Technical Report

TR-2010-005

Dynamic Query Processing for P2P Data Services in the Cloud

by

Pawel Jurczyk, Li Xiong

Mathematics and Computer Science

EMORY UNIVERSITY

Dynamic Query Processing for P2P Data Services in the Cloud

Pawel Jurczyk and Li Xiong

Abstract—With the trend of cloud computing, data and computing are moved away from desktop and are instead provided *as a service* from the cloud. Data-as-a-service enables access to a wealth of data across distributed and heterogeneous data sources in the cloud. We designed and developed DObjects, a general-purpose P2P-based query and data operations infrastructure that can be deployed in the cloud. This paper presents the details of the dynamic query execution engine within our data query infrastructure that dynamically adapts to network and node conditions. The query processing is capable of fully benefiting from all the distributed resources to minimize the query response time and maximize system throughput. We present a set of experiments using both simulations and real implementation and deployment.

Index Terms—Cloud computing, data-as-a-service, data access in the cloud, distributed query optimization.

1 Introduction

With the trend of cloud computing[1][2], data and computing are moved away from desktop and are instead provided *as a service* from the cloud. Current major components under the cloud computing paradigm include infrastructure-as-a-service (such as EC2 by Amazon), platform-as-a-service (such as Google App Engine), and application or software-as-a-service (such as GMail by Google). There is also an increasing need to provide data-as-a-service [1] with a goal of facilitating access to a wealth of data across distributed and heterogeneous data sources available in the cloud.

Consider a system that integrates the air and rail transportation networks with demographic databases and patient databases in order to model the large scale spread of infectious diseases (such as the SARS epidemic or pandemic influenza). Rail and air transportation databases are distributed among hundreds of local servers, demographic information is provided by a few global database servers and patient data is provided by groups of cooperating hospitals.

While the scenario above demonstrates the increasing needs for integrating and querying data across distributed and autonomous data sources, it still remains a challenge to ensure interoperability and scalability for such data services. To achieve interoperability and scalability, data federation is increasingly becoming a preferred data integration solution. In contrast to a centralized data warehouse approach, a data federation combines data from distributed data sources into one single *virtual* data source, or a data service, which can then be accessed, managed and viewed as if it was part of a single system.

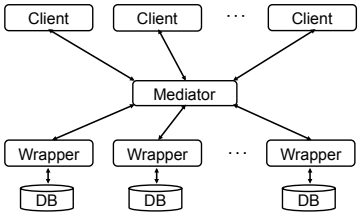


Fig. 1. Typical Mediator-Based Architecture

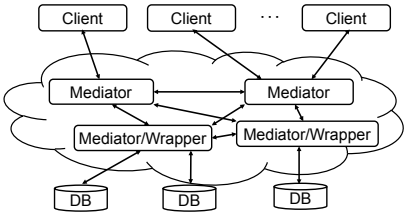


Fig. 2. P2P-Based Architecture

Many traditional data federation systems employ a centralized mediator-based architecture (Figure 1). We recently proposed DObjects [2], [3], a P2P-based architecture (Figure 2) for data federation services. Each system node can take the role of either a mediator or a mediator and wrapper at the same time. The nodes form a virtual system in a P2P fashion. The framework is capable of extending cloud computing systems with data operations infrastructure, exploiting at the same time distributed resources in the cloud.

Contributions. In this paper we present DObjects system and its query processing component containing novel dynamic query processing engine. We present

This work was presented in part in DEXA 2009 Conference, Linz, Austria, 31 August - 4 September

- *Pawel Jurczyk and Li Xiong are with the Department of Mathematics and Computer Science, Emory University, Atlanta, GA, 30322. E-mail: {pjurczy,lxiong}@emory.edu*

1. http://en.wikipedia.org/wiki/Cloud_computing
2. http://www.theregister.co.uk/2009/01/06/year_ahead_clouds/

our dynamic distributed *query execution and optimization* scheme.

DObjects builds on top of a *distributed* mediator-wrapper architecture where individual system nodes serve as mediators (mediating queries across data sources) and/or wrappers (retrieving data from individual data sources). They interact with each other in a P2P fashion and form a *virtual* system to provide a seamless and transparent data federation service. As an analogy, our system nodes can be considered as *droplets*, small elements that provide similar functionality in the cloud. An element can be a single physical machine or a service provided by a physical machine (in that case physical machine can function as several droplets). Just as thousands or millions of droplets form a single drop in nature, in cloud computing, groups of *droplets* that provide similar functionality can form a *micro-cloud*. *Micro-clouds* are an integral part of the whole cloud computing system and can provide specific services to users. In spirit, our data federation service which we propose here can be considered as such *micro-cloud*.

In addition to leveraging traditional distributed query optimization techniques, query processing and optimization in DObjects is focused on dynamically placing (sub)queries on the system nodes (mediators) to minimize the query response time and maximize system throughput. In our query execution engine, (sub)queries are deployed and executed on system nodes in a dynamic (based on nodes' on-going knowledge of the data sources, network and node conditions) and iterative (right before the execution of each query operator) manner. Such an approach guarantees the best reaction to network and resource dynamics. We experimentally evaluate our approach using both simulations and real deployment.

Compared to our preliminary work reported in [2], [4] and demonstrated in [3], this paper brings significant extensions to the query processing engine. First, we extended the query migration with a probabilistic workload distribution. Instead of migrating workload to the best candidate, it can be migrated to a random node from a set of candidates. Second, we extended the framework with dynamic estimation of parameters used in query migration model. The parameters are estimated based on knowledge of each node about latency to other nodes and load of other nodes. We also extended experimental evaluation section and provide a set of new experiments that validate the improvements we have added and evaluate a large-scale deployment of the real system implementation using PlanetLab testing environment.

2 Related work

Our work on DObjects and its query processing schemes was inspired and informed by a number of research areas. We provide a brief overview of the relevant areas in this section.

Distributed Databases and Distributed Query Processing. Distributed databases have been a subject of research for quite a long time [5], [6]. While sharing a modest target for scalability, earlier distributed database research and prototypes presented various distributed query processing techniques such as different join algorithms or alternative ways of shipping data from one site to the other and different architectures such as peer-to-peer, client-server and multitier architecture. DObjects adopts some of these "textbook" distributed query processing techniques such as semi-join.

Later distributed database or middleware systems, such as Garlic [7], DISCO [8] or TSIMMIS [9], target large-scale heterogeneous data sources. Many of them employ a *centralized* mediator-wrapper based architecture to address the database heterogeneity in the sense that a single mediator server integrates distributed data sources through wrappers. The query optimization focuses on integrating wrapper statistics with traditional cost-based query optimization for single queries spanning multiple data sources. As the query load increases, the centralized mediator may become a bottleneck.

