Adaptive Routing Filters for Robust Query Processing in Schema-Based P2P Systems

Katja Hose Marcel Karnstedt Kai-Uwe Sattler Ernst-August Stehr
Department of Computer Science and Automation, TU Ilmenau
P.O. Box 100565, D-98684 Ilmenau, Germany

Abstract

Peer Data Management Systems (PDMS) currently gain attention at an emerging scale in order to cope with the needs of growing organizational integration. Efficient query processing, as one of the main requirements in these systems, provides three major challenges: achieving robustness, scalability and self organization. In this paper we deal with the physical aspects of these requirements. We introduce an adaptive maintenance technique based on query feedback for keeping routing filters, used to optimize routing, up-to-date. These filters are applied in conjunction with an iterative query processing strategy and we show that this can improve robustness and scalability of query processing in distributed data management systems.

1 Introduction

P2P systems are massively distributed and decentralized. Schema-based variants of P2P systems are, among others, used for data integration purposes. These systems are also called *Peer Data Management Systems* (PDMS) and they bridge the gap between P2P networks and database systems. The coordinated interaction of all peers forms a users view on one virtual database. This work addresses the physical aspects of efficient query processing in PDMS. We introduce an adaptive query feedback technique used to build and keep routing filters up-to-date. These routing filters are used for optimizing routing, which prevents from flooding the network and therefore leads to improved scalability in PDMS. The usage in conjunction with an iterative processing strategy also improves robustness and the collaborating peers are able to process queries in a totally self-organizing manner.

Early approaches based on simple flooding do not scale satisfyingly. Improved query processing strategies, even those developed for distributed systems, are not suitable in order to fully cope with the challenges in P2P systems. One basic idea of these strategies is to plan and optimize queries statically *before* evaluating them. This will not lead to satisfying results, because in P2P systems there is

(i) not enough knowledge available at one peer
(ii) the dynamic character clearly requires dynamic optimization and adaptive techniques

Query processing must be optimized at each peer again, taking different knowledge and changing situations of the network into account.

Several contributions deal with semantic aspects of schema-based PDMS, e.g., [10]. Our work does not focus on the logical level of query processing (schemes and mappings), though it is an indispensable part of our query engine. For more details on this we refer, among others, to the work cited above.

The rest of this paper is organized as follows. Section 2 presents related work. In Section 3 we present an incremental query processing strategy, developed in order to overcome disadvantages resulting from the high dynamic in P2P systems. Section 4 deals with the statistics the strategy uses and how to maintain them efficiently using an adaptive approach based on query feedback. In Section 5 we take a brief look at some important results gained using a P2P simulator and finally Section 6 concludes. This article is the short version of a more detailed work[1].

2 Related Work

To the best of our knowledge contributions concerning the pure physical processing of queries in P2P systems are rare. [6] gives a good survey of strategies for distributed systems in general, but lacks in details and the reference to ”pure” P2P systems. In [8] *Mutant Query Plans* were introduced, a technique we adapted to our needs. This is one of the first works presenting a physical processing strategy, but their algorithms still need to be optimized for P2P systems. One main disadvantage is the missing ability to process a query in parallel on different peers. The problem with most works is that they do not focus on all three main requirements we have to meet in P2P systems simultaneously or that they do not explain how they are met in detail: scalability, robustness and self-organization.

[1] http://mordor.prakinf.tu-ilmenau.de/papers/dbis/2005/TR05adrf.pdf

The problem of routing queries in P2P systems where neither central instance nor global knowledge exists was studied sufficiently ([2, 4, 3]). The problem is mostly solved by introducing some kind of index structure – locally on each peer, distributed across all or some peers, central indexes (which actually is not "pure" P2P) - or similar approaches. A recent paper [5] for instance utilizes hierarchical bloom filters to overcome the peers' limited knowledge. Index data structures are the containers for statistics, which are used in cost models and for routing decisions. Systems where routing, query optimization and maybe even processing are delegated to so-called super-peers can be interpreted as specializations of the systems investigated in our work. An important problem in the context of routing as well as of cost-based optimization is the maintenance of the underlying statistics in order to keep them up-to-date. As we argue in this paper the usage of corresponding update messages is not suitable, because this behavior leads to a message explosion and therefore will not scale. Alternative approaches are based on query feedback. This was introduced for relational systems by [9], as well as for XML data in distributed systems, e.g. by [7].

