Distributed Query Processing and Catalogs for Peer-to-Peer Systems

Vassilis Papadimos David Maier Kristin Tufte

OGI School of Science & Engineering
Oregon Health & Science University
{vpapad,maier,tufte}@cse.ogi.edu

Abstract

Peer-to-peer (P2P) architectures are commonly used for file-sharing applications. The reasons for P2P's popularity in file sharing – fault tolerance, scalability, and ease of deployment – also make it a good model for distributed data management. In this paper, we introduce a scalable P2P framework for distributed data management applications using *mutant query plans:* XML serializations of algebraic query plan graphs that can include verbatim XML data, references to resource locations (URLs), and abstract resource names (URNs). We show how we can build distributed catalogs based on multi-hierarchic namespaces that can efficiently handle content indexing and query routing. We also discuss how peers can convey the currency and coverage of their data, and how queries can use this information to manage the inherent tradeoffs between answer completeness, timeliness, and latency.

1. Introduction

Many file-sharing systems today use peer-to-peer (P2P) architectures, where participants simultaneously serve and receive files. Most P2P systems handle file *sharing* in a decentralized P2P fashion. Some systems however fall back to a client-server architecture for *indexing* and *searching*. There are thus two main approaches, which we will name after the first popular systems that implemented them:

- The "Napster" (also called *hybrid* in [YG01]) approach: A centralized group of servers indexes filenames, and all queries must go through them.
- The "Gnutella" (or *pure*) approach: No central indices are maintained; queries are broadcast to a node's "neighbors" (which then broadcast them to all of *their* neighbors, and so on, up to a fixed number of steps, called the *horizon*).

P2P systems are successful for several reasons, including:

- Ease of deployment: Each user installs a single package that encompasses both client and server code; its initial configuration depends only on knowing a fixed index server or a single other installation; servers need not be continuously active.
- Ease of use: The server code is bundled with a user interface application to publish, search and retrieve content.
- Fault tolerance: Failure or unavailability of a single server (other than a central index) does not disable the system. It might render some content unavailable, but much of the content ends up being heavily replicated.
- Scalability: As the number of users and amount of content increase, so does the number of servers; protocols do not require "all-to-all" communication or coordination.

However, there are limitations that come with these advantages. The schema and queries for searching for content are typically hardwired into the application; there can be bottlenecks at the centralized index; there are no mechanisms for combining or otherwise manipulating the content itself. Recently there is interest in adapting the P2P model to distributed data management scenarios. We see two major issues for current P2P approaches here: weak query capabilities, and limitations in index scalability and result quality.

Current P2P systems offer very limited querying functionality: simple selection on a predefined set of index attributes, IR-style string matching or containment, no manipulation of content. These limitations are acceptable for file-sharing applications, since people find ways to encode metadata about a file in the filename, but more general P2P applications will require a richer query model. We want to enable content publishers to export

Proceedings of the 2003 CIDR Conference

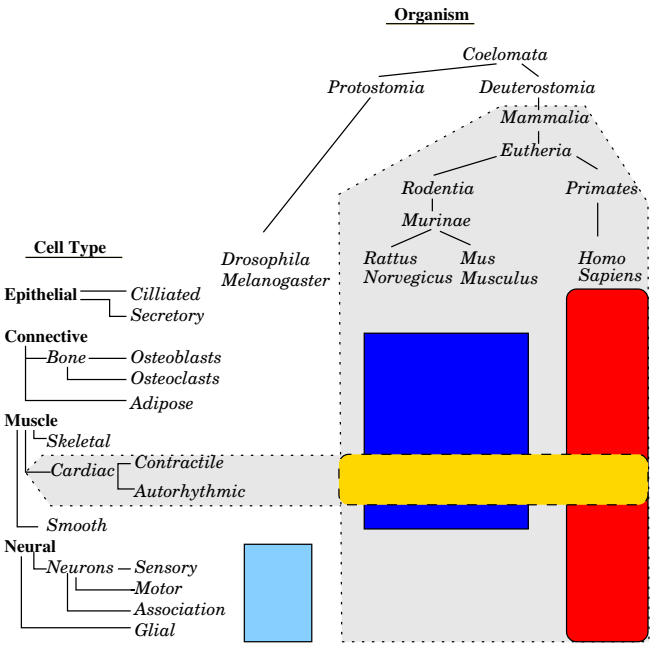

Figure 1: *Of Mice and Men*. Using hierarchies to describe and query repositories of gene expression data. A query about mammalian heart cells partially covers a database on connective and muscle rodent cells, and a database on human cells.

structured or semi-structured views of their data (for example using XML), and allow users to query them using a full-featured query language.

In terms of index scalability and result quality both the Napster and Gnutella approaches have serious limitations. Centralized index servers don't scale with the number of clients. Query broadcasting wastes network bandwidth and hurts result quality by limiting the availability of rare content. Again, file-sharing networks thrive despite these limitations: Finding 10 out of the 100 available copies of the same file is usually good enough, but for general-purpose P2P query systems, we will have to do better.

The assumption usually made by file-sharing implementations is that any file can potentially be replicated at any node in the system. This is a reasonable assumption for file-sharing systems, but not necessarily true for P2P applications in general, and database-style applications in particular. A content provider might not want its content replicated in bulk; the natural unit of retrieval (e.g. a record) might be at too small a level of granularity for a file-based approach; effective evaluation of query conditions may require having certain content aggregated (lowest price, closest location).

In this paper, we describe a peer-to-peer architecture for distributed querying that works well for application domains where content providers have specific affinities for storing, replicating, or indexing different subsets of a global data namespace. Peers express their preferences for the data they are serving or looking for using a name space of multiple hierarchical categories. Queries are routed efficiently, without depending on centralized index servers or query broadcasting, and peers can make intelligent choices about query latency, data completeness and currency tradeoffs.

For example, consider different biomedical research groups hosting on-line repositories of gene-expression data (such as those obtained from microarray experiments). Emerging data interchange standards such as MIAME [BHQ+01] allow groups to exchange and replicate expression data. Groups choose what data to host, generally based on their own research interests. In our approach, groups can indicate their interest areas relative to organism and cell-type hierarchies. In Figure 1 we see interest areas of three groups: one for neural cells in fruit flies, a second for connective and muscle cell in rodents, and a third with all cell types for humans. Given this coverage information, a site processing a query related to cardiac muscle cells in mammals can route the query to the second or third site (where it *might* find relevant data), but can ignore the first site (where it surely will not). More generally, interest areas can describe indexing coverage of other groups' data, or even "meta-coverage" of other groups data and index interest areas. (Note that MIAME defines many more metadata attributes for expression data, such as anatomical location and developmental stage. We used just two categories here because it is easy to depict graphically.)

