of two main components: *peer nodes* register-

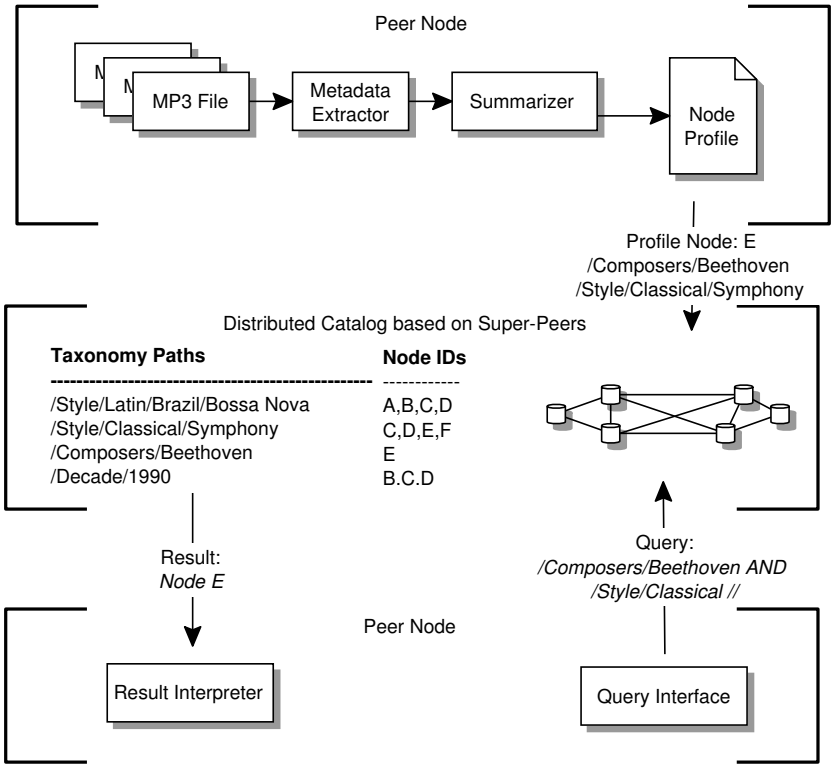

Fig. 2. Generell Architecture

ing its model and issuing queries and *super-peers* managing incoming peer registrations and answering queries:

Peer Node Each peer node may register, deregister and query the index. For registering the index the peer node extracts meta data from its mp3 files. Current music meta data, such as ID3v2 [3] tags, already provide a week genre classification. We assume, that each mp3 file includes a rich genre description, including a classification in a music genre taxonomy based on a semi-structured XML description, see the following example in the left side. This example could be extended to other meta data included already in the ID3v2 standard such as artist, title, album and so on. For

[3] http://www.id3.org/

simplifications we only show the style, decade and composer description.

Sample File Description:

```
<MP3File>
    <File>
        Jobim_Corcovado.mp3
    </File>
    <Decade>
        <1990/>
    </Decade>
    <Style>
        <Latin>
            <Brazil>
                <Bossa Nova/>
            </Brazil>
        </Latin>
    </Style>
</MP3File>
```

Sample Node Description:

```
<Node>
    <ID>D</ID>
    <Decade>
        <1990/>
        <Num>33</Num>
    </Decade>
    <Style>
        <Latin>
            <Brazil>
                <Bossa Nova/>
            </Brazil>
        </Latin>
        <Num>21</Num>
    </Style>
    <Style>
        <Classical>
            <Symphony/>
        <Classical>
        <Num>3</Num>
    </Style>
</Node>
```

The summarizer analyzes the the semantic file description of each file and summarizes them into a node profile. E.g. see the example node profile from above on the right side., it consists of the node ID, the decade and the number of music files related to this decade as well as the style and the number of music files related to this style. In this example 33 mp3 files for the /decade/1990, 21 files for /Style/Latin/Bossa Nova and 3 for /Style/Classical/Symphony are shown. We use this information to decide if a node has a particular large collection on a specific topic.