3 Query Processing Strategies

3.1 Basics of Query Processing

For the remaining we assume the following:

- XML is the natural data format for all peers, queries are formulated using a subset of XQuery (we essentially focus on the contained XPath expressions).
- No global knowledge or central instance is available, no kind of distributed hash table (DHT) or similar distributed index structure is used.
- Pairwise bidirectional mappings are defined between connected peers (neighbors).
- Each peer has equal logical and physical processing capabilities. Any query may be initiated at any peer.

This is a simple example of a supported XPath selection:

```
//object[propermotion/pmra='13052']/coordinates
```

In general query processing strategies for distributed networks can be divided into data shipping (DS) and query shipping (QS) based approaches [6]. In preceding works, e.g., [4], we implemented a QS approach as an extended combination of mutant query plans [8] and the approach presented in [6]. Among others we showed that this better suits for high-scaled P2P systems than a pure DS approach. A main aspect of that strategy is the utilization of local index structures used to optimize routing of a query if enough information is available – if not, we flood the network, i.e., the query is forwarded to each neighbor (Flood If No Route, FINR for short).

3.2 Routing Filters

Since the main focus of our work is reducing execution costs by routing a query efficiently, this subsection describes the structure of our *routing filters* that contain all information our strategy needs for this purpose.

Routing filters are local index structures established at each peer from which we can extract distributional information on instance and schema level. This approach is based on routing indexes [2]. Summarizing cost factors in a local index structure at each peer we can apply a simple model for query routing, which builds a basis for establishing a distributed, dynamic cost model in order to implement adaptive processing techniques. The routing filters compound information about data obtainable by the connections to all neighboring peers, each peer is restricted to limited knowledge about global data location and other participating peers by limiting the filters to a maximal *hop count*. Each peer holds one separate filter instance for each neighbor. Establishment and maintenance of the filters is implemented using a query feedback approach. In this way we can adapt to actual workloads and query loads, times of high traffic, low bandwidth, etc. Moreover, we do not need any peer to provide more information than it is willing to provide when joining the network. We simply extract the needed information from what is returned to processed queries.

The structure of the routing filters is based on *path trees* [1], an adaptable summary structure for XML data. We define a path tree as a deep copy of the actual XML data. However all siblings (elements with the same name and the same ancestors) are merged into one tree node. Path trees can represent the full schema of known data and may be pruned with low information loss in order to meet lower memory resources. In order to optimize the physical processing of a constraint based search (a constraint denotes everything in the brackets of an XPath query, e.g., `[propermotion/reference='J2000.0']`), each of the filter elements has a two-dimensional histogram matrix associated. In this way we collect information on instance level additionally to schema information.

One of the dimensions in the histogram is a division into buckets, as they are used in classical database systems (we use equi-width histograms, but others are suitable as well). The second dimension is conceptionally a time axis of expected returns. histogram$[i][j]$ denotes the total count of all data records in bucket j, that are expected to be returned within n time steps, where n is a function of i, e.g., $n = i^2$. This allows us to use the filters not only to estimate the number of returned documents, but even estimate the number of documents in the remaining time to return of the query. By switching our concept of limiting knowledge from hop count to timeout, we adapt to the real network load as soon as we have gained this information by the actual round-trip-times of queries. Details about the utilization of this methodology will follow in Section 4.

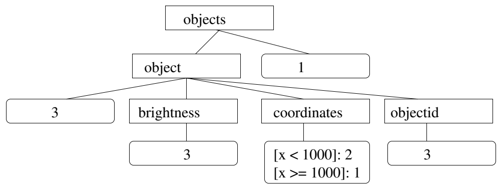


Figure 1. Part of a routing filter

Figure 3.2 pictures a part of an example routing filter, depicting schema elements grouped together in the path tree with the corresponding count of occurrences in the XML data. For simplicity we indicate only one very simple histogram with two coarse imaginary buckets at the schema element 'coordinates'. The example is based on astronomical data. Such and similar XML schemes are used by observatories that catalog their collected sky data like pictures of astronomical objects and corresponding meta data.