We will often use the terms *client* and *server* for participants in our system. There are some activities where participants act as peers, and others, most notably query submission and processing where there is clearly a client and a server. The important distinction between the P2P model and the client-server model is not that such roles do not exist, but that they are not fixed or pre-assigned; this query's client may well become the next query's server.

Here is an overview of the rest of this paper. In Section 2 we introduce our running example, a P2P "garage-sale" application, and present *mutant query plans*, our coordinator-less distributed query execution framework. In Section 3 we present multi-hierarchic namespaces and explain their use in P2P indexing and querying. Section 4 discusses how peers can reason about answer completeness, redundancy, and currency of answers, and tradeoffs versus query latency. Section 5 covers issues and extensions to our framework. Section 6 presents related work, and Section 7 concludes.

2. Mutant queries and the P2P garage sale

We will use a distributed garage sale as our running example. A garage sale is the real world situation that most closely resembles a P2P network. People sell and buy things without middlemen, or predetermined seller/buyer (server/client) roles.

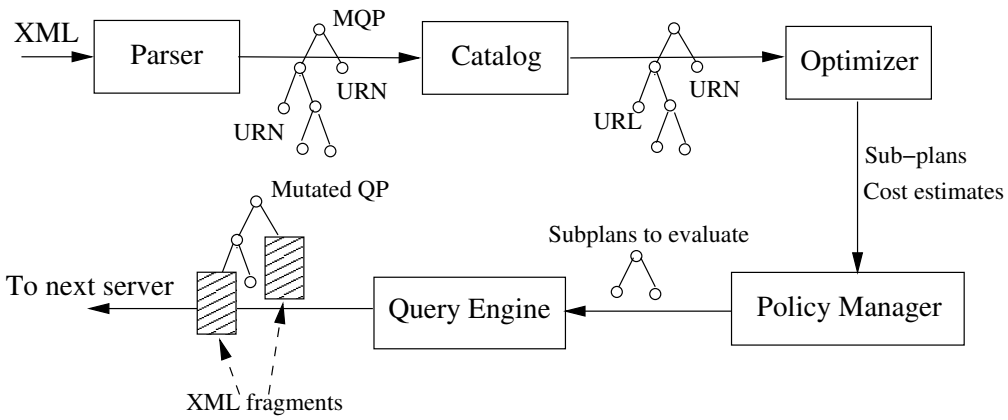

Figure 2: Mutant Query Processing.

In the P2P garage sale, data about items in garage sales, second hand stores, and auctions come online. The system simply brings together people who want to sell or buy used items; the actual transactions happen outside the system. We posit a collection of local consignment shops that handle the actual storage, sales and delivery of goods for a commission, and which can co-operate with each other to transfer items closer to a potential buyer for inspection and purchase. Most participants who post information to our system will have registered to sell through a particular shop, but nothing prevents someone from selling directly, say, out of his or her garage.

Each for-sale item has an associated data bundle with various information in it: item name, seller location, description, condition, images, quantity, price, etc. We will assume that sellers export these data bundles in XML. Notice that our data are more structured and varied than the typical file description, and support much more meaningful queries; our query language therefore should be more powerful than the typical IR-based string matching interfaces found in most P2P systems. A seller can run his or her own server to publish items for sale, or can post them to a server run by a consignment shop.

Many queries will combine data residing in multiple peers. Transferring all relevant data to a central location wastes time and bandwidth. For-sale data is likely to have locality in terms of geographic location or category of merchandise (e.g., a consignment shop/server that specializes in used clothing). We need a distributed query execution mechanism, so that we can run our queries "closer" to the relevant data.

In the remainder of this section we will briefly describe Mutant Query Plans (MQPs), a framework for coordination-free distributed query execution. (You can find additional information about MQPs in [PM02a, PM02b]). We have implemented an MQP prototype with the basic features described in this section using the NIAGARA system [NDM+01] as our XML query engine.

A mutant query plan is an algebraic query plan graph, encoded in XML, that may also include verbatim XML-encoded data, references to resource locations (URLs), and references to abstract resource names (URNs). Each MQP is tagged with a *target:* a network address to send the result to, once the MQP is fully evaluated.

The ability of mutant query plans to package together query operators and data means that we can use them to represent all the stages in the evaluation of a distributed query. An MQP starts out as a regular query operator tree at the client, and is then passed around from server to server, accumulating partial results, until it is fully evaluated into a constant piece of XML data and returned to the client.

A server can choose to *mutate* an incoming MQP in two ways. It can *resolve* a URN to one or more URLs, or a URL to its corresponding data. The server can also *reduce* the MQP by evaluating a sub-graph of the plan that contains only data at the leaves, and substituting the results in place of the sub-plan. If the plan is now completely evaluated (i.e., it has been reduced to a constant piece of XML-encoded data), the server sends it to the target, otherwise it routes the plan to another server that can continue processing.

Figure 2 shows this process in more detail. An MQP arrives at a server encoded in XML. The server parses the plan into an in-memory graph, and determines the URNs that it can resolve. The optimizer finds the locally evaluable sub-plans (a sub-plan is locally evaluable if all its leaves are verbatim XML data, URLs, or resolvable URNs), optimizes them and estimates their costs. A policy manager component decides which of those sub-plans to evaluate, and forwards them for execution to the query engine. The server then substitutes the resulting XML fragments as verbatim XML data in the place of the evaluated sub-plans, serializes the mutated plan in XML and sends it to some other server that can continue the plan's evaluation.

As an example, suppose we are looking for CDs for $10 or less in the Portland area. Sellers publish lists that include CD titles. Our P2P client has a list of our favorite songs, and we can use an online track-listing service, such as CDDB [CDB] or FreeDB [FDB], to connect these two resources.

Figure 3 shows a mutant query plan for this request. The plan includes regular query operators such as select and join, a *display* pseudo-operator that specifies the query plan's target, a constant piece of XML with the songs we are looking for, and two URNs: `urn:ForSale:Portland-CDs`, and `urn:CD:TrackListings`.

There are no reduction steps we can perform on the MQP of Figure 3, unless we resolve one of the two abstract resources (URNs) to specific URLs or to raw data. Figure 4 shows two steps in this process. In Figure 4(a), a server resolves the `ForSale` URN to a union of two seller URLs, pushes the select operator through the union, and forwards the plan to one of the seller servers. In Figure 4(b) the server substitutes its CD data for its URL, evaluates the select and reduces its part of the plan to a constant piece of XML data. This series of URN-to-URL-to-data resolutions and sub-plan reductions will continue until the whole plan is evaluated, and forwarded to its target.