The profile of each node is registered in the distributed catalog. A peer node can query the distributed catalog for a particular style or decade. To search for hierarchical data stored in the catalog, we need to specify broad queries that can match multiple descriptions. We use a subset of the XPath XML addressing language, which offers a good compromise between expressiveness and simplicity. XPATH allows to specify and select parts of trees. An XPATH expression contains one or more location steps separated by slashes. Predicates are generally specified between brackets. An XML document matches an XPATH expression when the evaluation of the expression yields a non-null object.

The following query selects *nodes providing music from the 90ies on Bossa Nova*:

$$q1 = /Node[[Decade/1990][Style/Latin/Brazil/BossaNova]]$$

Furthermore the ancestor/descendant operator // matches all children. The next query selects *nodes providing classical music*:

$$q2 = /Node/Style/Classic//$$

The result of a query is a set S_i of node Ids matching the query, e.g. $S_{q1} = (B, D)$ and $S_{q2} = (C, D, E, F)$. The exact query processing strategies are explained in section 5.

Super Peer Node A super-peer is a node that acts as a centralized server to a subset of clients, e.g. weaker peer nodes. They introduce hierarchy into the network in the form of super-peer nodes, peers which have extra capabilities and duties in the network. E.g. they stay significantly longer in the network than other peers and provide high processing and bandwidth capabilities. Super-peers are connected to each other as peers in a pure system are, routing messages over this overlay network. Peer nodes register its profile to a super-peer node. Thus, the catalog of peer profiles is distributed among a significant smaller number of peer nodes. Peer nodes submit queries to their super-peer node and receive results from it, as in a hybrid system. In our architecture each peer node may select itself to become a super-peer node. Observations in music sharing peer-to-peer networks have shown, that one peer of 500-1000 peers chooses to become a super-peer node [8].

4 Distributed Storage of Taxonomies and Peer Profiles

In this section we give an overview on storing node ids and taxonomies in distributed hash tables. We show how to store key value pairs and hierarchical relations are stored in a distributed hash table. Further we discuss challenge when integrating different taxonomies.

4.1 Storing Peer Profiles in Distributed Hash Tables

Distributed Hash tables have very desirable characteristics. Their goal is to provide the efficient location of data items in a very large and dynamic distributed system. Like a centralized hash table a distributed hash table stores hash keys to object values. In this paper a hash key is generated by hashing a particular taxonomy path using a consistent hashing function, such as SHA-1. In the following example on the left side the hash table for the example scenario in Figure 1 is shown. The object value of a hash key represents a set of peers classified with this taxonomy path. To illustrate the example on the right side the non hashed values for the hash keys are presented.

Hash Key	**PeerID**
$EA66	A,B,C,D
$CCEE	C,D,E,F
$AB55	B,C,D
$6520	E

```
$EA66 /Style/Latin/Brazil/Bossa Nova
$CCEE /Style/Classical/Symphony
$AB55 /Decade/1990
$6520 /Composers/Beethoven
```

A distributed Hash Table (DHT) divides the hash table into a number of different parts stored on different network nodes. As an example DHT we will use the CHORD [17] protocol. 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. Each key is assigned to its successor node, which is the nearest node travelling the ring clockwise.

Nodes and keys may be added or removed at any time, while Chord maintains efficient lookups using just O(log N) state on each node in the system. The complete hash table will be stored as a distributed hash table, in our approach in the CHORD system. Using the example above Figure 3 shows a Chord ring with four nodes. Each node SP1..SP4 represents a super-peer. Keys are stored clockwise at the closest node with the next highest hash value, e.g. $AB55 is stored at node SP2 since its node ID is $C0BB and is the closest node ID clockwise. Often two or more profiles are classified using the same taxonomy path. E.g. in Figure 3 at the key $EA66 $/Style/Latin/\ Brazil/BossaNova$ four peer nodes are registered: A,B,C,D. In the DHT this is reflected by building an inverted index of the corresponding PIDs at the corresponding hash key, where the hash key represents the path value and the object value peer classified at this path.