A number of distributed query systems were targeted at Internet-scale in recent years. HyperQueries framework [10] is based on an idea of electronic market that serves as an intermediary between clients and providers executing their sub-queries referenced via hyperlinks. Similar approach is introduced in Active XML[3]. Instead of data objects, response to user queries is XML which has references (active links) to Web services providing given information. PIER [11], [12] is one of the first general-purpose relational query processors built on top of a distributed hash table (DHT) structured overlay for massively distributed networks. In many ways, PIER's architecture and algorithms are closer to parallel database systems particularly in the use of hash-partitioning during query processing. The initial work in Seaweed [13] targets at ad-hoc query processing for distributed end-systems and their main focus is on dealing with end-system unavailability. Most of these solutions target at geographic scalability. In addition, they provide interfaces for querying data and limit other operations (such as deletions, updates or creation).

The main issue in Internet-scale systems described above is how to efficiently route the query to data sources, rather than on integrating data from multiple data sources. As a result, the query processing in such systems is focused on efficient query routing schemes for network scalability.

Another related body of work such as Piazza [14] and PeerDB [15] are focused on the semantic challenge of integrating many peer databases with heterogeneous schemas. There are also recent works focusing on specific components of query processing in Internet-scale data networks such as cardinality estimation [16]. A number of works are focused on distributed query evaluation

3. http://www.activexml.net

on semi-structured data or XML using techniques such as query decomposition [17] and partial evaluation [18]. These research agendas complement our research.

While we focus on system and structural heterogeneity as the mediator-based systems, an important and related challenge we do not address is the semantic heterogeneity of data sources. We refer to a survey of the issues [19] and a few recent proposals [20], [21] focusing on schema mediation in distributed systems.

The recent software frameworks, such as map-reduce-merge [22] and Hadoop[4], support distributed computing on large data sets on clusters of computers and can be used to enable cloud computing services. The focus of these solutions, however, is on data and processing distribution rather than on data integration.

It is important to position DObjects among the existing distributed system frameworks and distributed database systems. A number of distributed systems technologies and mechanisms such as DCOM[5], CORBA[6], and RMI[7] support distributed objects paradigm by allowing objects to be distributed across a heterogeneous network and can be used to build distributed applications. Our system builds on top of above technologies and offers a general platform for metacomputing with support for distributed data integration and data operation services. In particular, current implementation of DObjects builds on top of a resource sharing platform H2O [23] that builds on top of RMIX (an extension of RMI). The data services provided by DObjects offer query language and query execution optimization substrate that is fully integrated with the metacomputing middleware and can be used easily and transparently in distributed applications.

While it is not the aim of DObjects to be superior to these works, our system distinguishes itself by addressing an important problem space that has been overlooked, namely, integrating large-scale heterogeneous data sources with both network and query load scalability without sacrificing query complexities and transaction semantics. In spirit, DObjects is a *distributed* P2P mediator-based system in which a federation of mediators and wrappers forms a virtual system in a P2P fashion (see Figure 2). Our optimization goal is focused on building effective sub-queries and optimally placing them on the system nodes (mediators) to minimize the query response time and maximize throughput.

The most relevant to our work are OGSA-DAI and its extension OGSA-DQP [24] introduced by a Grid community as a middleware assisting with access and integration of data from separate sources. While the above two approaches share a similar set of goals with DObjects, they were built on the grid/web service model. In contrast, DObjects is built on the P2P model and provides resource sharing on a peer-to-peer basis.

4. http://hadoop.apache.org/core/
5. http://msdn2.microsoft.com/en-us/library/ms809340.aspx
6. http://www.corba.org/
7. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

Data Streams and Continuous Queries. A large amount of efforts was contributed to the area of continuous or pervasive query processing [25], [26], [27], [18], [28], [29], [30]. A methods of sharing work in the context of distributed aggregation queries that vary in their selection predicates was analyzed in [31]. The query optimization engine in DObjects is most closely related to SBON [32]. SBON presented a stream based overlay network for optimizing queries by carefully placing aggregation operators. DObjects shares a similar set of goals as SBON in distributing query operators based on on-going knowledge of network conditions. SBON uses a two step approach, namely, virtual placement and physical mapping for query placement based on a cost space. In contrast, we use a single cost metric with different cost features for easy decision making at individual nodes for a local query migration and explicitly examine the relative importance of network latency and system load in the performance.

Load balancing. Past research on load balancing methods for distributed databases resulted in a number of methods for balancing storage load by managing the partitioning of the data [33], [34]. Mariposa [35] offered load balancing by providing marketplace rules where data providers use bidding mechanisms. Load balancing in a distributed stream processing was also studied in [36] where load shedding techniques for revealing overload of servers were developed.

3 DObjects Overview

In this section we briefly describe DObjects framework. For further details we refer readers to [2], [3].

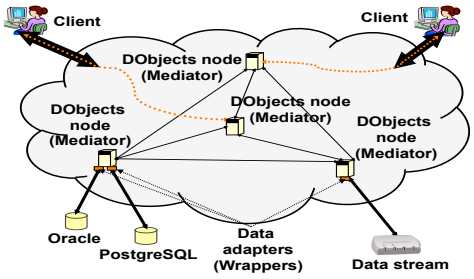


Fig. 3. System architecture.

A novel aspect of our architecture is that it is implemented on top of a *distributed resource sharing* framework. It consists of multiple decentralized system nodes which can serve as wrappers and/or mediators and form a *virtual system* in a P2P fashion. Figure 3 depicts DObjects framework deployed for the cloud. The system has no centralized services and uses the *metacomputing* paradigm as a resource sharing substrate to benefit from computational resources available in the cloud. Each node in the system can be considered a *droplet* as it provides similar functionality to other nodes and all the droplets form a *micro-cloud*. Each droplet can serve as a

```
1: generate high-level query plan tree
2: active element ← root of query plan tree
3: choose execution location for active element
4: if chosen location ≠ local node then
5:     delegate active element and its subtree to chosen location
6:     return
7: end if
8: execute active element;
9: for all child nodes of active element do
10:    go to step 2
11: end for
12: return result to parent element
```

Alg. 1. Local algorithm for query processing

mediator that provides its computational power for query mediation and results aggregation. Each droplet can also serve as a data adapter or *wrapper* that pulls data from data sources and transforms it to a uniform format that is expected while building query responses. Users can connect to any system node; however, while the physical connection is established between a client and one of the system nodes, the logical connection is established between a client node and a virtual system consisting of all available nodes, or the *micro-cloud*.

4 Query Execution and Optimization

In this section we focus on the query processing issues of DObjects, present an overview of the dynamic distributed query processing engine that adapts to network and resource dynamics, and discuss details of its cost-based query placement strategies.

4.1 Overview

As we have discussed, the key to query processing in our framework is to have a decentralized and distributed query execution engine that dynamically adapts to network and resource conditions. In addition to adapting "textbook" distributed query processing techniques such as distributed join algorithms and the learning curve approach for keeping statistics about data adapters, our query processing framework presents a number of innovative aspects. First, instead of generating a set of candidate plans, mapping them physically and choosing the best ones as in a conventional cost based query optimization, we create one initial abstract plan for a given query. The plan is a high-level description of relations between steps and operations that need to be performed in order to complete the query. Second, when the query plan is being executed, placement decisions and physical plan calculation are performed dynamically and iteratively. Such an approach guarantees the best reaction to changing load or latency conditions in the system.