Resolution drives the query evaluation process of MQPs. Resolving URLs is straightforward: we can either connect to the specified server to fetch the data, or forward the whole MQP to it. In our current implementation, we resolve URNs by consulting a *catalog*, which we maintain locally at each peer. A catalog contains mappings from URNs to (sets of) URLs, or from URNs to servers that know how to resolve them.

Traditional distributed query processing depends on coordinators, servers that must know all about data replication and statistics, to optimize a query. Mutant query plans have no need for such omniscient coordinators as they allow query optimization and source discovery to work with whatever information is available locally, and to proceed in parallel with query execution.

Compared to traditional, distributed query processing, mutant query plans trade away pipelining and parallelism for robustness, autonomous optimization at each peer and reduced deployment costs. A preliminary performance comparison, and ideas on how to hedge this bet and get some parallelism back are described elsewhere [PM02a].

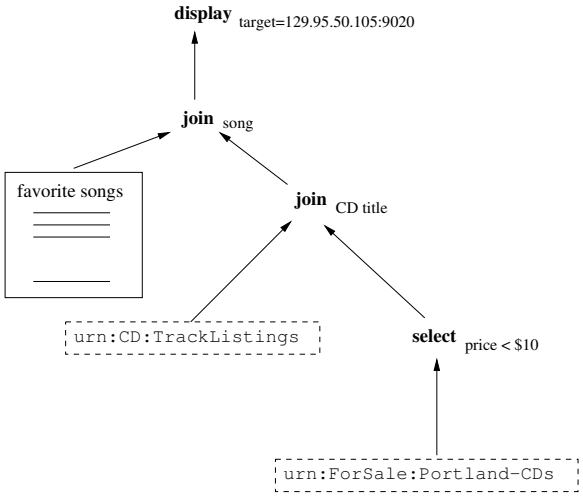

Figure 3: A mutant query plan.

MQPs present interesting new optimization issues (more elsewhere [PM02b]). Here is an example. Each server must materialize its partial results to ship the mutated query plan to the next server. We have to transfer these partial results over the network; their size matters. We can come up with query rewritings which would help MQP optimization that regular query optimizers would not normally consider. Suppose resources A and B are available locally, while X is not. If we know that $|A \bowtie B| \leq |A|$ we can reduce network traffic by rewriting $(A \cup X) \bowtie B$ into $(A \bowtie B) \cup (X \bowtie B)$, and evaluating the left branch. Depending on the number of B tuples that join with A, we may also opt to evaluate only the right outer join of A and B locally, by rewriting the plan to:

$\sigma_{B \neq \perp}(Z) \cup (\pi_B(Z) \bowtie X)$ where $Z = (A \bowtie = B)$

Mutant query execution, where each server in turn optimizes the plan, executes it and materializes the temporary results is reminiscent of INGRES-style query evaluation [SWK+76,WY76], where query decomposition interleaves sub-query execution and optimization.

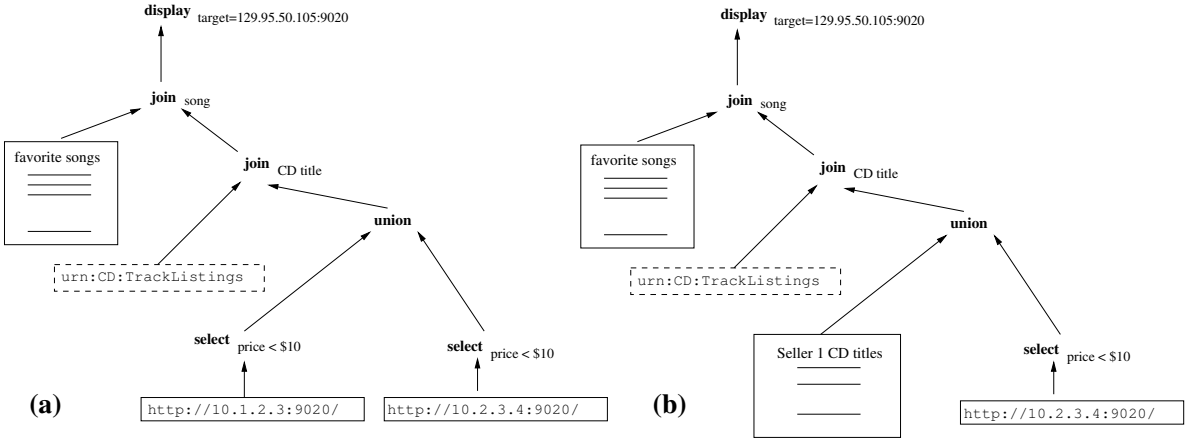

Figure 4: Two steps in the evaluation of a mutant query: (a) resolution and rewriting, (b) reduction.

3. Distributed Catalogs

In the previous section we glossed over an important issue with mutant query plans (and P2P systems in general): How do peers find out about resources available in other peers? In our example, how did we know that we could resolve the `ForSale` URN of Figure 3 into the union of two URLs in Figure 4(a)? For that matter, how did the user formulating the query know there was a `Portland-CD` resource to query over?

We want the P2P network to maintain distributed catalogs that can efficiently route queries to peers with relevant data. This index structure cannot scale unless it *is itself distributed or partitioned* among peers.

We believe that the main obstacle for building such distributed catalogs for file-sharing systems is the flat "filename" namespace, where any peer can potentially serve any file. In many applications, such as the P2P garage sale, we have much richer, structured metadata about our content.

In this section we describe how peers can use multi-hierarchic namespaces to categorize data; data providers use multi-hierarchic namespaces to describe the kind of data they serve and data consumers use them to formulate queries. We then detail the different roles that peers can play in the system, and the resource resolution process.

3.1. Multi-hierarchic namespaces

There are many ways we can use data attributes to organize data. For example we could use location to place items in the P2P garage sale into categories based on the seller's country, state, or city. This categorization is by no means the only one possible: we could also choose one of the various categorization schemes you find at online auction sites, or define categories based on price, weight, color, etc.

These categorization schemes can be flat (e.g. categorizing by price into items costing more than $100 vs. items costing less than $100), or hierarchical, with categories specified at different granularities or *levels*. `USA/OR/Portland` (all items located in Portland) is a city-level category, while `France` is a country-level category. We will call such multi-level categorizations *categorization hierarchies.* Within a categorization hierarchy, each item belongs to one (and only one) category called its *most-specific category*, and to all of its parents. For example every item in the `USA/OR/Portland` category also belongs in the `USA/OR` and `USA` categories.

