Complex Queries in DHT-based Peer-to-Peer Networks

Matthew Harren
UC Berkeley
matth@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
jmh@cs.berkeley.edu

Ryan Huebsch
UC Berkeley
huebsch@cs.berkeley.edu

Boon Thau Loo
UC Berkeley
boonloo@cs.berkeley.edu

Scott Shenker
International Computer Science Institute
shenker@icsi.berkeley.edu

Ion Stoica
UC Berkeley
istoica@cs.berkeley.edu

Abstract

Recently a new generation of P2P systems, offering distributed hash table *(DHT) functionality, have been proposed. These systems greatly improve the scalability and exact-match accuracy of P2P systems, but offer only the exact-match query facility. This paper outlines a research agenda for building complex query facilities on top of these DHT-based P2P systems. We describe the issues involved and outline our research plan and current status.*

1 Introduction

Peer-to-peer (P2P) networks are among the most quickly-growing technologies in computing. However, the current technologies and applications of today's P2P networks have (at least) two serious limitations.

Poor Scaling: From the centralized design of Napster, to the notoriously inefficient search process of Gnutella, to the hierarchical designs of FastTrack [4], the scalability of P2P designs has always been problematic. While there has been significant progress in this regard, scaling is still an issue in the currently deployed P2P systems.

Impoverished query languages: P2P networks are largely used for filesharing, and hence support the kind of simplistic query facility often used in filesystem "search" tools: *Find all files whose names contain a given string.* Note that "search" is a limited form of querying, intended for identifying ("finding") individual items. Rich query languages should do more than "find" things: they should also allow for combinations and correlations among the things found. As an example, it is possible to search in Gnutella for music by J. S. Bach, but it is not possible to ask specifically for all of Bach's chorales, since they do not typically contain the word "chorale" in their name.

The first of these problems has been the subject of intense research in the last few years. To overcome the scaling problems with unstructured P2P systems such as Gnutella where data-placement and overlay network construction are essentially random, a number of groups have proposed *structured* P2P designs. These proposals support a *Distributed Hash Table* (DHT) functionality [11, 14, 12, 3, 18]. While there are significant implementation differences between these DHT systems (as we will call them), these systems all support (either directly or indirectly) a hash-table interface of `put(key,value)` and `get(key)`. Moreover, these systems are extremely scalable; lookups can be resolved in $\log n$ (or n^α for small α) overlay routing hops for an overlay network of size n hosts. Thus, DHTs largely solve the first problem. However, DHTs support only "exact match" lookups, since they are hash tables. This is fine for fetching files or resolving domain names, but presents an even more impoverished query language than the original, unscalable P2P systems. Hence in solving the first problem above, DHTs have aggravated the second problem.

We are engaged in a research project to address the second problem above by studying the design and implementation of complex query facilities over DHTs; see [7] for a description of a related effort. Our goal is not only bring the traditional functionality of P2P systems – filesharing – to a DHT implementation but also to push DHT query functionality well beyond current filesharing search, while still maintaining the scalability of the DHT infrastruc-

tures. This note offers a description of our approach and a brief discussion of our current status.

2 Background

Before describing our approach, we first discuss some general issues in text retrieval and hash indexes and then explain why we are not proposing a P2P database.

2.1 Text Retrieval and Hash Indexes

As noted above, DHTs only support exact-match lookups. Somewhat surprisingly, it has been shown that one can use the exact-match facility of hash indexes as a substrate for textual similarity searches, including both strict substring searches and more fuzzy matches as well [17]. The basic indexing scheme is to split each string to be indexed into "n-grams": distinct n-length substrings. For example, a file with ID I and filename "Beethovens 9th" could be split into twelve trigrams: Bee, eet, eth, tho, hov, ove, ven, ens, ns%, s%9, %9t, 9th (where '%' represents the space character). For each such n-gram g_i, the pair (g_i, I) is inserted into the hash index, keyed by g_i. One can build an index over n-grams for various values of n; it is typical to use a mixture of bigrams and trigrams.

Given such an index, a substring lookup like "thoven" is also split into n-grams (*e.g.*, tho, hov, ove, ven), and a lookup is done in the index for each n-gram from the query. The resulting lists of matches are concatenated and grouped by file ID; the count of copies of each file ID in the concatenated list is computed as well. For strict substring search, the only files returned are those for which the count of copies is as much as the number of n-grams in the query (four, in our example). This still represents a small superset of the correct answer, since the n-grams may not occur consecutively and in the correct order in the results. To account for this, the resulting smallish list can be postprocessed directly to test for substrings.