4.2 Storing hierarchical Data Structures in Distributed Hash Tables

To represent the tree structure of the taxonomy for each path so called *tree structured sub sequences* are computed and hashed, e.g. for the $/Style/Latin/Brazil/BossaNova$ and $/Style/Classical/Symphony$:

```
$EA66 /Style/Latin/Brazil/Bossa Nova
$AB33 /Style/Latin/Brazil
$3A56 /Style/Latin
$FF7A /Style

$CCEE /Style/Classical/Symphony
$BA7F /Style/Classical
```

Please note, that in this example the path $/Style$ is computed only once, but used in both paths. For representing the *successor relations* between different keys we store in the object value a *key-to-query relationship* represented this with the $SUCC$ symbol. E.g., the key $BA7F, that represents $/Style/Classical$, has in our example one successor, $/Style/Classical/Symphony$. For the above mentioned in the Table 1 key and object values, as well as key-to-query relationships, are shown. Figure 3 shows

Hash Key	**Object Value**
$FF7A	SUCC /Style/Classical
	SUCC /Style/Latin
$BA7F	SUCC /Style/Classical/Symphony
$CCEE	C,D,E,F
$3A56	SUCC /Style/Latin/Brazil
$AB33	SUCC /Style/Latin/Brazil/Bossa Nova
$EA66	A,B,C,D

Table 1. Key-to-Query Relation Ships for /Style/Latin/Brazil/Bossa Nova and /Style/Classical/Symphony

the successor relations from the example above in the CHORD Topology distributed among the catalogue nodes.

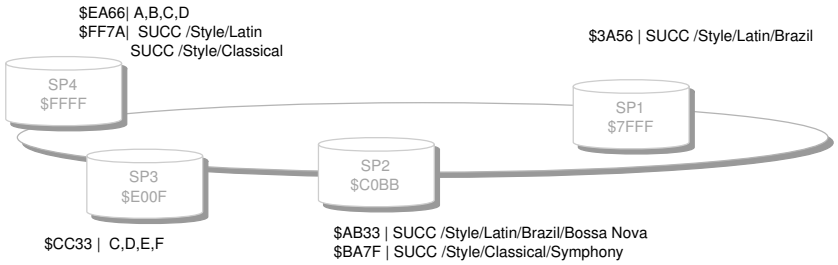

Fig. 3. Storage of Taxonomies in a DHT using Key-to-Query References for /Style/Latin/Brazil/Bossa Nova and /Style/Classical/Symphony

4.3 Challenges when Integrating Heterogenous Taxonomies

An important preliminary of our approach is the assumption that the classification only bases on one taxonomy, thus all users choose a genre from the same taxonomy. However, each user may build its own vocabulary, probably heterogenous from a standard taxonomy used in the system. In this case a large number of small semantic and syntactic heterogeneous taxonomies exist. Figure 4 shows the taxonomies T1 and T2 and two possible cases for an integration:

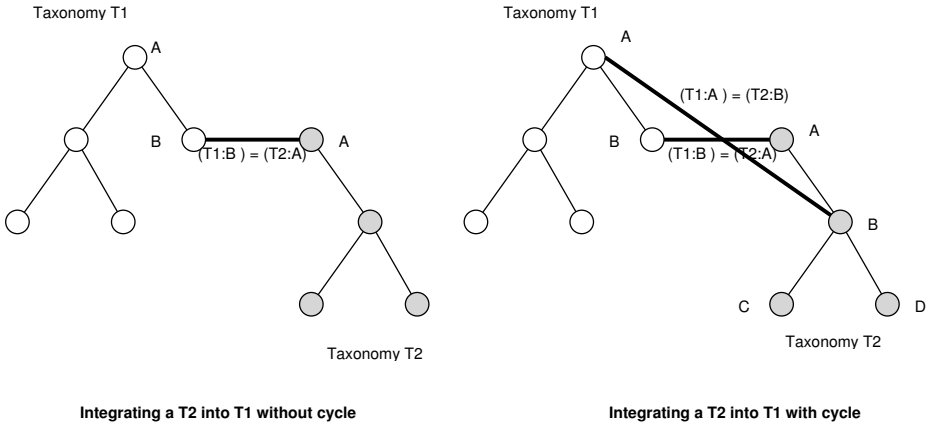

Fig. 4. Potential Taxonomy Integrations

Single Shared Node Both taxonomies share only one node: T1:B is an end node of a leaf in T1 and T2:A is a root node of T2. There occurs no conflict while an integration, on node T1:B taxonomy T2 is integrated through the relationship T1:B OWL:SAMEAS T2:A, thus T2:B, T2:C and T2: are more narrowed concepts for node T1:B. In the DHT this is expressed by the SUCC symbol, e.g.: *$EA66 SUCC http:// www.bossanovaguitar.com/ Slow Bossa*

Multiple shared Nodes Again T1:B is a the end of a leaf of T1 and T2:A is the root node of T2. T2:A has a more specialized node T2:B. However, this node is also a more generalized node of T2:A, since it is equal to T1:A which is more generalized than T1:B. Thus a cycle between T1:A and T2:B exists. There exists some strategies to resolve this cycle: We accept both correspondences, we reject both, we accept one of them.

Accepting correspondences may be a question of trust of the correspondence and not yet solved [2]. As long as this problem is not solved, we do not allow this form of integration.

5 Lookup Peer Node Profiles

In this section we propose three basic lookup operations and give ideas for complex matching operations using and propose first ideas towards keyword to taxonomy matchings.

The basic lookup operations presented in this section are helpful when querying for genres already known to the user, e.g. from his own mp3 files and their genre classification. Thus our approach allows users to browse the available genres and **discover new music** as they find genres that describe their own musical tastes. Examples of such rich semantic queries are:

1. Select users sharing Bossa Nova music
   Example $Q1 = /Style/Latin/Brazil/BossaNova$
2. Select users sharing classic music
   Example $Q2 = /Node/Style/Classic//$
3. Select users sharing the same music (Jazz and Bossa Nova) as me
   Example $Q3 = /Node[[Decade/1990][Style/Latin/Brazil/BossaNova]]$
4. Select Bossa Nova Music from the 90ies
   Example $Q4 = /Node[[Decade/1990][Style/Latin/Brazil/BossaNova]]$

These queries are based on the following three basic lookup operations supported by the data structure stored in the distributed hash table:

Exact Lookup This is the simplest form of a lookup. Consider the query *Q1 = /Style/ Latin/ Brazil/ Bossa Nova*. The value of the query is hashed to $EA66. This hash key is looked up in the DHT. Using the example from Figure 3 peer nodes with the ID A,B,C,D match the query.

This query is executed using one lookup, it's lookup costs are O(log N).

Tree Based Lookup Although DHT's only allow exact lookups, we can broaden a lookup query using the hierarchical structure of the classification path in the query and or narrow the query using the *SUCC* relationships stored in the object values of the hash keys. As an example we used Breath First Search to receive a complete answer.

Consider the example query *Q2=/Node/Style/Classic//* searching for peer nodes storing classical music. This path in query is hashed to $BA7F. Looking up this key in the DHT returns the object value of *SUCC /Style/Classical/Symphony*. Thus there

are no direct peer nodes classified at this path but peer nodes with more specialized, or narrowed, music are available. Using a BFS, the algorithm follows this relation and hashes the new path value. Thus $CCEE is looked up in the DHT, here the peer node E is registered, no further SUCC relationships exist.
BFS is an uninformed search strategy. It has an exponential complexity. Costs for time and memory are distributed among the number of super-peer nodes sharing the catalog. The number of messages grows with the number of edges passed during the search. The message routing costs, measured in the number of messages, are $O(log\ N) * edges$. Since our data structure is based on a tree, other tree search algorithms and its optimizations could be applied as well. Depending on the search goal, e.g. find the top N peer nodes categorized "close" to my query, *iterative deepening depth-first search* could be used. The implementation of this search strategy is still ongoing work.