The main idea behind our distributed catalogs is that for many P2P applications, the distribution of the underlying data among servers is not random. It is often the case that data are stored, grouped, replicated and queried according to one or more categorization hierarchies that are natural for the application. All the items sold by the same seller in the P2P garage sale will usually have the same address. If this address is in `USA/OR/Portland`, most prospective buyers will come from Portland, or locations close to Portland in the location hierarchy. People collect things: baseball cards, CDs, books, … If I am trading baseball cards, chances are that I have more than one.

Whenever this assumption holds, we can use these "natural" categorization hierarchies to build distributed indices for query routing. We call the set of categorization hierarchies relevant to an application domain a *multi-hierarchic namespace*. We will also borrow some OLAP terminology and call each hierarchy in a multi-hierarchic namespace a *dimension.* We assume that each dimension has an all-inclusive "top" category, called "*".

For simplicity in our example we will focus on just two dimensions: *merchandise* and *location*. Merchandise is the typical categorization scheme you can see in online auction sites such as eBay. An armchair, for example, might be classified under "`Furniture/Chairs`". Location is a three level country-state-city hierarchy. We can visualize dimensions as axes in a coordinate system. You can see parts of this multi-hierarchic namespace in Figure 5. The "coordinates" of an item in this system are expressed as n-tuples, e.g., [`USA/OR/Portland, Furniture/Tables`].

In Figure 5 you can see two *interest areas,* subsets of the cross product of the two dimensions. An interest area is made up of *interest cells*. An interest cell is the cross product of a category in the location dimension with a category in the merchandise dimension. Interest cells are also expressed as n-tuples. For example [`USA, Furniture`] is a cell that includes all pieces of furniture in the United States. Interest area (a) covers furniture in Vancouver ([`USA/WA/Vancouver, Furniture`]) and Portland ([`USA/OR/Portland, Furniture`]), while area (b) covers every item for sale in Portland ([`USA/OR/Portland, *`]).

We say that an interest cell *x covers* an interest cell *y* if, for every dimension in our namespace, the category of *x* for that dimension is a parent of, or the same as, the corresponding category of *y*. An interest area *a covers* an interest area *b* if every interest cell in *b* is covered by an interest cell in *a*. Two interest areas *overlap* if there exists a cell that they both cover.

Data providers use interest areas to describe the kind of data they serve. Data consumers also use interest areas to form queries. Suppose we are looking for second-hand armchairs in the Portland area. Our interest area is then [`USA/OR/Portland, Furniture/Chairs`] and we only have to contact servers whose interest areas overlap with ours to find out about all pertinent items.

3.2. Peer roles

At this point, we have defined enough terminology to describe the various roles that peers can perform in our

system. Peers can choose to perform one or more of the following roles:

• A *base server* maintains or replicates named collections of data within an interest area. A seller in our P2P garage-sale example might have an interest area of [USA/OR/Portland, Music/CDs].

• An *index server* keeps track of base servers, and other index servers with interest areas overlapping its own. An index server for the P2P garage sale could index, for example, servers overlapping [USA/OR, *]. Index servers can also maintain indices on data attributes not used for categorization, e.g., price.

• A *meta-index server* is an index server that maintains *only* multi-hierarchic namespace indices, keeping track of base, index and meta-index servers with interest areas overlapping its own.

• A *category server* can answer queries about the dimensions themselves (e.g., "What are the immediate subcategories of Furniture? ").

An index server's entry for a base data item includes a URL (containing host name and port number of the base server) and an XPath [CD99] expression (the base server's identifier for the collection). For example, an index server for [USA/OR, SportingGoods] might include a reference to golf clubs available at a seller's site as (http://10.3.4.5, /data[id=245]).

Note that a server's interest area completely describes its data, but this does not guarantee that the server stores or indexes *all* the data in that interest area; there is no way to make such a statement in our system (although we do allow relative statements such as "Server *A* contains all of Server B's data, for this interest area" – see Section 4).

There is a tradeoff between a server's index area, and the detail of the indices it maintains, which is the reason for having both an index server, and a meta-index server role. The richer these extra indices are, the better we can route a query. On the other hand, extra indices use up resources, and have to be updated when their base data change, thus limiting their scalability. Meta-index servers can afford to cover much larger interest areas than index servers, because they only maintain multi-hierarchic namespace indices.

Peers can maintain caches with index and meta-index servers they used in the past. A peer that joins the P2P network for the first time will have to discover category servers, and also meta-index servers that serve top-level categories, for example a meta-index server that covers [France, *], and it obviously cannot use the P2P network for that. Peer software can either include hardwired locations of such servers, or preferably discover them out-of-band, for example by doing a search on a web search engine.

3.3. Authoritative servers

An *authoritative server* strives to know about all base servers within its area of interest. Routing a plan through an authoritative index or meta-index server will allow it to find out the known base servers in a particular interest area. A more realistic scenario is that a *group* of servers chooses to stay authoritative for an area, guaranteeing that the *union of their answers* includes paths to the relevant base servers.

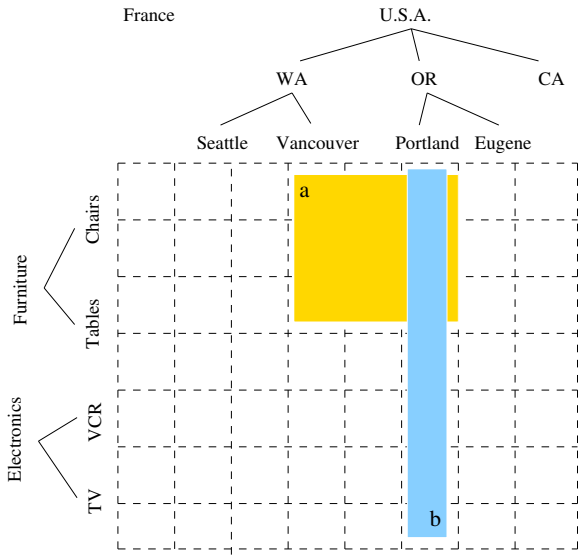

Figure 5: A multi-hierarchic namespace with two categorization dimensions and two highlighted interest areas: (a) Vancouver-Portland furniture, (b) items in Portland.

A base server joining the P2P network needs to register with index or meta-index servers that intersect with its interest area, to make their data available to other peers. Ideally, the servers it registers with should include authoritative servers whose union covers its interest area. Thus servers with more specific interest areas *push* the data about their existence to an authoritative server that covers them. We can also have a complementary *pull* process, where index servers query their base servers for their data, to build more detailed indices.