It is important to highlight that our approach does not attempt to optimize physical query execution performed on local databases. Responsibility for this is pushed to data adapters and data sources. Our optimization goal is at a higher level focusing on building effective sub-queries and optimally placing those sub-queries on the system nodes to minimize the query response time.

```
select   c.name, r.destination,
         f.flightNumber, p.lastName
from     CityInformation c, c.IRails r, c.IFlights f,
         f.IPassengers p
where    c.name like „San%" and p.lastName=„Adams"
```

Fig. 4. Query example

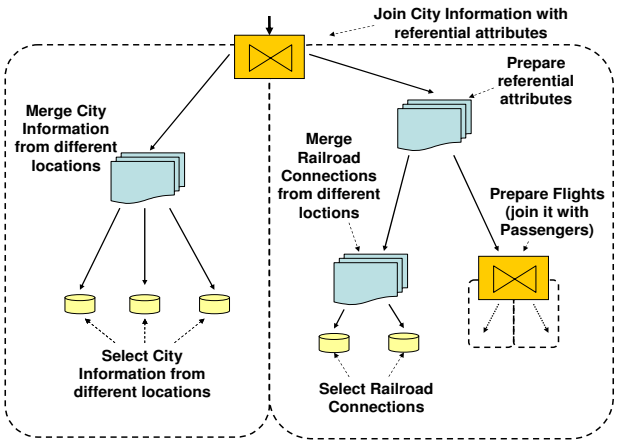


Fig. 5. Example of high-level query plan.

Our query execution and optimization consists of a few main steps. First, when a user submits a query, a high-level query description is generated by the node that receives it. An example of such a query plan is presented in Figure 5. The plan corresponds to the query introduced in Figure 4 that queries for cities along with related referential attributes: railroad connections and flights. In addition, each flight will provide a list of passengers. Note that each type is provided by a different physical database. The query plan contains such elements as *joins*, *horizontal* and *vertical data merges*, and *select* operations that are performed on data adapters. Each element in the query plan has different algorithms of *optimization* (see Section 4.2).

Next, the node chooses active elements from the query plan one by one in a top-down manner for execution. Execution of an active element, however, can be delegated to any node in the system in order to achieve load scalability. If the system finds that the best candidate for executing current element is a remote node, the *migration of workload* occurs. In order to choose the best node for the execution, we deploy a network and resource-aware cost model that dynamically adapts to network conditions (such as delays in interconnection network) and resource conditions (such as load of nodes) (see Section 4.3). If the active element is delegated to a remote node, that node has a full control over the execution

of any child steps. The process works recursively and iteratively, therefore the remote node could decide to move child nodes of submitted query plan element to other nodes or execute it locally in order to use the resources in the most efficient way to achieve good scalability. Algorithm 1 presents a sketch of the local query execution process. Note that our algorithm takes a greedy approach without guaranteeing the global optimality of the query placement. In other words, each node makes a local decision on where to migrate the (sub)queries.

4.2 Execution and Optimization of Operators

In previous section we have introduced the main elements in the high-level query plan. Each of the elements has different goals in the optimization process. It is important to note that the optimization for each element in the query plan is performed iteratively, just before given element is executed. We describe the optimization strategies for each type of operators below.

Join. Join operator is created when user issues a query that needs to join data across sites. In this case, join between main objects and the referenced objects have to be performed (e.g., join flights with passengers). The optimization is focused on finding the most appropriate join algorithm and the order of branch executions. The available join algorithms are nested-loop join (NLJ), semi-join (SJ) and bloom-join (BJ) [6]. In case of NLJ, the branches can be executed in parallel to speedup the execution. In case of SJ or BJ algorithms, the branches have to be executed in a pipeline fashion and the order of execution has to be fixed. Our current implementation uses a semi-join algorithm and standard techniques for result size estimations. There is also a lot of potential benefits in parallelization of the join operator execution using such frameworks as map-reduce-merge [22]. We leave this to our future research agenda.

Data merge. Data merge operator is created when data objects are split among multiple nodes (horizontal data split) or when attributes of an object are located on multiple nodes (vertical data split). Since the goal of the data merge operation is to merge data from multiple input streams, it needs to execute its child operations before it is finished. Our optimization approach for this operator tries to maximize the parallelization of sub-branch execution. This goal is achieved by executing each sub-query in parallel, possibly on different nodes if such an approach is better according to our cost model that we will discuss later.

Select. Select operator is always the leaf in our high-level query plan. Therefore, it does not have any dependent operations that need to be executed before it finishes. Moreover, this operation has to be executed on locations that provide queried data. The optimization issues are focused on optimizing queries submitted to data adapters for a faster response time. For instance, enforcing an order (sort) to queries allows us to use merge-joins in later operations. Next, *response chunks* are built in order to support queries returning large results. Specifically, in case of heavy queries, we implement an iterative process of providing smaller pieces of the final response. In addition to helping to maintain a healthy node load level in terms of memory consumption, such a feature is especially useful when building a user interface that needs to accommodate a long query execution.

4.3 Query Migration

The key of our query processing is a greedy local query migration component for nodes to delegate (sub)queries to a remote node in a dynamic (based on current network and resource conditions) and iterative (just before the execution of each element in the query plan) manner. In order to determine the best (remote) node for possible (sub)query migration and execution, we first need a cost metric for the query execution at different nodes. Suppose a node migrates a query element and associated data to another node, the cost includes: 1) a transmission delay and communication cost between nodes, and 2) a query processing or computation cost at the remote node. Intuitively, we want to delegate the query element to a node that is "closest" to the current node and has the most computational resources or least load in order to minimize the query response time and maximize system throughput. We introduce a cost metric that incorporates such two costs taking into account current network and resource conditions. Formally Equation 1 defines the cost, denoted as $c_{i,j}$, associated with migrating a query element from node i to a remote node j:

$$c_{i,j} = \alpha * (DS/bandwidth_{i,j} + latency_{i,j}) + (1-\alpha) * load_j \quad (1)$$

where DS is the size of the necessary data to be migrated (estimated using statistics from data sources), $bandwidth_{i,j}$ and $latency_{i,j}$ are the network bandwidth and latency between nodes i and j, $load_j$ is the current (or most recent) load value of node j, and α is a weighting factor between the communication cost and the computation cost. Both cost terms are normalized values between 0 and 1 considering the potential wide variances between them.