Intersection-based Lookup In most cases a query consists of two or more taxonomy paths, e.g. query Q3 and query Q4. Consider the query *q3=/Node [[Decade/1990] [Style/Latin/Brazil/Bossa Nova]]*, thus both taxonomy paths

```
$AB55 /Decade/1990
AND
$EA66 /Style/Latin/Brazil/Bossa Nova
```

must match a profile of a peer. To solve this problem we have to find the intersection I of the object values sets S_j of the corresponding hash keys, thus executing two lookups and computing the intersection. In the following example only peer nodes D, B matches both taxonomy paths:

$S_{\$EA66} = (A, B, C, D)$, $S_{\$AB55} = (B, D)$;
$I_{S_{\$AB55} \cap S_{\$EA66}} = (A, B, C, D) \cap (B, D) = (B, D)$,

The primary challenge in computing the intersection of two sets of peer IDs in a peer to peer environment is limiting the amount of bandwidth used. So far we haven't implemented a specific strategy for computing remote intersections. Although we studied existing research and will give a brief overview of possible strategies: [11] investigated combinations of **compression technologies** for intersecting large sets, such as Caching, Bloom Filter, Gap Compression and Adaptive Set Intersection. Their results show that Bloom Filter can reduce the amount of the transferred data by the factor of 50 and when using Adaptive Set and Gap Compression by a factor of 40 while **caching and pre-computation** only reduces by a factor of 1.5-2. Reynolds and Vahdat suggest streaming results to users using **incremental intersection** [15]. Incremental intersection results are more effective when the final result set is big relative to the intersecting posting sets. Assuming users are usually satisfied with only a partial set of matching results, this will allow savings in communication as users are likely to terminate their queries early. The likelihood that users will terminate their queries early will be increased if the incremental results are prioritized based on a good ranking function. As a ranking function in our approach we use the number of mp3 files related to a particular path. We believe, a higher number of files for a path indicates a strong interest of the

user in this genre/decade/composer of music. Finally Fagin algorithm [5] is used in conjunction with this ranking function to generate incrementaly ranked results.

The basic lookup operations presented are helpful when querying for genres already known to the user, e.g. from his own mp3 files and their genre classification. However, in some cases a user does not know the exact taxonomy path before issuing a query and enters one or more keywords. Matching of, maybe misspelled, keywords to exact vocabularies in a DHT is an ongoing research area. A first approach for full text matching using Latent Semantic Indices is presented in [18]. However, a simple work around is to replicate the music taxonomy and the keyword matching at each super-peer. Query processing is a two fold process: A keyword query is posed against the super-peer, generating the matching taxonomy using 'traditional' IR technology. In the second step in the DHT the matching peer nodes are looked up. However this approach comes at the cost replicating the entire taxonomy at each super-peer.

6 Storage Load Balancing Strategies

Using a DHT abstraction, distributes objects randomly among peer nodes in a way that results in some nodes having O log(n) times as many objects as the average node. Further imbalance may result due to nonuniform distribution of objects in the identifier space and a high degree of heterogeneity in object loads and node capacities. To reduce this imbalance and to avoid overloading a single super-peer we allow multiple super-peers to share the load of a popular taxonomy path. As a single key can be mapped by the DHT to only one peer we need to introduce a level of indirection. The super-peer that is responsible for a taxonomy path does not store the data under the taxonomy path itself. It only has pointers to other super-peers that are responsible for storing the data. When more than one peer is responsible for storing the data then each responsible peer only stores a part of the data in a partition. A super-peer periodically determines whether its current load is above its target load. In this case the peer requests a new partition for the overloaded key. A new partition is immediately requested when the load jumps above an emergency threshold. New nodes for partitions are acquired from directories that store load information about nodes that have not reached their target load. These directories are also stored in the DHT. Nodes with light load periodically advertise their current target load to one of the directories.