An index or meta-index server that wishes to become authoritative for an interest area must first find the most detailed authoritative server group that covers it. At that point, the server must register with the other servers in that group so that it can start receiving registrations and updates from servers within its interest area, and also start receiving queries. Again, update propagation can be a pull, or a pull process.

3.4. Resource resolution

To form queries, we can encode interest areas into the "namespace-specific string" part of URNs, which we will from now on treat as structured entities instead of opaque strings. For example we can encode interest area (a) in Figure 5 as:

"urn:InterestArea:(USA.OR.Portland,Furniture)+(USA.WA.Vancouver,Furniture)"

Encoding is a purely lexical process of transliterating our interest area notation to URN syntax.

A server trying to resolve such a URN should first seek an authoritative index or meta-index server that covers it, and recursively follow the index references until it finds all the relevant base servers and data items (or until it finds *enough* data items, in case the user asked a top-n type of query).

In our example query of Figure 3, the URN we are trying to resolve has an interest area of [USA/OR/Portland, Music/CDs]. Our client may already know an authoritative meta-index server for [USA, *], so it sends the query plan there. This server may forward the query plan to a server for [USA, Music], which may then forward it to a server that knows about [USA/OR, Music] and so on, until we reach an index server that will replace the URN with a combination of URLs, such as the one in Figure 4(a). To avoid flooding high-level servers with plans, peers maintain caches of index and meta-index servers for interest areas, so that they can route plans more efficiently in the future.

There is no guarantee that we can find an authoritative server for every query. It may very well be that we cannot find *any* servers for some part of a query's interest area, or that, to get a complete answer, we may have to contact *multiple* servers that collectively cover an interest area. It is usually in a data provider's best interest to register its data with one or more authoritative servers (sellers in our garage-sale example would do that to reach the widest possible audience). However, unless we are in a restricted context (e.g. a corporate intranet) where data providers can be *compelled* to do so, we cannot provide any absolute service guarantees. Fortunately, as with most internet services, users have learned not to expect them.

3.5. Category servers

Category servers maintain data about the categorization hierarchies themselves. Categorization hierarchies can be administered independently of each other (you can imagine a location hierarchy managed by the Post Office).

Since our system uses categories for both index construction and query formulation, it is important that they are relatively stable and consistent. Fortunately, we can expect hierarchy nodes at higher levels to be more stable (countries and state names will change less frequently than zip codes or road names). Also, since nodes in a hierarchy properly contain their descendents, we can approximate a reference to a hierarchy node we don't know about with a reference to one of its ancestors. For example, we could rewrite a reference to USA/OR/Portland into USA/OR, with a possible loss of precision, but no loss of recall.

As with index and meta-index servers, category servers can cooperate with each other to manage their namespaces. Category servers can delegate portions of the namespace they manage to other category servers, much like the way DNS servers can delegate sub-domains to other servers.

4. Completeness, Redundancy, Currency and Latency

In this section we discuss how index and meta-index servers can convey the relationships between the data they cover, and how mutant queries can use this information to make intelligent choices about completeness, currency and latency tradeoffs.

To simplify the formulas, from now on we will specify coordinates using only their most detailed levels, for example we will write Portland instead of USA/OR/Portland wherever the meaning is clear from context. We will also use res(E) to denote the result of evaluating the query expression E.

4.1 Completeness and Redundancy

In the distributed catalog architecture we described in the previous section, meta-index servers map interest areas to collections of URLs at index or base servers (or possibly other meta-index servers). The implicit semantics is that the interest area is covered by the union of those URLs. This simple interpretation is problematic for two reasons. One is that some of the servers may be wholly or partially redundant with others. For example, an index server on [Portland, Sporting Goods] and a server on [Oregon, Golf Clubs] could be redundant on a query involving [Portland, Golf Clubs]. The second problem is that one can't know for sure when one has consulted enough meta-index servers. Will the next one reveal previously unknown index or base servers for an interest area, or just previously discovered ones? Our catalog scheme can thus benefit if servers can also announce their policies to replicate or index information at other servers. The simplest such "intensional statement" for a server is for it to say that it will exactly duplicate the contents of another server (meta-index, index or base). More useful is an intension to replicate on an area of interest. For example, server R might replicate everything from server S for the Portland category of the Location hierarchy. We can express this intensional statement as

```
base[Portland, *]@R =
 base[Portland, *]@S.
```

(Such a statement is a an instance of a coordination formula, as defined by Bernstein et al. [BGK+02], with a simplified syntax.)

A more complicated relationship between R and S might be

```
base[Oregon, Sporting Goods]@R =
 base[Portland, Golf Clubs]@S ∪
   base[Eugene, Golf Clubs]@S.
```

Here we see that the only Oregon sporting goods information that R holds is for Portland and Eugene golf clubs at S. Note that this intensional statement is not equivalent to the pair

```
base[Portland, Golf Clubs]@R =
 base[Portland, Golf Clubs]@S
 base[Eugene, Golf Clubs]@R =
 base[Eugene, Golf Clubs]@S.
```

With those statements, R might also contain data items for `[Oregon, Sporting Goods]` that are not from Portland or Eugene, or that are not golf clubs.

The replication can occur at the meta-index or index levels as well, for example

```
index[Portland, *]@R =
 index[Portland, *]@S.
```

We can also capture connections across different levels at different servers. For example, to indicate that R's index on Oregon golf clubs covers exactly the base records at S, we write

```
index[Oregon, Golf Clubs]@R =
 base[Oregon, Golf Clubs]@S.
```

More likely, R will index several base servers. If for example, it covers base data at servers S, T and U, the intensional statement is

```
index[Oregon, Golf Clubs]@R =
base[Oregon, Golf Clubs]@S ∪
base[Oregon, Golf Clubs]@T ∪
 base[Oregon, Golf Clubs]@U.
```

In general, exact replication will be too strict. It may be that R wants to replicate everything for Portland at S, but also possibly keep additional data about Portland. In that case, the intensional statement is

```
base[Portland, *]@R ≥
 base[Portland, *]@S.
```

That is, R knows everything that S does about Portland, and possibly more.