3.3 Incremental Strategy

In former works we implemented a query processing strategy based on QS (see [4] for details). Evaluating the strategy we realized the following main challenges:

- networks sized at scales as large as Internet-scale or similar sizes
- dealing with unknown global distributions of data and unknown execution costs
- coping with highly dynamic networks, especially networks with high failure frequencies

In order to overcome these challenges we developed a merely stateless, incremental and dynamically optimizing query processing strategy named IMS. The main differences to the QS strategy and thus advantages of IMS are based on two main principles: *(i)* trying not to wait at all in any stage of processing and *(ii)* sending first results as fast as possible back to the initiator.

```
Input:  Message $mess$
1   if $mess$ is query and no cache entry with same ID existing then
2       if $mess.TTL > 0$ then forward-query(get-neighbors(), $mess$); fi
3       process-query-locally($mess$);
4       add-to-cache($mess$);
5       if not initiator and relevant local data existing then
6           send-answer($mess$);
7       fi
8   else if $mess$ is answer then
9       $cEntry$ = get-cache-entry($mess.MessageID$);
10      $values, totalResult$ = merge-pops($cEntry$, $mess$);
11      update-cache-entry($totalResult$);
12      if not initiator then send-answer($values$, $mess$, $cEntry.addresser$); fi
13  fi
```

Figure 2. Procedure *process-IMS-msg*

Due to the lack of space we only list the algorithm that is used for processing messages applying the IMS strategy in Figure 2. As a basic principle processing query messages takes place in four main steps, which are: 1. query forwarding (line 2), 2. local query processing (line 3), 3. updating the cache for intermediate results (line 4), and 4. sending an answer message (lines 5-7) to that peer that has sent the query message in the first place. The received query message is forwarded to all those neighboring peers that, according to the routing filters, can contribute to the final result, which reflects a simple cost-based routing choice. Due to the lack of space we omit details about how the information is applied exactly, and rather focus on the information maintenance. More detailed cost models may be applied here. In contrast to non-incremental strategies the answer is sent immediately without waiting for any other peer if the second step led to any results. If no new data was found when processing the query locally no answer has to be sent.

The second part of figure 2 shows the processing of answer messages. The main difference to non-incremental approaches consists in the merging process, but resulting problems (e.g., incremental processing of joins) are not focus of this work.

As we will see in Section 5 IMS is nearly as scalable as QS and provides the expected robustness. Thus, it is predestinated for processing queries in P2P systems with high frequencies of failure.

4 Adaptive Routing Filter Maintenance

In this section we introduce how to obtain and maintain the knowledge contained in the routing filters. For more details we refer to the extended version of this work. We utilize a dynamic and adaptive maintenance approach based on query feedback, where the results of executed queries are used in order to refine statistics and to detect changes in the network. Reducing the number of update messages increases scalability and robustness. Moreover, the routing filters adapt to the actual workload and changing network situations. In the following we concentrate on recording peer fluctuations resulting from frequent peer joins and leaves, crashes, respectively. The presented method also applies to updates in a peer's data itself and, with slight modifications, to the building of the filters.

Query feedback approaches were discussed for relational systems before ([9]) as well as for distributed systems ([7]). One main challenge of query feedback in combination with IMS is that we get a separate answer from every peer that can provide result elements. In a trivial approach we would just update the contained values according to the results as they come in. However, this would result in strong oscillations in the routing filters, when many results with few tuples arrive. To avoid this effect we aggregate the incoming results and feed them back into the filters in multiple batches at pre-configured times using learning rates. In order to do so, we make use of the histogram structure described in Section 3.2. This bears another advantage: by limiting the maximal batch timeout we can adapt the filters to really occurring answer times, rather than a more abstract hop count. In every batch we guarantee, that only the current time slot is affected in order to avoid duplicated learning of "older" slots as well as a preemptive adaption of times lots that *could not* have arrived yet.

A difficulty arises when a processed query includes constraints. If there is an error in the estimation of the difference between expected results and the actual results, we have to distribute the error either to the constraints or to the total count of tuples in the routing filter.

```
Input:  RootElement of QF-filter $qf$, RootElement of filter $idx$
1  /* Estimate selectivities */
2  $selectivity$ = 1;
3  forall $constraintqueries$ do
4      $idxCount$ = 0;
5      forall elements $e$ that match $constraintquery$ in $idx$ do
6          get Histogram $hist$ of $e$;
7          $idxCount$ += $hist$.totalDocs;
8      od
9      $localselectivity = \frac{constraintquery.count}{selectivity*idxCount}$;
10     $selectivity$ *= $localselectivity$
11 od
12 Vector $matches$ = locate all elements in filter that match query;
13 feed-back($qf, idx, matches, selectivity$);
14 forall remaining $matches$ do
15     recursively reduce entries of histograms of matches in filter;
16 od
```