Inserting data in the DHT requires two steps (see Figure 5). First the insert location needs to be resolved. A request to get an insert location is sent to the peer that would normally store the data. This peer returns one of the peers that are responsible for storing data under the requested key. Every time an insert request is issued a different peer is returned. Thus the data is uniformly distributed among the partitions. The data is then sent to the received insert location. Queries also take two steps. The query locations are resolved in the same way but this time all partitions are returned. In order to query all data the query must be send to all returned partitions.

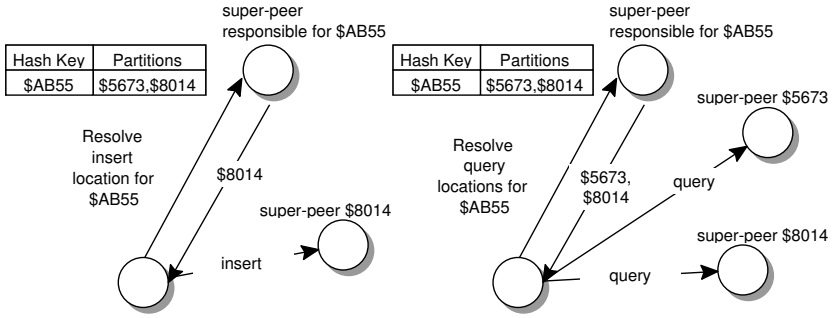

Fig. 5. Partition based Storage Load Balancing

7 Implementation and Evaluation

The main benefits of our load balancing approach over systems like [14] that move keys between virtual nodes is that we don't have the overheads caused by virtual nodes. Further, virtual nodes storing popular keys would receive a high number of object insertions, which would be too high for a single node to handle.

But this comes at an increased query cost when there is more than one partition. To prove this theory and compare the costs of our approach in terms of used bandwidth in contrast to virtual server we implemented both approaches using the *Narses* simulation framework [12]. We simulated a catalog based on 50 super-peers and 15000 peers. Each peer node joins and leaves the catalog within 3600 sec. The generated profile of each peer node includes on average ten taxonomy paths and has an average size of 800 bytes. The taxonomy paths in the peer profiles are Zipf distributed, regarding the fact that most of the peers share popular music but only a few share rare music. We simulated the required bandwidth of the peer-to-super network and the super-peer to super-peer network, using four different load balancing approaches: without load balancing (-LBM, -VS), virtual server(+VS), partition based load balancing (+LBM) and finally the combination of partition based load balancing and virtual server (+LBM,+VS). Figure 6 shows the results of our simulation. Our load balancing approach performs better than virtual server and the simulation without any load balancing. The combination of our approach with virtual server produces an extra overhead of 10% in contrast to a simulation using partition based load balancing only. This result are valid for a small super-peer network, such as simulated in our experiment. However we assume that for larger super-peer networks the combination of our approach and virtual server will work best. Please note, in our approach we are only able to reduce the number of taxonomy paths a super-peer is responsible for. Virtual server allow to distribute the load over several super-peers using a many to many scheme. To prove this thesis we will conduct further experiments and mathematical calculations.