4.2 Utilizing Intensional Statements

How are such intensional statements used in the processing of MQPs? First of all, whenever a server registers an interest area with a meta-index server, it can also provide intensional statements that the meta-index server can retain. Servers can then use such information in binding and routing MQPs. To incorporate information from intensional statements into an MQP, we allow a new operator, "or", in plans. An "or", written as 'I', can be viewed as a "conjoint union", saying that either expression it connects holds the necessary data (or index information). The essential semantics of "or" are captured by the pair of rewrite rules

```
A | B → A
A | B → B.
```

Example 1: Assume meta-index server M knows about servers R and S, with interest areas `[Portland, Recreation]` and `[Oregon, Sporting Goods]`, respectively. Suppose M receives an MQP that contains the resource name `[Portland, Golf Clubs]`. With the basic catalog structure, as described in Section 3, that name could be bound to

```
base[Portland, Golf Clubs]@R ∪
 base[Portland, Golf Clubs]@S.
```

If in addition M knows the intensional statement

```
base[Portland, Sporting Goods]@R =
 base[Portland, Sporting Goods]@S,
```

it could bind to

```
base[Portland, Golf Clubs]@R |
 base[Portland, Golf Clubs]@S.
```

Then the MQP could be routed to either R or S, but it need not go to both.

Example 2: Consider this intensional statement about index coverage:

```
index[Oregon, Golf Clubs]@R =
base[Oregon, Golf Clubs]@S ∪
base[Oregon, Golf Clubs]@T ∪
 base[Oregon, Golf Clubs]@U.
```

In an MQP, the resource name `[Portland, Putters]` can be bound to

```
index[Oregon, Golf Clubs]@R |
base[Oregon, Golf Clubs]@S ∪
base[Oregon, Golf Clubs]@T ∪
 base[Oregon, Golf Clubs]@U.
```

The MQP can then be routed to R (and to S, T and U as needed) or directly to all of S, T and U, in some order.

Example 3: Let us consider containment statements. Assume a meta-index server M knows about servers R and S, and the intensional statement

```
base[Portland, *]@R ≥
 base[Portland, *]@S.
```

Suppose M receives an MQP with resource name `[Portland, CDs]`. One possible binding for this name is

```
base[Portland, CDs]@R.
```

However,

```
base[Portland, CDs]@R |
base[Portland, CDs]@R ∪
 base[Portland, CDs]@S
```

is also correct. This second binding might not seem particularly useful at first glance. However, there are conditions where it makes sense. One is if the MQP passes through server S for other reasons, and evaluation at S can reduce intermediate result size. Consider if the MQP contains the (partially evaluated) sub-expression

```
res(E) - πC(base[Portland, CDs]@R ∪
      base[Portland, CDs]@S).
```

That expression can be transformed to

```
(res(E) - πC(base[Portland, CDs]@S)) -
      πC(base[Portland, CDs]@R,
```

in which the first difference can be evaluated, and may be much smaller than `res(E)` itself. Other reasons to prefer the second binding is that R may be unavailable at some point, and we can use S for a partial answer to the query, or that R replicates S with a delay (see the next section) and we want a more current answer.

4.3. Currency and Latency

In a loosely coupled Internet setting, it is impossible to guarantee that queries run *instantly* against the *complete*, *latest* information. There will be compromises on latency, completeness and currency. However, we would like a query issuer to have some control over the tradeoffs made. For example, a user may be willing to sacrifice completeness for a fast answer, or prefer completeness to currency in a query with a fixed time budget.

We also recognize that replication between servers cannot be both scalable and instantaneous. More likely, servers will periodically contact other servers to update content. We therefore extend intensional statements to include a possible delay factor. For example, suppose server R polls every 30 minutes to update the data it replicates from S. We can express that intension as:

```
base[Portland, *]@R ≥
base[Portland, *]@S{30},
```

saying that R replicates everything at S for Portland, but can be up to 30 minutes out of date. Referring back to Example 3, a binding for resource `[Portland, CDs]` might then be

```
base[Portland, CDs]@R{30} |
(base[Portland, CDs]@R ∪
base[Portland, CDs]@S){0}.
```

This binding indicates that one can get an answer (more) quickly by just routing the MQP to R, but that answer could be up to 30 minutes out of date. Alternatively, by routing the MQP to both R and S, one can have a complete and current answer (modulo the delay to finish evaluating the MQP and routing it back to the client). The latency for query evaluation will likely be longer in the second case, because of the need to visit two sites rather than one.

One can imagine quite rich schemes for expressing user preferences among latency, completeness and currency, and query processing strategies to meet those preferences. Our initial inclination is to start with something simple: a query carries a target evaluation time (e.g., 30 seconds) plus a binary preference for complete versus current answers. Even with such a simple expression of tradeoffs, we expect to develop non-trivial methods for binding, evaluation and routing.

5. Issues and Extensions

In this section we discuss several possible extensions to our framework, as well as issues of security and privacy.

5.1. Carrying Additional Information in MQPs

We have described the process of MQP evaluation in terms of binding URNs to URLs of indexes and data, thence to the data itself, and replacing evaluable sub-expressions with their results. Our initial implementation works in this manner. However, we see reasons to support annotation of URNs and URLs, and also to retain a copy of the original query plan in the MQP as it gets evaluated. We describe two uses of such information.

Accumulating catalog and statistics information. As an MQP passes through a server, that server may have information about portions of a query it chooses not to evaluate, but that may be useful at later processing steps. For example, consider a server S that gets an MQP P = $\pi_C(\sigma_D(A) \bowtie B)$ to evaluate, and has B but not A. Suppose B has a million elements in it. S may decline to evaluate B at this point, because of the size of res(B). Rather than forwarding P intact to another server, S could annotate B with its cardinality, the unique cardinality of the join column, or even a histogram. Other servers could then avoid sending P back to S (or another server for B) until there was enough additional data in P to give a smaller result at S. Maintaining the original query along with the partially evaluated query also allows a server to improve or enhance bindings (or even undo them). For example, a server could add other possible URLs for a URN that it knows. A server can also improve its catalog information by examining a URN in the original query and its set of URLs in the partially evaluated query.

Maintaining provenance. An MQP can also carry along a history of all the servers it has visited, as well as what each one did (provided bindings, provided data, re-optimized the MQP, evaluated a sub-expression, or merely forwarded the MQP), when it did it, and how current the information was. That provenance can then be used at the final destination or at intermediate servers for a variety of purposes:

Judging the quality of an answer: Knowing the processing history of a query can allow judgment about the currency or completeness of the result.