Basic migration model. To perform query migration, each node in the system maintains a list of candidate nodes that can be used for migrating queries. For each of the nodes, it calculates the cost of migration and compares the minimum with the cost of local execution. If the minimum cost of migration is smaller than the cost of local execution, the query element and its subtree are moved to the best candidate. Otherwise, the execution will be performed at the current node. To prevent a (sub)query being migrated back and forth between nodes, we require each node to execute at least one operator from the migrated query plan before further migration. Alternatively, a counter, or Time-To-Live (TTL) strategy, can be implemented to limit the number of

migrations for the same (sub)query. TTL counter can be decreased every time a given (sub)tree is moved, and, if it reaches 0, the node has to execute at least one operator before further migration. The decision of a migration is made if the following equation is true:

$$min_j\{c_{i,j}\} < \beta * (1-\alpha) load_i \quad (2)$$

where $min_j\{c_{i,j}\}$ is the minimum cost of migration for all the nodes in the node's candidate list, β is a tolerance parameter typically set to be a value close to 1 (e.g. we set it to 0.98 in our implementations). Note that the cost of a local execution only considers the load of the current node.

Probabilistic migration model. The basic migration model as described above poses a risk of overloading one node when large number of queries is submitted to given node in a short period of time. In this case, if the decision about migration all the queries would be made, each query could be migrated to the same remote node. To minimize the risk of such a phenomenon, we have implemented a probabilistic workload distribution. In our model, not only the best remote candidate is chosen, but a set of all the nodes that have smaller processing cost than the local node. We will assume that set S contains all the possible candidates for query migration:

$$S = \{c_{i,j} : c_{i,j} < \beta * (1-\alpha) * load_i\} \quad (3)$$

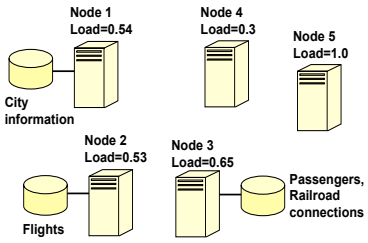
When the decision on migration is made, one of the nodes is chosen randomly from set S, and the subquery is migrated to that node. The nodes are chosen with inverse probability to the cost of migration what guarantees that the least loaded node is chosen the most often. Specifically, the probability of choosing remote node j from set S is calculated as follows:

$$p(j) = \frac{\frac{1}{c_{i,j}}}{\sum \frac{1}{c_{i,j}}} \quad (4)$$

Estimation of optimal α value. An important issue in our workload migration model is the α parameter and its proper value. Although our experimental evaluation contains indication of its optimal value for our deployment, one can expect that this value will vary from one deployment to another. For instance, in systems where differences between latency are small, the alpha value should favor load information. On the other hand, if latency differences are significant, the load information should have less impact.

The optimal value of α parameter needs to be estimated based on the knowledge of the system by each node independently. Our framework for predicting this value works as follows. Once started, each node is in the initial phase where there is a little or no information about load of other nodes or latency to other nodes, and the default α value is used. The default α value is currently set to 0.3 (this value was found to be the most efficient in our initial experiments; see the experimental evaluation section). As the nodes are present in the system, they learn more and more facts about the systems. As the situation about load of other nodes and distance to other nodes is propagated among nodes, they switch from the initial phase of alpha estimation to the final phase. In the final phase each node calculates its local α by analyzing the load of other nodes and latency to other nodes.

We already mentioned that in a system with a uniform latency, load information should have bigger impact. On the other hand, in a system with a uniform load, latency should have bigger impact. Based on this premise, our system for predicting the optimal alpha value looks at standard deviation of latency and load, and tries to find a correlation between those two factors and optimal α value. The two standard deviations can be calculated based on node's knowledge of the current system conditions. Once those values are known, the system can predict optimal α values using an experimentally defined function that correlates values of standard deviation of load and latency with the alpha value. We will show how to find such a function in the experimental evaluation section where we use a series of experiments in simulated environment and show how to use a non-linear least squares fitting to find the relation between latency, load and α.



	1	2	3	4	5
1	0	10	30	30	50
2	10	0	20	40	50
3	30	20	0	40	30
4	30	40	40	0	20
5	50	50	30	20	0

Latency between nodes (ms)

Fig. 6. Setup for Optimization Illustration

Illustration. To illustrate our query optimization algorithm, let us consider a query from Figure 4 with a sample system deployment as presented in Figure 6. Let us assume that a client submits his query to Node 5 which then generates a high-level query plan as presented in Figure 5. Then, the node starts a query execution. The operator at the root of the query plan tree is join. Using the equation 1 the active node estimates the cost for migrating the join operator. Our calculations will neglect the cost of data shipping for simplicity and will use $\alpha = 0.3$ and $\beta = 1.0$. The cost for migrating the query from Node 5 to Node 1 is: $c_{5,1} = 0.3 * (50/50) + (1-0.3) * 0.54 = 0.68$. Remaining migration costs are $c_{5,2} = 0.671$, $c_{5,3} = 0.635$ and $c_{5,4} = 0.33$. Using the equation 2 Node 5 decides to move the query to Node 4 ($c_{5,4} < 1.0 * (1-0.3) * 1.0$). After the migration, Node 4 will start execution of join operator at the top of the query plan tree. Let us assume that the node decides to execute the left branch first. CityInformation is provided by only one node, Node 1, and no data merge is required. Once the select operation

is finished on Node 1, the right branch of join operation can be invoked. Note that Node 4 will not migrate any of the sub-operators (besides selections) as the cost of any migration exceeds the cost of local execution (the cost of migrations: $c_{4,1} = 0.558$, $c_{4,2} = 0.611$, $c_{4,3} = 0.695$ and $c_{4,5} = 0.82$; the cost of local execution: 0.21).

4.4 Cost Metric Components

The above cost metric consists of two cost features, namely, the *communication latency* and the *load* of each node. We could also use other system features (e.g. memory availability), however, we believe the load information gives a good estimate of resource availability at the current stage of the system implementation. Below we present techniques for computing our cost features efficiently.

Latency between nodes. To compute the network latency between each pair of nodes efficiently, each DObjects node maintains a virtual coordinate, such that the Euclidean distance between two coordinates is an estimate for the communication latency. Storing virtual coordinates has the benefit of naturally capturing latencies in the network without a large measurement overhead. The overhead of maintaining a virtual coordinate is small because a node can calculate its coordinate after probing a small subset of nodes such as well-known landmark nodes or randomly chosen nodes. Several synthetic network coordinate schemes exist. We adopted a variation of Vivaldi algorithm [37] in DObjects. The algorithm uses a simulation of physical springs, where each spring is placed between any two nodes of the system. The rest length of each spring is set proportionally to current latency between nodes. The algorithm works iteratively. In every iteration, each node chooses a number of random nodes and sends a ping message to them and waits for a response. After the response is obtained, initiating node calculates the latency with remote nodes. As the latency changes, a new rest length of springs is determined. If it is shorter than before, the initiating node moves closer towards the remote node. Otherwise, it moves away. The algorithm always tends to find a stable state for the most recent spring configuration. An important feature about this algorithm is that it has great scalability which was proven by its implementation in some P2P solutions (e.g. in OpenDHT project [38]).