While the text-search literature tends not to think in relational terms, note that the query above can be represented nearly directly in SQL:

```
  SELECT  H.fileID, H.fileName
    FROM  hashtable H
   WHERE  H.text IN (<list-of-n-grams-in-search>)
GROUP BY  H.fileID
  HAVING  COUNT(*) >= <#-of-n-grams-in-search>
     AND  H.fileName LIKE <substring expression>
```

In relational algebra implementation terms, this requires an index access operator, a grouping operator, and selection operators[1].

The point of our discussion is not to dwell on the details of this query. The use of SQL as a query language is not important, it merely highlights the universality of these operators: they apply not only to database queries, but also to text search, and work naturally over hash indexes. If we can process relational algebra operators in a P2P network over DHTs, we can certainly execute traditional substring searches as a special case.

2.2 Why Not Peer-to-Peer Databases?

We have noted that relational query processing is more powerful than the search lookups provided by P2P filesharing tools. Of course, traditional database systems provide a great deal of additional functionality that is missing in P2P filesharing – the most notable features being reliable, transactional storage, and the strict relational data model. This combined functionality has been the cornerstone of traditional database systems, but it has arguably cornered database systems and database research into traditional, high-end, transactional applications. These environments are quite different from the P2P world we wish to study. Like the users of P2P systems, we are not focused on perfect storage semantics and carefully administered data. Instead, we are interested in ease of use, massive scalability, robustness to volatility and failures, and best-effort functionality.

The explosive growth of the P2P networks show that there is a viable niche for such systems. Transactional storage semantics are important for many

[1] We have kept the substring match example strict, for clarity of exposition. In many cases, n-gram search can also support a useful fuzzy semantics, in which files are simply ranked by descending count of n-gram matches (or some more subtle ranking metric), down to some cutoff. This allows the searching to be more robust to misspellings, acronyms, and so on, at the expense of false positives in the answer set. Such a ranking scheme can also be represented in SQL via an ORDER BY clause containing an externally-defined ranking function.

Also note that our index can be augmented so that each entry holds the offset of the n-gram in the string – this allows tests for ordering and consecutiveness to be done without observing the actual string. This optimization usually only helps when the strings being indexed are long – *e.g.* for full-text documents rather than file names. It is also clumsy to express this in SQL, though the relational algebra implementation is relatively straightforward.

applications, but are not familiar to most end-users today, who typically use file systems for storage. Most users do not want to deploy or manage a "database" of any kind. We believe this is a "street reality" worth facing, in order to maintain the grassroots approach natural in P2P. As part of that, we do not see transactions as an integral part of the research thrust we describe here.

On the other hand, relational data modeling is in some sense universal, and its details can be abstracted away from the user. All data is relational at some level, inasmuch as one can think of storing it in a table of one column labeled "bits". In many cases, there is somewhat more natural structure to exploit: as noted above, sets of P2P files can be thought of as relations, with "attributes" name, ID, host, etc. The user need not go through a complex data modeling exercise to enable a system to index and query these attributes in sophisticated ways.

Hence we do not see a pressing need for users of P2P system to load their data into a database; we prefer to build a query engine that can use the natural attributes exposed in users' existing data, querying those attributes intelligently while providing the storage semantics that users have learned to live with. In fact, we wish to stress the point that it may be strategically unwise to discuss peer-to-peer *databases* at all, with their attendant complexities in software and administration associated with database storage. Instead, we focus on peer-to-peer *query processing*, and separate it from the problem of storage semantics and administration. Of course we leave the option open: our ideas could be combined with P2P transactional mechanisms, *e.g.* as suggested in [10]. However, we do not wed ourselves to the success (both technical and propagandistic) of such efforts. It is worth noting that despite current commercial packaging, relational database research from the early prototypes onward [1, 15] has separated the storage layer from the query processing layer. We are simply following in those footsteps in a new storage regime.

3 P2P Query Processing

Our design is constrained by the following goals:

Broad Applicability: A main goal for our work is that it be broadly and practically usable. In the short term, this means that it should be able to interact with user's filesystems in the same way as existing P2P filesharing systems.

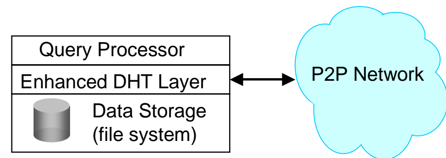

Figure 1: The software architecture of a node implementing query processing.

Minimal Extension to DHT APIs: DHTs are being proposed for use for a number of purposes in P2P networks, and we do not want to complicate the design of a DHT with excessive hooks for the specifics of query processing. From the other direction, we need our query processing technology to be portable across DHT implementations, since a clear winner has not emerged in the DHT design space. For both these reasons, we wish to keep the DHT APIs as thin and general-purpose as possible. The relational operators we seek to implement can present challenging workloads to DHT designers. We believe this encourages synergistic research with both query processing and DHT design in mind.