- Rewards system: If server S observes that many of the queries it is getting for its data are because of indexes maintained at server T, S might reward T in some way. For example, S might devote a larger percentage of its index space to T's data in return.
- Meta-index updating: If server S is getting a lot of MQPs forwarded from server T that it just ends up forwarding to server R, S might be able to send T a meta-index entry to allow it to route some of those queries directly to R. Or S might observe that T declines to bind source B even though T holds a copy of B. S might then decide to route MQPs needing B elsewhere in the future.
- Detection of spoofing: To this point, we have been assuming that MQP servers behave correctly, and certainly not maliciously. But what if server S tried to tinker with queries to the detriment of a competitor's

server T? For example, server S may get an MQP P with an expression $\sigma_D(A) \cup \sigma_D(B)$, where A has data records at S and B has records at T. S could bind A to its actual value, but bind B to the empty set, making it appear that T has no qualifying items. If provenance is recorded, the resulting MQP would show that P never visited T (or any other site for B). If A also spoofs the provenance, to make it appear T participated, then it possible to construct a *verification query* (e.g., count($\sigma_D(B)$)) to send to T to check the result in P. To make the provenance more trustworthy, each addition to it could be digitally signed by the server that adds it and encrypted with the public key for the destination site. However, provenance is not a complete solution to a misbehaving server. In the example above, it is hard to detect if S is lying about A's contents.

5.2. Security and Privacy

As with any distributed application, issues of security and privacy arise as soon as data is sent from one site to another. In-transit security for MQPs is neither more difficult nor less difficult than for other distributed query approaches. Security and privacy at MQP servers does raise some new issues, however. In a coordinator/ subordinate model of distributed query processing, only queries, and not data, are sent to the subordinate servers. (This assumption does not hold if semi-joins [BC81] are being used for distributed processing.) With MQPs, data, in the form of partial results, is divulged to other, possibly unknown, servers, which may be undesirable. For example, a query submitter might not want his or her music preferences known to a track-list server. Or an intermediate server might not want its data exposed to a competitor's server down the line. Thus, MQPs will need to incorporate ordering and transfer policies, such as "do not bind `preferences` until `playlist` is bound" or "only let this MQP pass through servers on this list." Obviously, such restrictions will be challenging to support in general in a loosely coupled environment. An alternative in some cases will be to encrypt data or data elements with the public key of the result recipient, although encrypted data can limit evaluation options en route.

However, we point out that MQPs can allow a query submitter to obtain answers that might not be obtainable under given server security policies with conventional distributed query processing. It may be that two servers will allow data to pass between themselves that they will not directly divulge to a third party. For example, suppose a law enforcement agency wants to know which employees of a given company have made charitable contributions over $5000 to organizations that are believed to be fronts for illegal activities. The IRS has tax returns showing itemized deductions for contributions, and the State Department has a list of front organizations. But the IRS may balk at disclosing all contributions for all employees at a company, and the State Department may not want to reveal its list of suspect organizations. If, however, the IRS is willing to pass data to the State Department (knowing how it will be used from the query), then an MQP for this query can be executed in the following manner. The MQP first goes to the IRS, where names of people are found who work for the company (from W-2 forms) together with charity names for charitable deductions over $5000 (from Schedule A). That information is then bound into the MQP, which travels to the State Department. There, the results from the IRS are joined with the front-organization list, and then projected onto person name. The fully evaluated MQP now is routed back to the law enforcement agency. Neither the IRS nor the State Department had to disclose excessive sensitive information to the agency.

6. Related work

You can find a general introduction to mutant query plans, our prototype implementation, and a preliminary performance comparison with traditional pipelined distributed query execution in previous work [PM02a, PM02b]. Query optimization issues for mutant query plans include: *consolidation* (rewriting a plan so that locally evaluable sub-plans come together), *absorption* (plan rewritings that might not make sense in pipelined query execution but reduce the size of the partial result), and *deferment* (avoiding local execution of operators that increase the partial result size unjustifiably).

Categorizing things in hierarchies is of course not a new idea — humans have been doing it for millennia! DNS [AL01] and LDAP [HS97] are examples of widely deployed systems based on hierarchical namespaces. DNS in particular has managed to scale admirably with an exponential growth in the amount of data it indexes and serves (mappings from human-readable machine names to IP addresses). Each DNS server covers a well-defined address space, and new "branches" (domains) can be added to accommodate growth. Most DNS queries (host-name resolutions) can be served out of caches, and clients of the system only have to contact a few servers at most to resolve any host name. The contribution of our system is the combined categorization of data into *multiple* hierarchies, to accommodate different types of users with different viewpoints and ways to group data together.

Gribble et al. [GHI+01] present the benefits of transplanting established data management techniques, stronger semantics, and theoretical underpinnings of databases to P2P networks. Bernstein et al. [BGK+02] introduce the *Local Relational Model* (LRM), a data model for P2P database applications. In the LRM, peers use declarative *coordination formulas* to describe the relationships and constraints between their schemas. Intensional statements, as we used them to describe relationships between index and meta-index servers, can be expressed using coordination formulas.

Distributed hash table (DHT) algorithms are a very active research area. Systems such as CAN [RFH+01], Chord [SMK+01], Pastry [RD01], and Tapestry [ZKJ01] offer a scalable hashtable interface with extremely fast lookups (usually logarithmic in the number of hosts). Fast key lookups by themselves, however, cannot provide the data manipulation capabilities we regularly expect from a database – what about range queries, or joins? Harren et al. [HHH+02] analyze these issues, and describe the missing pieces (including a hierarchical namespace) we need in order to build a query processor on top of DHTs.

An assumption that is frequently made for DHTs is that the system decides which nodes route, index (and sometimes store or cache) which data, for the benefit of the whole. We assume a much more loosely federated system, where these decisions are left to the peers, and depend on the application. In our gene expression data scenario, we expect that many laboratories would serve as the "authoritative" sources for their own data, and volunteer to index or cache data in related areas. Government agencies, such as the NIH, would provide meta-index services, and fund the development of controlled vocabularies and ontologies. An interesting alternative would be Mariposa's microeconomic paradigm [SAL+96], where peers buy or sell data objects, and place bids to execute subqueries.

Describing server holdings with interest areas is an instance of *summarization*, an idea widely used in the OLAP community. Walker [W80] analyzed the conditions under which a query over a summarized database gives correct answers, and proposed a *succinctness* ordering for comparing the quality of inexact answers. Lakshmanan et al. [LNW+02] present a generalization of the minimum description-length principle for summarization, which can lead to fewer summary regions, by allowing regions to contain "don't care" cells.

Our ideas on intelligent routing of query plans based on intensional statements about server coverage, completeness and redundancy are a form of semantic query optimization (SQO). Chakravarthy et al. [CGM90] used first-order logic to formalize an SQO framework for deductive and relational databases. Levy and Sagiv [LS95] studied the effects of allowing recursive rules, order constraints and negated subgoals in the rules and integrity constraints of a deductive database. Grant et al. [GGM+97] applied SQO techniques to object databases. Hsu and Knoblock [HK00] used SQO to optimize distributed queries, both at a local level (e.g., by eliminating redundant joins) and at a global level (e.g., by minimizing data transmission).