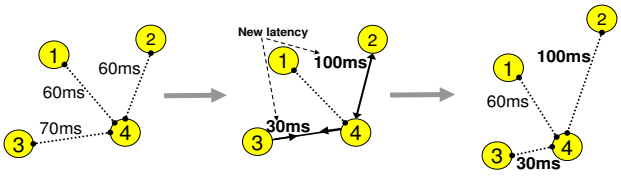


Fig. 7. Illustration of Virtual Coordinates Computation for Network Latency

Figure 7 presents an example iteration of the Vivaldi algorithm. The first graph on the left presents a current state of the system. New latency information is obtained in the middle graph and the rest length of springs is adjusted accordingly. As the answer to the new forces in the system, new coordinates are calculated. The new configuration is presented in the rightmost graph.

Load of nodes. The second feature of our cost metric is the *load* of the nodes. Given our desired goal to support *cross-platform* applications, instead of depending on any OS specific functionalities for the load information, we incorporated a solution that assures good results in a heterogeneous environment. The main idea is based on time measurement of execution of a predefined test program that considers computing and multithreading capabilities of machines [39]. The program we use specifically runs multiple threads. More than one thread assures that if a machine has multiple CPUs, the load will be measured correctly. Each thread performs a set of predefined computations including a series of integer as well as floating point operations. When each of the computing threads finishes, the time it took to accomplish operations is measured which indicates current computational capabilities of the tested node. In order to improve efficiency of our load computation method, we can dynamically adjust the interval between consecutive measurements. When a node has a stable behavior, we can increase this interval. On the other hand, if we observe rapid change in the number of queries that reside on a given node, we can trigger the measurement.

After the load information about a particular node is obtained, it can be propagated among other nodes. Our implementation builds on top of a distributed event framework, REVENTS[8], that is integrated with our platform for an efficient and effective asynchronous communication among the nodes.

5 Experimental Evaluation

Our framework is fully implemented with a current version available for download[9]. In this section we present an evaluation through simulations as well as a real deployment of the implementation.

5.1 Simulation Results

We ran our framework on a discrete event simulator that gives us an easy way to test the system against different settings. The configuration of data objects relates to the configuration mentioned in Section 4 and was identical for all the experiments below. The configuration of data sources for objects is as follows: object CityInformation was provided by node1 and node2, object Flight by node3 and node4, object RailroadConnection by node1 and finally object Passenger by node2. All nodes with

8. http://dcl.mathcs.emory.edu/revents/index.php
9. http://www.mathcs.emory.edu/Research/Area/datainfo/dobjects

TABLE 1
Experiment Setup Parameters. * - varying parameter

Test Case	Figure	# of Nodes (Mediators)	# of Clients	α
α vs. Query Workloads	8	6	14	*
α vs. # of Nodes	9	*	32	*
α vs. # of Clients	10	6	*	*
Comparison of Query Optimization Strategies	11	6	14	0.33
System Scalability	12, 13	20	*	0.33
Impact of Load of Nodes	14	*	256	0.33
Impact of Network Latencies	15	6	14	0.33
Workload distribution comparison	16	20	*	0.33
Alpha estimation	N/A	16, 32, 64	16, 32, 64, 128, 256	N/A

numbers greater than 4 were used as computing nodes. Load of a node affects the execution time of operators. The more operators were invoked on a given node in parallel, the longer the execution time was assumed. Different operators also had different impact on the load of nodes. For instance, a join operator had larger impact than merge operator. In order to evaluate the reaction of our system to dynamic network changes, the communication latency was assigned randomly at the beginning of simulation and changed a few times during the simulation so that the system had to adjust to new conditions in order to operate efficiently. The change was based on increasing or decreasing latency between each pair of nodes by a random factor not exceeding 30%. Table 2 gives a summary of system parameters (number of nodes and number of clients) and algorithmic parameter α with default values for different experiments.

Parameters Tuning - Optimal α. An important parameter in our cost metric (introduced in equation 1) is α that determines the relative impact of load and network latency in the query migration strategies. Our first experiment is an attempt to empirically find optimal α value for various cases: 1) different query workloads, 2) different number of nodes available in the system, and 3) different number of clients submitting queries.

For the first case, we tested three query workloads: 1) small queries for CityInformation objects without referential attributes (therefore, no join operation was required), 2) medium queries for CityInformation objects with two referential attributes (list of Flights and RailroadConnections), and 3) heavy queries with two referential attributes of CityInformation of which Flight also had a referential attribute. The second case varied the number of computational nodes and used the medium query submitted by 32 clients simultaneously. The last case varied a number of clients submitting medium queries.

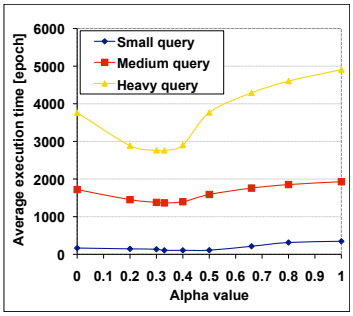


Fig. 8. Parameter Tuning - Query Workloads

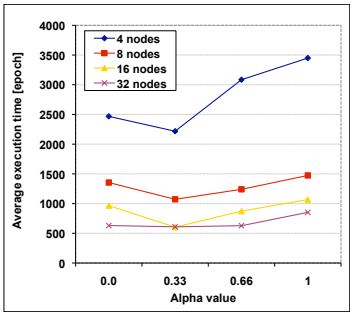


Fig. 9. Parameter Tuning - Number of Nodes

Figure 8, 9 and 10 report average execution times for different query loads, varying number of computational nodes, and varying number of clients respectively for different α. We observe that for all three test cases the best α value is located around the value 0.33. While not originally expected, it can be explained as follows. When more importance is assigned to the load, our algorithm will choose nodes with smaller load rather than nodes located closer. In this case, we are preventing overloading a group of close nodes as join execution requires considerable computation time. Also, for all cases, the response time was better when only load information was used ($\alpha = 0.0$) compared to when only distance information was used ($\alpha = 1.0$). For all further experiments we set the α value to be 0.33.

Comparison of Optimization Strategies. We compare a number of varied optimization strategies of our system with some baseline approaches. We give average query response time for the following cases: 1) no optimization (a naive query execution where children of current query operator are executed one by one from left to right), 2) statistical information only (a classical query optimization that uses statistics to determine the order of branch executions in join operations), 3) location information only ($\alpha = 1$), 4) load information only ($\alpha = 0$), and 5) full optimization ($\alpha = 0.33$).

The results are presented in Figure 11. They clearly show that, for all types of queries, the best response time corresponds to the case when full optimization is used. In addition, the load information only approach provides an improvement compared to the no optimization, statistical information only, and location information only

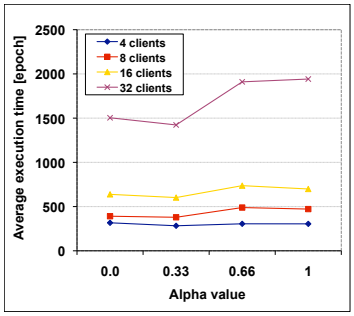


Fig. 10. Parameter Tuning - Number of Clients

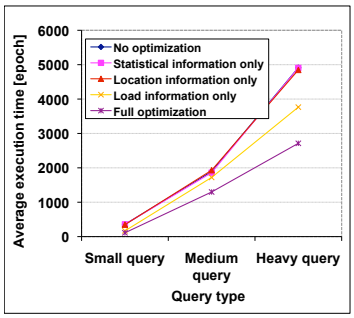


Fig. 11. Comparison of Different Query Optimization Strategies

approaches (the three lines overlap in the plot). The performance improvements are most manifested in the heavy query workload.