To analyze the overall bandwidth of the approach including queries we conducted a number of experiments using the load balancing strategies from above. Each peer issues each 240 sec an exact query for a taxonomy path. Our simulation in Figure 7 has shown that the average required bandwidth for serving queries and joining and leaving peers of

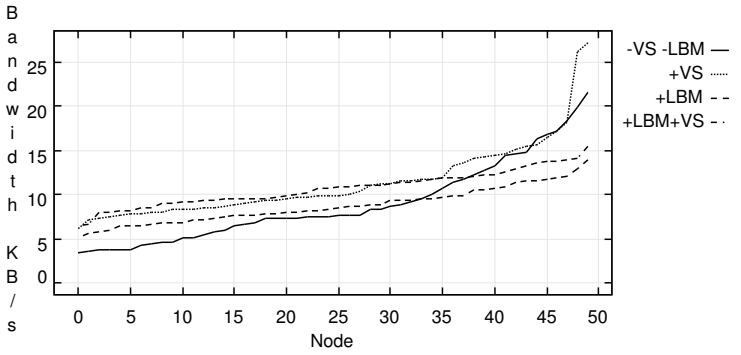

Fig. 6. Bandwidth usage for storing and removing peer profiles

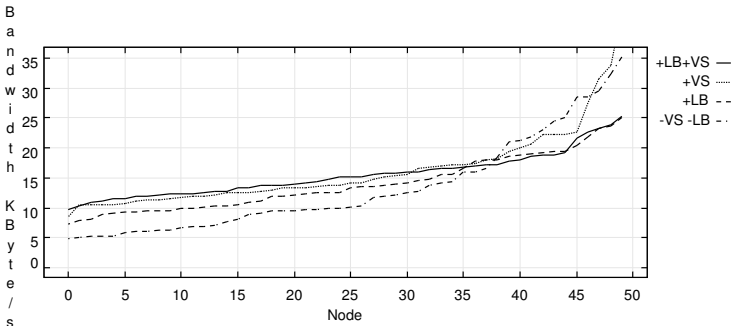

Fig. 7. Overall Bandwidth usage of the System

each super-peer is 25 KByte/sec. For running a small network with only 15.000 peers less than DSL bandwidth is required. However, our storage load balancing approach is not completely able to reduce the query load balancing costs. Figure 8 shows the costs using our storage load balancing approach only for joining leaving peer nodes (J/L) and for issuing queries and joining and leaving peer nodes (J/L +Query). Since queries are Zipf distributed as well, we suggest caching taxonomy paths at intermediate nodes that lie on the path taken by search query and refer to this as *Path Caching with Expiration (PCX)* because cached taxonomy path entries typically have expiration times after which they are considered stale and require a new search. [16] proposes *Controlled Update Propagation* a more sophisticated approach to maintain caches of meta data in a peer-to-peer network where each (super-peer) node maintains a directory of the cached copies of its meta data, thus no expiration time is needed. PCX and CUP are desirable because they distributes query load for popular taxonomies across multiple nodes, reduces latency, and alleviates hot spots.

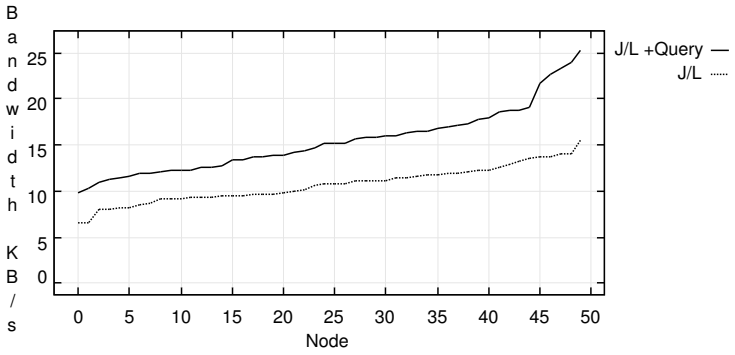

Fig. 8. Overall Bandwidth Usage vrs. Bandwidth Usage for Join/Leave Operations only