3.1 Architecture

Based on these design decisions, we propose a three-tier architecture as diagrammed in Figure 1. Note that all networking is handled by the DHT layer: we will use DHTs not only as an indexing mechanism, but also as a network routing mechanism. We proceed with a basic overview of the architecture and its components.

The bottom layer is a local data store, which must support the following API:

(1) An Iterator (as in Java, or STL) supporting an interface to scan through the set of objects.

(2) For each object, accessors to the attributes localID and contents. The former must be a store-wide unique identifier for the object, and the latter should be "the content" of the object, which can be a byte-array.

(3) A metadata interface to find out about additional attributes of the objects in this store.

(4) Accessors to the additional attributes.

Note that we do not specify many details of this data store, and our interface is read-only. Based on our first design goal, we expect the store to often be a filesystem, but it could easily be a wrapper over a database table or view.

The next layer is the DHT layer, which supports

the put/get interface, enhanced with the following:
(1) An Iterator called ℓscan, which can be allocated by code that links to the DHT library on this machine (typically the Query Processor of Figure 1). ℓscan allows the local code to iterate through all DHT entries stored on this machine.
(2) A callback newData that notifies higher layers of the identifier for new insertions into the local portion of the DHT.

ℓscan is important because various query processing operators need to scan through all the data, as we shall see below. The addition of scanning is not unusual: other popular hashing packages support scanning through all items as well [13]. Note that we break the *location transparency* of the DHT abstraction in ℓscan, in order to allow scans to be parallelized across machines – a distributed scan interface would only have a a single caller per Iterator. The ℓscan interface allows code to run at each machine, scanning the local data in parallel with other machines. newData is desirable because we will use DHTs for temporary state during query processing, and we will want insertions into that state to be dealt with in a timely fashion.

The top layer is the query processing (QP) layer, which includes support for the parallel implementations of query operators described below, as well as support for specifying queries and iterating through query results. Our query executor will be implemented in the traditional "pull-based" *iterator* style surveyed by Graefe [6], with parallel "push-based" communication encapsulated in exchange operators [5]. We plan to support two query APIs: a graph-scripting interface for specifying explicit query plans, and a simplified SQL interface to support declarative querying. Common query types (such as keyword search) can be supported with syntatic sugar for SQL to make application programming easier.

3.2 Namespaces and Multicast

DHT systems assume a flat identifier space which is not appropriate to manage multiple data structures, as will be required for query processing. In particular, we need to be able to name tables and temporary tables, tuples within a table, and fields within a tuple. One approach is to implement an hierarchical name space on top of the flat identifier space provided by DHTs, by partitioning the identifiers in multiple fields and then have each field identify objects of the same granularity.

A hierarchical name space also requires more complex routing primitives such as multicast. Suppose we wish to store a small temporary table on a subset of nodes in the network. Then we will need to route queries to just that subset of nodes. One possibility would be to modify the routing protocol such that a node forwards a query to all neighbors that make progress in the identifier space towards *any* of the identifiers covered by the query.

3.3 Query Processing Operators

We will focus on the traditional relational database operators: selection, projection, join, grouping and aggregation, and sorting. A number of themes arise in our designs. First, we expect communication to be a key bottleneck in P2P query processing, so we will try to avoid excessive communication. Second, we wish to harness the parallelism inherent in P2P, and we will leverage traditional ideas both in intra-operator parallelism and in pipelined parallelism to achieve these goals. Third, we want answers to stream back in the style of *online query processing* [9, 8]: P2P users are impatient, they do not expect perfect answers, and they often ask broad queries even when they are only interested in a few results. Next, we we focus on the example of joins; grouping and other unary hashing operators are quite analogous to joins, with only some subtle differences [2].

Our basic join algorithm on two relations R and S is based on the pipelined or "symmetric" hash join [16], using the DHT infrastructure to route and store tuples. The algorithm begins with the query node initializing a unique temporary DHT namespace, T_{joinID}. We assume that data is iterating in from relations R and S, which each may be generated either by an ℓscan or by some more complex query subplan. The join algorithm is fully symmetric with respect to R and S, so we describe it without loss of generality from one perspective only. The join operator repeatedly gets a datum from relation R, extracts the join attribute from the datum, and uses that attribute as the insertion key for DHT T_{joinID}. When new data is inserted into T_{joinID} on some node, the newData call notifies the QP layer, which takes that datum and uses its join key to probe local data in T_{joinID} for matches. Matches are pipelined to the next iterator in the query plan (or to the client machine in the case of

final results). In the case where one table (say S) is already stored hashed by the join attribute, there is no need to rehash it – the R tuples can be scanned in parallel via ℓscans, and probe the S DHT.[2]

Selection is another important operator. Relational selection can either be achieved by a table-scan followed by an explicit test of the selection predicate, or by an index lookup (which can optionally also be followed by a predicate test). Clearly explicit tests can be pushed into the network to limit the flow of data back. Index-supported selections further limit network utilization by sending requests only to those nodes that will have data. DHT indexes currently support only equality predicates. An interesting question will be to try and develop range-predicate support in a manner as efficient as current DHTs.