System Scalability. An important goal of our framework is to scale up the system for number of clients and load of queries. Our next experiment attempts to look at the throughput and average response time of the system when different number of clients issue queries. We again use three types of queries and a similar configuration to the above experiment.

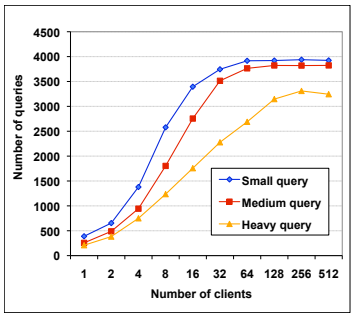


Fig. 12. System Scalability (Throughput) - Number of Clients

Figures 12 and 13 present the throughput and average response time for different number of clients respectively. Figure 12 shows the average number of queries that our system was capable of handling during a specified time frame for a given number of clients. As expected, the system throughput increases as the number of clients increases before it reaches its maximum.

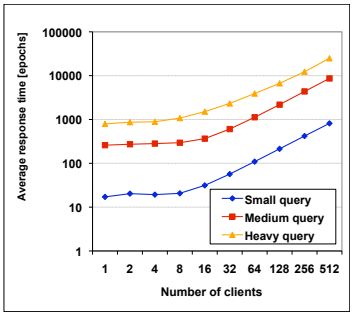


Fig. 13. System Scalability (Response Time) - Number of Clients

However, when the system reaches a saturation point (each node is heavily loaded), new clients cannot obtain any new resources. Thus, the throughput reaches its maximum (e.g., around 3950 queries per specified time frame for the case of small queries at 64 clients). Figure 13 reports the average response time and shows a linear scalability. Please note that the figure uses logarithmic scales for better clarity.

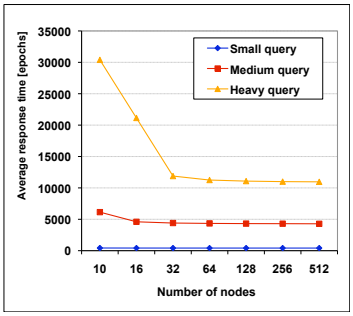


Fig. 14. Impact of Computational Resources

Impact of Available Computational Resources. In order to answer the question how the number of nodes available in the system affects its performance, we measured the average response time for varying number of available system nodes with 256 clients simultaneously querying the system. The results are provided in Figure 14. Our query processing effectively reduces the average response time when more nodes are available. For small queries, 10 nodes appears to be sufficient as an increase to 16 nodes does not improve the response time significantly. For medium size queries 16 nodes appears to be sufficient. Finally, for heavy queries we observed improvement when we used 32 nodes instead of 16. The behavior above is not surprising and quite intuitive. Small queries do not require high computational power as no join operation is performed. On the other hand, medium and heavy queries require larger computational power so they benefit from a larger number of available nodes.

Impact of Network Latency. Our last experiment was aimed to find the impact of a network latency on the performance. We report results for three network speeds:

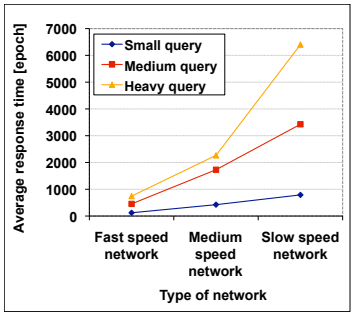


Fig. 15. Impact of Network Latency

a fast network that simulates a Fast Ethernet network offering speed of 100MBit/s, a medium network that can be compared to an Ethernet speed of 10MBit/s, and finally a slow network that represents speed of 1MBit/s.

The result is reported in Figure 15. The network speed, as expected, has a larger impact on the heavy query workload. The reason is that the amount of data that needs to be transferred for heavy queries is larger than medium and small queries, and therefore the time it takes to transfer this data in slower network will have much larger impact on the overall efficiency.

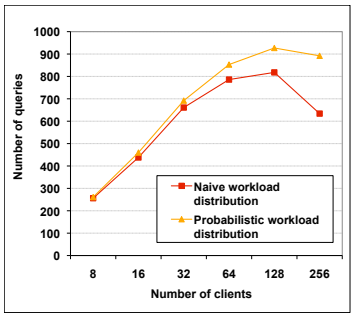


Fig. 16. Comparison of workload distribution algorithms

Workload distribution strategies. Our next experiment in the simulated environment had a goal of comparing the workload distribution strategies. We wanted to compare the two possible solutions. First, the probabilistic distribution that we implemented in our system where the best candidate is chosen from a set of candidate nodes based on the probability that is inversely proportional to the cost. Second, the naive distribution in which the workload is always migrated to the node with minimum migration cost. We tested those two approaches, and report the results in Figure 16. The plot presents an average number of executed large queries during the simulation for 20 system nodes and variable number of clients. The α value we used in this experiment was 0.33.

Again, the results confirm what we expected. The probabilistic distribution outperforms the naive one, and this phenomenon is more significant as the overall load of the system increases (the number of clients increases).

Estimating α based on load and latency. As we have already mentioned in section 4.3, the optimal α value will depend on particular system deployment characteristics. In this section we attempt to devise a technique for estimating optimal value of this parameter based on information about load and latency between nodes.

TABLE 2
Results of parameters fitting for used functions

Function	Estimated parameter	RSS
$\alpha = a * load + b * lat + c$	a=0.21 b=0.21 c=0.23	0.635
$\alpha = a * log(load) + b * lat + c$	a=0.05 b=0.21 c=0.38	0.589
$\alpha = a * load + b * log(lat) + c$	a=0.2 b=0.074 c=0.42	0.553
$\alpha = a * log(load) + b * log(lat) + c$	a=0.05 b=0.074 c=0.51	0.506

To collect the data we ran simulation of DObjects query processing for different configurations. The configuration for our simulations used 16, 32 and 64 DObjects nodes. The number of clients submitting queries varied, and we tested 16, 32, 64, 128 and 256 clients. Each possible combination of number of system nodes and clients was tested for 11 possible values of α parameter (values between 0.0 - 1.0, increasing by factor of 0.1). In total, we had 165 possible combinations of simulation configurations. For each of those combinations we further used three different configurations, each using different network characteristics (e.g., different configurations had different standard deviation of latency). This gave us a total number of possible configurations equal to 495.

Each possible simulation configuration was ran 10 times, and an average query response time was recorded. To obtain standard deviation of load and latency, during each of the simulations we recorded a latency and load information at few time stamps, and once the simulation was finished we calculated standard deviation of both factors. Additionally, for each combination of number of system nodes, number of clients and network latency, we selected an optimal alpha alpha value (e.g., the value that resulted in smallest query response time). Before proceeding with analysis, we normalized values for both standard deviations. Given such a data set, we were ready to analyze the correlation between standard deviation of load, latency and optimal α.