Figure 3. Procedure *execute-batch*

Imagine, for instance, the query `//objects[brightness>4]/coordinates` returns – if at all – only coordinate objects but no information about the *selectivity* of the constraint `[brightness>4]`. Without additional information we cannot decide, whether an error is due to a wrong selectivity estimation (between 0..1) or the total count of coordinate objects. Therefore we execute an additional query ($constraintquery$) which returns just the number of constraint elements ($constraintquery.count$). In the example this is the count of all `//objects/brightness` objects that match the constraint `[brightness>4]`. This query is initiated at a peer together with the main query, so we do not need extra messages. By adding this small overhead we can now compute the actual selectivity by the formula $\frac{constraintquery.count}{idx\_of\_coordinates.totalCount}$. So we can update our counts for the coordinate objects by the formula $\frac{\#coordinate-objects}{selectivity}$. Additionally, in a further step we can adapt the data distribution of our constraint histograms.

When processing a batch we have to take the following cases into account:

1. The feedback returns an element that is not yet in the filter, so we have to extend the filter.
2. The feedback returns no match for a specific subelement in the filter, then we have to reduce the count (that is all buckets in the histogram in the corresponding time slot).
3. The feedback returns matches for an element, but the counts differ. Then we have to adapt the filter with a learning rate.

```
Input:  RootElement of QF-filter $qf$, RootElement of filter $idx$, Vector $matches$,
        double $selectivity$
1  if $qf == idx$ then
2      $error$ = ($qf.count$/$selectivity$) - $idx.count$;
3      $idx.count$ = $idx.count$ + $error$ * $learningRate$;
4      if $idx$ in $matches$ then remove $idx$ from matches; fi
5  fi
6  forall children $idxChild$ of $idx$ do
7      $currChild$ = $qf$.get-child($idxChild$);
8      if $currChild$ == null then recursively reduce the count of $idxChild$;
9      else feed-back($currChild, idxChild, matches, selectivity$);
10     fi
11 od
12 forall children $newChild$ of $qf$, not used before do
13     $idx$.add($newChild$);
14 od
```

Figure 4. Procedure *feed-back*

4. The query feedback returns nothing at all, then we have to adapt all elements contained in the filter that match the overall query as in case 2.

Figure 3 and Figure 4 depict the developed algorithms that process query feedback. QF-Filter qf is build from the information of a received answer, the filters are in the introduced XML-like structure. Seemingly, case 4 is worked on in the last part of Figure 3 as all matches to the query not returned by the query feedback are downgraded, that is the count is reduced. In Figure 4 lines 1–5 handle case 2, while lines 12–14 cover case 1. Case 2 is treated in between.

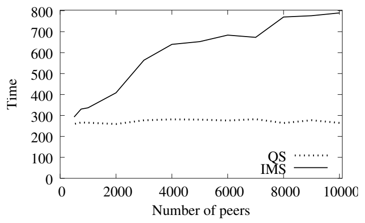
5 Evaluation

P2P Simulator We implemented a simulation environment using Java and Threads based on a central clock mechanism that meets the concerns of P2P systems in reality. Several XML files provide the ability of setting configuration parameters to ensure repeatability, dynamic, logging and control, as well as performance. This also includes characterizing the dynamic of P2P networks by setting peers join and crash frequency. In the current state we implemented the two query processing strategies QS and IMS and the introduced adaptive routing filters.

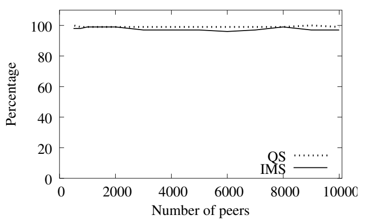
Incremental Strategy We ran multiple experiments to evaluate the impact of IMS in contrast to QS. As this is not focus of the presented short paper, we only take a very brief look on the results and concentrate on evaluating the query feedback approach in the next subsection.

Among other values we measured the time needed for answering all queries of our query mix, the number of generated messages and the sizes of the received results. In general we did multiple runs with identical configurations and figured the average values. We used a query mix of 25 queries (mainly selections, but also joins and unions, on TPC-H test data in XML format[2]), one query initiated at a randomly chosen peer in one time step with 1-2 time steps (randomly chosen) between the initiations. We used a hop count of 5 for limiting the routing filters.