8 Summary and Further Work

We presented a completely new approach for enabling efficient semantic query routing in P2P file sharing systems. Queries are no longer flooded throughout the whole system but are routed only to peers providing the right content to answers a query. Our basic concept is a distributed catalog, storing semantic annotations about the available content of peers in the network. Our approach is capable of storing a large number of peer profiles in a space efficient way. Due to the use of key-to-query mappings, inverted indices and tree structured sub sequences we are able to store large taxonomies in a DHT. Our data structure allows to resolve exact, narrowed and broadened queries. Further we presented strategies for realizing conjunctive queries in our system. The catalog is built on top of a distributed hash table based infrastructure using a partition based storage load balancing approach. Our simulation has shown, that our approach scales to ten thousands of peers using a small set of super-peers connected to each other with less than DSL bandwidth.

However, this paper offered only a high-level description of our approach. Much work remains, for example dynamic storage load balancing strategies allowing super-peers to join and leave the catalog with a high frequency while the catalog remains robust. The use of existing query load balancing strategies, such as CUP or PCX should be evaluated, to allow the system to scale to a high number of queries. Further, using our existing literature survey we will evaluate conjunctive lookup queries based on incremental intersections of ranked sets. To reduce costs of a tree based search we will investigate the use of an iterative deepening depth first search strategy, limiting results only to top N peer nodes. Last, we allow only hierarchical relations to be stored in the DHT. Although not needed in our scenario our data structure allows to store much more complex constructs than rather simple hierarchies. Therefore our further work will include the formulation of a road map towards storing and querying such structures.

References

1. M. Balazinska, H. Balakrishnan, and D. Karger. Ins/twine: A scalable peer-to-peer architecture for intentional resource discovery. In *M. Balazinska, H. Balakrishnan, and D. Karger. In Proceedings of Pervasive 2002.*, 2002.
2. J. Caroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs provenance and trust. Technical Report HPL-2004-57, Hewlett Packard Ressearch Labs, 2004.
3. A. Crespo and H. G. Molina. Semantic overlay networks, November 2002. Stanford University, Technical Report.
4. R. Dolin, D. Agrawal, L. Dillon, and A. E. Abbadi. Pharos: A scalable distributed architecture for locating heterogeneous information sources. Technical Report TRCS96-05, 16, 1996.
5. R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Symposium on Principles of Database Systems*, 2001.
6. J. C. French, A. L. Powell, C. L. Viles, T. Emmitt, and K. J. Prey. Evaluating database selection techniques: A testbed and experiment. In *Research and Development in Information Retrieval*, pages 121–129, 1998.
7. L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating data sources in large distributed systems. In *Proceedings of the 29 VLDB*, Berlin, Germany, 2003.
8. H. Garcia-Molina and B. Yang. Efficient search in peer-to-peer networks. In *Proceedings of ICDCS*, 2002.
9. L. Gravano and H. García-Molina. Generalizing GlOSS to vector-space databases and broker hierarchies. In *International Conference on Very Large Databases, VLDB*, pages 78–89, 1995.
10. P. Keegan. Is the music store over? *Business 2.0*, March 2004.
11. J. Li, B. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *2nd International Workshop on Peer-to-Peer Systems. ITPTS*, 2003.
12. M.Baker and T. Giuli. Narses: A scalable flow-based network simulator. Technical report, Stanford University, 2002.
13. M. Oezsu and P. Valduriez. *Principles of distributed database systems*. Prentice Hall, 2nd edition edition, 1999.
14. A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *Proc. IPTPS*, 2003.
15. P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *ACM MIDDLEWARE Conference*, 2003.
16. M. Roussopoulos and M. Baker. Cup: Controlled update propagation in peer to peer networks. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.
17. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. pages 149–160.
18. C. Tang, Z. Xu, and M. Mahalingam. pSearch: Information retrieval in structured overlays. In *ACM HotNets-I*, October 2002.
19. P. Triantafillou and T. Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks. In *VLDB '03 Workshop on Databases, Information Systems, and Peer-to-Peer Computing*, 2003.