In the fitting process we used four possible functions. To find the correlation between our parameters we used R programming language and used the Nonlinear Least Squares fitting method to find the parameters of our fitting functions. The functions we tried in the fitting process were as follows: 1) $\alpha = a * load + b * lat + c$, 2) $\alpha = a*log(load)+b*lat+c$, 3) $\alpha = a*load+b*log(lat)+c$ and 4) $\alpha = a*log(load)+b*log(lat)+c$, where lat and $load$ are standard deviation of latency and load, respectively. The functions using logarithm were suggested by the initial plots of the optimal alpha values, where we plotted optimal alpha value for fixed latency deviation and variable load deviation and vice versa.

The results of our analysis are presented in table 2. The table presents values of parameters for each of the chosen functions, and the residual sum of squares, RSS.

The RSS is calculated as follows:

$$RSS = \sum_{i=1}^{n} (y_i - f(x_i))^2 \quad (5)$$

where y_i is a predicted value of the function and $f(x_i)$ is the real value.

The best results were achieved for fourth function (RSS was the lowest for this function). We also tried more complex functions, including polynomials of higher degree and mix of polynomials and logarithms. However, for all those functions the RSS was only slightly lower than that for the fourth function.

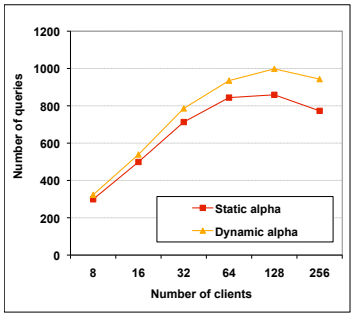


Fig. 17. Impact of calculating α based on load and latency

The results of the analysis indicate that our best predicting function can serve as a reasonable estimate for finding the optimal alpha value given the standard deviation of load and latency. To verify our analysis, we have ran another simulation where latency was changed many times during the runtime. The simulation was ran for 8, 16, 32, 64, 128 and 256 clients. One set of experiments was ran for fixed α value, and the other one used the formula we devised in our experiments that allows to calculate value of α based on load and latency. The results of our experiment are presented in Figure 17. Clearly they confirm that the throughput of the system increases when using the framework for estimating value of α.

5.2 Testing of a Real Implementation - small scale

We also deployed our implementation in a real setting on four nodes started on general-purpose PCs (Intel Core 2 Duo, 1GB RAM, Fast Ethernet network connection). The configuration involved three objects, CityInformation (provided by node 1), Flight (provided by nodes 2 and 3) and RailroadConnection (provided by node 3). Node 4 was used only for computational purposes. We ran the experiment for 10,000 CityInformation, 50,000 Flights (20,000 in node 2 and 30,000 in node 3) and 70,000 RailroadConnections. The database engine we used was PostgreSQL 8.2. We measured the response time for query workloads including small queries for all relevant CityInformation and medium queries for all objects mentioned above. We used various number of parallel clients and $\alpha = 0.33$.

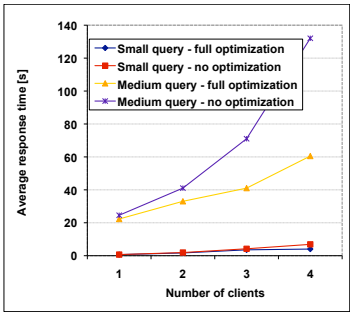


Fig. 18. Average Response Time in Real System

Figure 18 presents results for small and medium queries. It shows that the response time is significantly reduced when query optimization is used (for both small and medium queries). The response time may seem a bit high at the first glance. To give an idea of the actual overhead introduced by our system, we integrated all the databases used in the experiment above into one single database and tested a medium query from Java API using JDBC and one client. The query along with results retrieval took an average of 16s. For the same query, our system took 20s that is in fact comparable to the case of a local database. While the overhead introduced by DObjects cannot be neglected, it does not exceed reasonable boundary and does not disqualify our system as every middleware is expected to add some overhead. In this deployment, the overhead is mainly an effect of the network communication because data was physically distributed among multiple databases. In addition, the cost of distributed computing middleware and wrapping data into object representation also add to the overhead which is the price a user needs to pay for a convenient access to distributed data. However, for a larger setup with larger number of clients, we expect our system to perform better than *centralized* approach as the benefit from distributed computing paradigm and load distribution will outweigh the overhead.

5.3 Testing of a Real Implementation - PlanetLab

Previous section shows that query optimization in DObjects works well in tightly-coupled systems, where machines are connected using fast network and are not geographically distributed. To complete the picture, we have also run the tests of the system using PlanetLab[10] platform to see how significant geographical distribution influences the query execution. To run the experiment we have used the same databases as described in previous section (10,000 CityInformation, 50,000 Flights and 70,000 RailroadConnections). This time, however, the system consisted of 20 system nodes. The machines we used were modern general-purpose PCs available in PlanetLab network that varied in CPU power and

10. http://www.planet-lab.org/

memory size[11]. The latency between nodes also varied significantly. We chose nodes so that some of them had relatively low communication latency, whilst other nodes had significant communication overhead. Specifically, we identified a few clusters of lower latency nodes, and distances between clusters of nodes were significant. Additionally, a few separate nodes were chosen that were farther apart from the clusters as well as from other nodes (outlier nodes). PlanetLab experiment poses also additional challenge. The users do not have control over PlanetLab nodes, so the conditions in the network can change rapidly. First, the latency to some nodes can increase or decrease. Second, the load of some machines can also significantly vary in time.

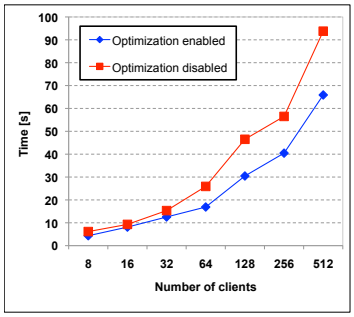


Fig. 19. Average Response Time in PlanetLab experiment

To run the experiment we loaded our system using 8 - 512 clients. The clients were started from independent PlanetLab nodes, and each node could start up to 32 clients. We measured an average response time for the medium-sized queries, and the results are presented in Fig. 19.

Our optimization scheme clearly improves the query response time. What can be noticed, the improvement is less drastic than for the tightly coupled system presented in previous subsection. This phenomenon is understandable, as PlanetLab network is much less predictable than the setup used in the previous subsection.