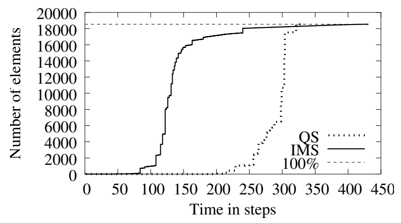
[2]http://www.cs.washington.edu/research/xmldatasets



(a) Average time per query

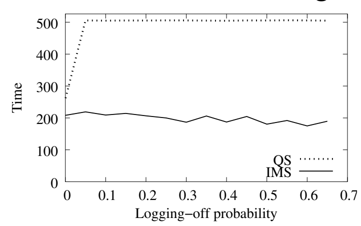


(b) Percentage of result data

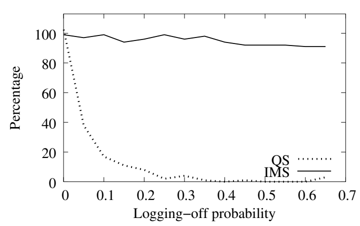


(c) Online query processing

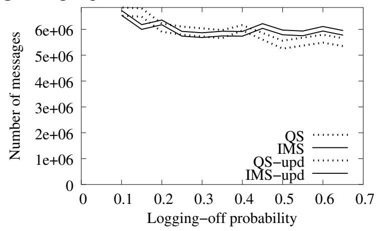
Figure 5. Measured values in a static environment



(a) Average time per query



(b) Percentage of result data



(c) Overhead of update processing

Figure 6. Measured values in a dynamic environment

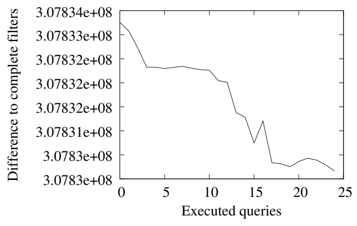
In our first tests we show that both strategies are scalable in static networks. Figure 5 illustrates the resulting answer times, received percentage of the complete result and the support of online queries. As we expected the QS approach is more efficient in static environments.

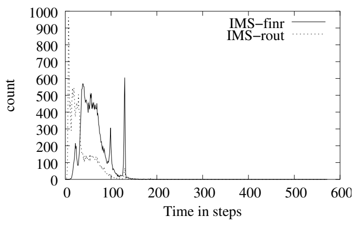
In real life applications we expect dynamic environments, which means networks with high frequencies of joins and leaves (crashes, respectively) among the peers. For the next tests we varied the probability of leaving and joining peers using a network size of 2000 peers exemplary. A logging off probability of x means that in each time step there is a chance of x that one peer crashes (as long as a minimal network size is given), the logging on probability we kept constant (in Figures 6(a) and 6(b) it is 0.0, so only peer failures happen; in Figures 6(c) it is 0.1). The incremental character of IMS, which minimizes waiting states during the processing of queries, solves the problems we are facing using QS while supporting peer leaves. The result sizes using QS are extremely small and the answer times are constantly high in contrast to IMS. Even a relatively small logging on probability leads to an unmanageable message explosion if we use a simple broadcast mechanism based on update messages ('upd' in 6(c)), thus we learned a clear lesson: in highly dynamic environments we strongly need adaptive approaches of maintaining cost factors.

Summarizing our first experiments, we appoint the following: IMS is scalable as QS in static networks but much more efficient and robust in dynamic networks. IMS should be chosen when best-effort approaches, as usual in dynamic P2P systems, are claimed – and they are additionally supported by its capability of online query processing.

Routing Filter Maintenance Now we take a look at the evaluation of the query feedback approach. In our first experiment we initialized a static network of 2000 peers with filters initialized empty and compared their content against statically initialized filters after $1, 2, \ldots, 23$ queries of the used query mix, each query initiated at a randomly chosen peer. Figure 7(a) shows the decreasing difference of the knowledge contained in the routing filters as more and more feedback becomes available in a static environment. The absolute values stem from a simple method of comparing routing filters, interesting is the trend in the difference. As we can see, there is even a point where some "negative" learning is applied (query 16). Intuitively, a query feedback approach can only work if we process queries with containment relations. Figures 7(b) and 7(c) are generated in a dynamic environment. They illustrate the impact of the chosen feedback timeout ('finr/rout': #floodings/routings). Choosing it too low, the number of floodings (indicating that no knowledge is available in the filters) even increases with the usage of query feedback. Using a larger timeout (Figure 7(c)), the number of floodings also increases, but it shows that our filter structures are aware of the changes in the network and its old knowledge is outdated and to be renewed with a stronger explorational behavior.