Yang and Garcia-Molina [YG01] compared the performance of several *hybrid* P2P architectures for file-sharing networks, using both analytical models and experimental data. In hybrid P2P architectures peers transfer data autonomously, but depend on one or more central servers for indexing and querying. They have also compared the performance of various search strategies for *pure* P2P architectures [YG02], using data from the Gnutella network. Crespo and Garcia-Molina [CG02] proposed using *Routing Indices* (RI) to direct queries in pure P2P networks. RIs are distributed indices, maintained at each node, that guide each query to the most promising neighbors of the node (in terms of number of relevant documents and their network distance). Since RIs record promising *directions*, and not addresses, they have reasonable storage requirements.

Galanis et al. [GWJ+02] presents a system for running IR-style containment queries over the documents in a P2P network. Their system maintains *Peer Inverted Indices* (PIs) at each node. PIs map keywords (whose number is usually much smaller than the number of documents) to peers with documents that contain them. Bayardo et al. [BAG+02] implemented a system called *YouServ* for hosting web content, where peers collaborate to host replicas of each other's web sites. YouServ uses dynamic DNS to map URLs to the replicas currently serving them.

The major difference between our proposal and most of the work on file-sharing systems is granularity: In a file sharing system, peers can only store, replicate, index, and query data in whole-file chunks; we, on the other hand, allow peers to deal with semi-structured data at any level.

7. Conclusions

We presented our framework for distributed data management based on mutant query plans and multi-hierarchic namespaces. Mutant query plans enable peers to independently optimize and partially evaluate queries without global knowledge, and with a minimum of coordination overhead. Our main assumption is that data and query result distributions can be mapped naturally to multi-hierarchic namespaces, allowing us to build decentralized indices for efficient query routing. We believe that this assumption is reasonable, not just for our P2P garage sale example, but for a wide range of P2P data management applications.

Acknowledgements

We would like to thank Juliana Freire, Lois Delcambre and Pete Tucker for their suggestions and ideas. Funding for this work was provided by DARPA through NAVY/SPAWAR contract N66001-99-1-8908 and by NSF ITR award IIS0086002.

References

[AL01] P. Albitz and C. Liu. *DNS and BIND.* (4th Ed.) O'Reilly and Associates, 2001.

[BAG+02] R. J. Bayardo Jr., R. Agrawal., D. Gruhl and A. Somani. YouServ: A Web Hosting and Content Sharing Tool for the Masses. *In Proc. of WWW 2002*.

[BC81] P. A. Bernstein, D.-M. W. Chiu. Using Semi-Joins to Solve Relational Queries. *JACM* 28(1): 25-40, 1981.

[BHQ+01] A. Brazma et al. Minimum Information About a Microarray Experiment (MIAME) - Toward Standards for Microarray Data. *Nature Genetics* 29(4):365-371, Dec. 2001.

[BGK+02] P. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data Management for Peer-to-Peer Computing: A Vision. In *Proc. of WebDB 2002*, pages 89-94.

[CD99] J. Clark and S. DeRose (editors). XML Path Language (XPath) Version 1.0, November 1999. Available at: http://www.w3.org/TR/1999/REC-xpath-19991116

[CDB] Gracenote's CDDB: http://cddb.com/music

[CG02] A. Crespo and H. Garcia-Molina. Routing Indices for Peer-to-Peer Systems. In *Proc. of the Int. Conf. on Distributed Computing Systems*. July 2002.

[CGM90] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-Based Approach to Semantic Query Optimization. *ACM TODS* 15(2):162-207, June 1990.

[FDB] FreeDB: http://freedb.org/

[GHI+01] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can Databases do for Peer-to-Peer? In *Proc. of WebDB 2001*.

[GGM+97] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic Query Optimization for Object Databases. In *Proc. of ICDE 1997*, pages 444-453.

[GWJ+02] L. Galanis, Y. Wang, S. R. Jeffery, D. J. DeWitt. Processing XML Containment Queries in a Large Peer-to-Peer System. Available from: http://cs.wisc.edu/niagara/papers/603i.pdf

[HHH+02] M. Harren, J. M. Hellerstein, R. Huebsch, B. Thau Loo, S. Shenker, I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *IPTPS 2002*, pages 242-259.

[HK00] C.-N. Hsu and C. A. Knoblock. Semantic Query Optimization for Query Plans of Heterogeneous Multidatabase Systems. *IEEE Transactions on Knowledge and Data Engineering,* 12(6), pp. 959-978, 2000.

[HS97] T. A. Howes, and M. C. Smith. *LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol.* Macmillan, 1997.

[LNW+02] L. V. S. Lakshmanan, R. T. Ng, C. Xing Wang, X. Zhou, and T. J. Johnson. The Generalized MDL Approach for Summarization. In *Proc of VLDB 2002.*

[LS95] A. Y. Levy, and Y. Sagiv. Semantic Query Optimization in Datalog Programs. In *Proc. Of PODS 1995,* pages 163-173.

[NDM+01] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The Niagara Internet Query System. *IEEE Data Eng. Bulletin* 24(2):27-33, 2001.

[PM02a] V. Papadimos and D. Maier. Mutant Query Plans. *Information and Software Technology*, 44(4):197-206, April 2002.

[PM02b] V. Papadimos and D. Maier. Distributed Queries without Distributed State. In *Proc. of WebDB 2002*, pages 95-100.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM 2001*, pages 161-172.

[SAL+96] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, A. Yu. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal* 5(1):48-63, 1996.

[SMK+01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM 2001*, pages 149-160.

[SWK+76] M. Stonebraker, E. Wong, P. Kreps, G. Held. The Design and Implementation of INGRES. *ACM TODS* 1(3): 189-222, 1976.

[W80] A. Walker. On Retrieval from a Small Version of a Large Data Base. *In Proc. of VLDB 1980.*

[WY76] E. Wong, K. Youssefi. Decomposition - A Strategy for Query Processing. *ACM TODS* 1(3):223-241, 1976.

[YG01] B. Yang and H. Garcia-Molina. Comparing Hybrid Peer-to-Peer Systems. *In Proc. of VLDB 2001*.

[YG02] B. Yang and H. Garcia-Molina. Improving Search in Peer-to-Peer Networks. In *Proc. of the International Conference on Distributed Computing Systems 2002*, pages 5-14.

[ZKJ01] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Report UCB/CSD-01-1141, U. C. Berkeley, 2001.