6 Conclusion and Future Work

In this paper we have presented the dynamic query processing mechanism for our P2P based data federation services to address both geographic and load scalability for data-intensive applications with distributed and heterogeneous data sources. Our approach was validated in different settings through simulations as well as real implementation and deployment. We believe that the initial results of our work are quite promising. Our ongoing efforts continue in a few directions. First, we are planning on further enhancement for our query migration scheme. We are working on incorporating a broader set of cost features such as location of the data. Second, we plan to extend the scheme with a dynamic migration of active operators in real-time from one node to another if load situation changes. This issue becomes important especially for larger queries which last longer time in the system. Finally, we plan to improve the fault tolerance design of our query processing. Currently, if a failure occurs on a node involved in execution of a query, such query is aborted and error is reported to the user. We plan to extend this behavior with possibility of failure detection and allocation of a new node to continue execution of the operator that was allocated to the failed node.

11. Each PlanetLab node has to satisfy minimum requirements; currently the minimum is set to a machine with 4 cores @ 2.4Ghz, 4GB RAM and 500GB HDD.

References

[1] D. Logothetis and K. Yocum, "Ad-hoc data processing in the cloud," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1472–1475, 2008.

[2] P. Jurczyk, L. Xiong, and V. Sunderam, "DObjects: Enabling distributed data services for metacomputing platforms," in *Proc. of the ICCS*, 2008.

[3] P. Jurczyk and L. Xiong, "Dobjects: enabling distributed data services for metacomputing platforms," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1432–1435, 2008.

[4] ——, "Dynamic query processing for p2p data services in the cloud," in *DEXA '09: Proceedings of the 20th International Conference on Database and Expert Systems Applications*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 396–411.

[5] T. M. Ozsu and P. Valduriez, *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 1999. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20{\&}path=ASIN/0136597076

[6] D. Kossmann, "The state of the art in distributed query processing." *ACM Comput. Surv.*, vol. 32, no. 4, 2000.

[7] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers, "Towards heterogeneous multimedia information systems: the Garlic approach," in *Proc. of the RIDE-DOM'95*, Washington, USA, 1995.

[8] A. Tomasic, L. Raschid, and P. Valduriez, "Scaling Heterogeneous Databases and the Design of Disco," in *ICDCS*, 1996. [Online]. Available: citeseer.ist.psu.edu/tomasic96scaling.html

[9] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom, "The TSIMMIS project: Integration of heterogeneous information sources," in *16th Meeting of the Information Processing Society of Japan*, Tokyo, Japan, 1994. [Online]. Available: citeseer.ist.psu.edu/chawathe94tsimmis.html

[10] A. Kemper and C. Wiesner, "Hyperqueries: Dynamic distributed query processing on the internet," in *The VLDB Journal*, 2001. [Online]. Available: citeseer.ist.psu.edu/kemper01hyperqueries.html

[11] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica, "Querying the internet with pier." in *VLDB*, 2003.

[12] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi, "The architecture of pier: an internet-scale query processor." in *CIDR*, 2005.

[13] R. Mortier, D. Narayanan, A. Donnelly, and A. Rowstron, "Seaweed: Distributed scalable ad hoc querying." in *ICDE Workshops*, 2006.

[14] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov, "The piazza peer data management system." *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 7, 2004.

[15] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou, "Peerdb: A p2p-based system for distributed data sharing." in *ICDE*, 2003.

[16] N. Ntarmos, P. Triantafillou, and G. Weikum, "Counting at large: Efficient cardinality estimation in internet-scale data networks." in *ICDE*, 2006.

[17] D. Suciu, "Distributed query evaluation on semistructured data." *ACM Trans. Database Syst.*, vol. 27, no. 1, 2002.

[18] N. Trigoni, Y. Yao, A. J. Demers, J. Gehrke, and R. Rajaraman, "Multi-query optimization for sensor networks." in *DCOSS*, 2005.

[19] A. P. Sheth and J. A. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Comput. Surv.*, vol. 22, no. 3, 1990.

[20] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov, "Schema mediation in peer data management systems." in *ICDE*, 2003.

[21] P. Cudré-Mauroux, K. Aberer, and A. Feher, "Probabilistic message passing in peer data management systems." in *ICDE*, 2006.

[22] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2007, pp. 1029–1040.

[23] D. Kurzyniec, T. Wrzosek, D. Drzewiecki, and V. Sunderam, "Towards self-organizing distributed computing frameworks: The H2O approach," *Parallel Processing Letters*, vol. 13, no. 2, 2003. [Online]. Available: http://www.worldscinet.com/ppl/13/1302/S0129626403001276.html

[24] M. N. Alpdemir, A. Mukherjee, A. Gounaris, N. W. Paton, A. A. Fernandes, R. Sakellariou, P. Watson, and P. Li, *Scientific Applications of Grid Computing*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, ch. Using OGSA-DQP to Support Scientific Applications for the Grid, pp. 13–24.

[25] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: A tiny aggregation service for ad-hoc sensor networks." in *OSDI*, 2002.

[26] R. van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining." *ACM Trans. Comput. Syst.*, vol. 21, no. 2, 2003.

[27] P. Yalagandula and M. Dahlin, "A scalable distributed information management system." in *SIGCOMM*, 2004.

[28] R. Huebsch, M. Garofalakis, J. M. Hellerstein, and I. Stoica, "Sharing aggregate computation for distributed queries," in *SIGMOD*, 2007.

[29] S. Xiang, H. B. Lim, K.-L. Tan, and Y. Zhou, "Two-tier multiple query optimization for sensor networks," in *Proceedings of the 27th International Conference on Distributed Computing Systems*. Washington, DC: IEEE Computer Society, 2007.

[30] W. Xue, Q. Luo, and L. M. Ni, "Systems support for pervasive query processing," in *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. Washington, DC: IEEE Computer Society, 2005, pp. 135–144.

[31] R. Huebsch, M. Garofalakis, J. M. Hellerstein, and I. Stoica, "Sharing aggregate computation for distributed queries," in *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2007, pp. 485–496.

[32] P. R. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. I. Seltzer, "Network-aware operator placement for stream-processing systems." in *ICDE*, 2006.

[33] K. Aberer, A. Datta, M. Hauswirth, and R. Schmidt, "Indexing data-oriented overlay networks," in *Proc. of the VLDB '05*, 2005, pp. 685–696.

[34] P. Ganesan, M. Bawa, and H. Garcia-Molina, "Online balancing of range-partitioned data with applications to peer-to-peer systems," Stanford U., Tech. Rep., 2004. [Online]. Available: citeseer.ist.psu.edu/ganesan04online.html

[35] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. A. Olson, "Mariposa: A new architecture for distributed data," in *ICDE*, 1994. [Online]. Available: citeseer.ist.psu.edu/stonebraker94mariposa.html

[36] N. Tatbul, U. Çetintemel, and S. B. Zdonik, "Staying fit: Efficient load shedding techniques for distributed stream processing," in *VLDB*, 2007, pp. 159–170.

[37] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system," in *Proceedings of the ACM SIGCOMM '04 Conference*, 2004.

[38] B. G. Sean Rhea, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "Opendht: A public dht service and its uses," in *SIGCOMM*, 2005.

[39] G. Paroux, B. Toursel, R. Olejnik, and V. Felea, "A java cpu calibration tool for load balancing in distributed applications." in *ISPDC/HeteroPar*, 2004.