In a second series of experiments we compared the number of neighbors queried against the number of available neighbors in a small static network with empty starting filters over time. Figure 8 shows the actual number of messages in comparison to the number of messages, that flooding would have generated, and the difference between those values. In Figure 8(a) queries are initiated at different *Q*uery *I*nitiating *P*eers (QIP), 8(b) and 8(c) illustrate one query executed on random peers and on one peer, respectively. As expected, the bandwidth reduction increases when the same query is executed, as each query can profit from the feedback of the earlier ones. Whether a query is executed on the same peer or on different ones is only of minor importance,


(a) Adaption of filters


(b) Low QF-timeout

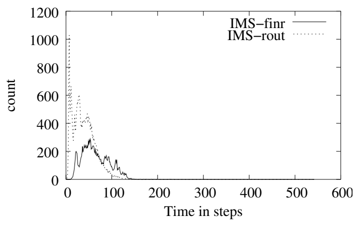
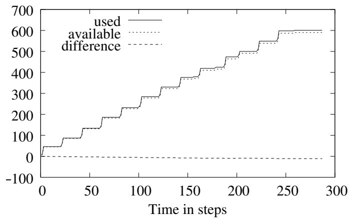
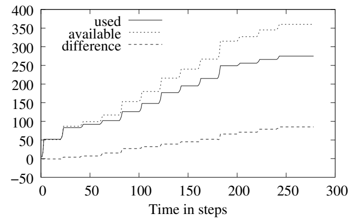

(c) High QF-timeout

Figure 7. Potential of adaption and comparison of flooded and routed query forwarding


(a) Random QIP, random query


(b) Random QIP, fixed query

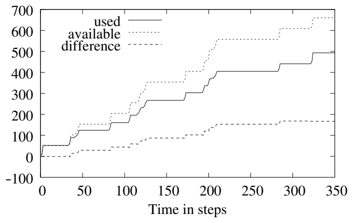

(c) Fixed QIP, fixed query

Figure 8. Bandwidth utilization

as all peers on the query path adapt their routing filters.

In the presented experiments we could reduce the number of sent messages and we expect greater effects in larger networks and on queries with higher selectivities. We expect our approach to be especially useful in large networks with high dynamics, where the filters cannot be initialized efficiently in a static way. As expected the query feedback only improves performance if the queries correlate semantically. A bundle of problems still remains to be solved, for instance finding sensible feedback timeouts. Another point is the number of queries needed to adapt a useful percentage of all routing filters. A possible solution is to include special maintenance queries that are sent out randomly in times of low network load. Last but not least we have to investigate the problem of "forgetting" redundant paths, which is fine for static networks, but can lead to critical problems in dynamic environments.

6 Summary

Efficient query processing is an essential requirement when implementing PDMS applications, as it is in any other distributed system. Main challenges are the mere size of the underlying networks and, due to the characteristics of P2P systems, the challenge of highly dynamic networks. In this work we introduced an iterative, almost stateless, query execution strategy in order to deal with these two challenges. In order to overcome the burden of optimizing processing while having only limited knowledge we use an approach based on extended local peer indexes, called routing filters. A strategy based on query feedback used to maintain these filters dynamically, adapted to the query and workload, proved to be suitable. Moreover, it promises more impact fine-tuned and in conjunction with dynamic cost-based execution of queries. Therefore, we plan to implement adaptive query processing techniques as well. We showed the usefulness of our algorithms to improve scalability, robustness as well as self-organization in P2P systems. In conjunction with a detailed evaluation we outlined some important aspects of future research.

References

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB Journal*, pages 591–600, 2001.

[2] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *22nd Int. Conf. on Distributed Computing Systems*, pages 23–32, July 2002.

[3] L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating Data Sources in Large Distributed Systems. In *VLDB'03*, pages 874–885, 2003.

[4] M. Karnstedt, K. Hose, and K.-U. Sattler. Query Routing and Processing in Schema-Based P2P Systems. In *DEXA'04 Workshops*, pages 544–548. IEEE Computer Society, 2004.

[5] G. Koloniari, Y. Petrakis, and E. Pitoura. Content-based overlay networks for xml peers based on multi-level bloom filters. In *DBISP2P*, pages 232–247, 2003.

[6] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[7] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Parr. Xpathlearner: an on-line selftuning markov histogram for xml path selectivity estimation. In *VLDB'02*, 2002.

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

[9] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB'01*, pages 19–28, 2001.

[10] I. Tatarinov and A. Halevy. Efficient query reformulation in peer data management systems. In *SIGMOD'04*, pages 539–550, 2004.