4 Status

We have implemented the join operation by modifying the existing CAN simulator [12] and performed exhaustive simulations. In addition to the solution presented in the previous section, we have implemented several other join variants. For example, in one of the variants, we rehash only one of the tables (say S) by the join attribute. Then each node scans locally the other table, R, and for each tuple it queries the tuples of S_t with the same join attribute value and performs local joins. The main metric we consider in our simulations is the join latency function $f(x)$, which is defined as the fraction of the total result tuples that the join initiator receives by time x. One interesting result is that this function is significantly smoother when we use a Fair Queueing like algorithm to allocate the communication and process resources. Other metrics we consider in our simulations are data placement and query processing hotspots, as well as routing hotspots. Preliminary results show that for realistic distributions of the join attribute values, there are significant hotspots in all dimensions: storage, processing, and routing.

[2]This degenerate case of hash join is simply a parallel index-nested-loops join over a DHT. It also suggests an index-on-the-fly scheme, in which only one table (S) is rehashed – after S is rehashed, R probes it. Index-on-the-fly blocks the query pipeline for the duration of the rehashing, however, and is unlikely to dominate our pipelining scheme.

5 Acknowledgments

The authors have benefited from discussions with Michael J. Franklin, Petros Maniatis, Sylvia Ratnasamy, and Shelley Zhuang. We thank them for their insights and suggestions.

References

[1] Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J., Griffiths, P. P., III, W. F. K., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W., and Watson, V. System r: Relational approach to database management. *ACM Transactions on Database Systems (TODS) 1*, 2 (1976), 97–137.

[2] Bratbergsengen, K. Hashing Methods and Relational Algebra Operations. In *Proc. of the International Conferrence on Very Large Data Bases (VLDB)* (1984), pp. 323–333.

[3] Druschel, P., and Rowstron, A. Past: Persistent and anonymous storage in a peer-to-peer networking environment. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS 2001)* (Elmau/Oberbayern, Germany, May 2001), pp. 65–70.

[4] Fsttrack. `http://www.fasttrack.nu/`.

[5] Graefe, G. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. ACM-SIGMOD International Conference on Management of Data* (Atlantic City, May 1990), pp. 102–111.

[6] Graefe, G. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv. 25*, 2 (June 1993), 73–170.

[7] Gribble, S., Halevy, A., Ives, Z., Rodrig, M., and Suciu, D. What can p2p do for database, and vice versa? In *Proc. of WebDB Workshop* (2001).

[8] Haas, P. J., and Hellerstein, J. M. Online Query Processing: A Tutorial. In *Proc. ACM-SIGMOD International Conference on Management of Data* (Santa Barbara, May 2001). Notes posted online at http://control.cs.berkeley.edu.

[9] Hellerstein, J. M., Haas, P. J., and Wang, H. J. Online Aggregation. In *Proc. ACM SIGMOD International Conference on Management of Data* (1997).

[10] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. OceanStore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)* (Boston, MA, November 2000), pp. 190–201.

[11] Plaxton, C., Rajaraman, R., and Richa, A. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA* (Newport, Rhode Island, June 1997), pp. 311–320.

[12] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A scalable content-addressable network. In *Proc. ACM SIGCOMM* (San Diego, CA, August 2001), pp. 161–172.

[13] Seltzer, M. I., and Yigit, O. A new hashing package for unix. In *Proc. Usenix Winter 1991 Conference* (Dallas, Jan. 1991), pp. 173–184.

[14] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference* (San Diego, California, August 2001).

[15] Stonebraker, M., Wong, E., Kreps, P., and Held, G. The design and implementation of ingres. *ACM Transactions on Database Systems (TODS) 1*, 3 (1976), 189–222.

[16] Wilschut, A. N., and Apers, P. M. G. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. First International Conference on Parallel and Distributed Info. Sys. (PDIS)* (1991), pp. 68–77.

[17] Witten, I. H., Moffat, A., and Bell, T. C. *Managing Gigabytes: Compressing and Indexing Documents and Images*, second ed. Morgan Kaufmann, 1999.

[18] Zhao, B. Y., Kubiatowicz, J., and Joseph, A. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, University of California at Berkeley, Computer Science Department, 2001.