Technical Report

TR-2007-015

DObjects: a metacomputing framework with dynamic query processing for distributed data networks

by

P. Jurczyk, L. Xiong

Mathematics and Computer Science

EMORY UNIVERSITY

DObjects: A Metacomputing Framework with Dynamic Query Processing for Distributed Data Networks

Pawel Jurczyk
Department of Math&CS
Emory University
pjurczy@emory.edu

Li Xiong
Department of Math&CS
Emory University
lxiong@emory.edu

Abstract

Many contemporary applications rely heavily on large scale distributed and heterogeneous data sources; examples are enterprise end-system management, workflow management, and computer-supported collaborative work. The key constraints for building a distributed data query infrastructure for such applications are: 1) scalability: the system needs to scale both for number of data sources and number of clients without sacrificing data integrity, 2) heterogeneity: the system needs to provide a unified access for heterogeneous data sources, and 3) network and resource dynamics: query processing needs to adapt to dynamic network and resource conditions to maximize resources utilization and scalability. To address these issues, we designed and developed DObjects, a general-purpose distributed data objects framework that provides an easy and scalable way of querying and operating data from heterogeneous data sources. We describe our novel architecture built on top of a metacomputing platform. By hiding the details of resource sharing and using object/relational mapping, DObjects simplifies the development of efficient and scalable integrated distributed data networks. In addition, we present the details of the dynamic query execution engine within our metacomputing framework that dynamically adapts to network and node conditions. We present an extensive performance evaluation through simulations as well as a real implementation.

1 Introduction

An emerging class of distributed data intensive applications rely on large scale distributed and heterogeneous data sources; examples are enterprise end-system management, workflow management, and computer-supported collaborative work. Consider a nation-wide IT network provider that owns hundreds of thousands of network devices across the country. The devices produce hundreds of megabytes of data every hour (such as alarms and maintenance events). It is nearly impossible to store the data in one, nation-wide database server. Instead, hundreds of smaller servers with potentially heterogeneous database systems are used where each of them connects to local devices and stores information reported by them. In order to develop applications such as an enterprise-scale device management system or a report generation tool, data from multiple distributed and heterogeneous sources must be queried and integrated. Different applications may require different query support. To satisfy diagnostic or management purposes, one would need adhoc or batch queries support. On the other hand, for monitoring applications, one would need continuous queries. Below we consider the key characteristics of such applications and the opportunities they present on developing a common infrastructure for querying and operating the distributed and heterogeneous data sources. We will mainly focus on the architectural and query processing issues.

Scale. The scale of the applications we consider requires good performance and scalability without sacrificing data integrity. At one end of the spectrum, earlier distributed database systems [12], such as SDD-1, R* and Mariposa, share modest targets for network scalability (a handful of distributed sites). They focus on making distribution transparent to users and applications and encapsulating distribution with ACID guarantees. At the other end of the spectrum, internet scale query systems, such as Astrolabe [28] and PIER [7], target a very large scale (thousands if not millions of nodes) but sacrifice on consistency semantics and data update functionalities. The scale of the applications we consider falls somewhere in the middle of the spectrum but with a stronger requirement on query semantics and query complexities than Internet-scale applications. This requires not only geographic scalability that allows systems scales to massively distributed geographic patterns but also load scalability that allow the system to expand or contract its resource pool to accommodate heavier or lighter query loads. Geographic scalability has been the focus of most Internet scale query systems that developed efficient distributed indexes and query routing schemes to address massively distributed data sources. In contrast, load scalability has caught less attention. This motivates us to explore the notion of metacomputing or resource sharing for dis-

tribution of query processing to achieve load scalability of distributed query systems. The metacomputing paradigm originated from distributed systems community offers a cost effective way to provide load scalability where nodes can share resources and form a virtual supercomputer for query processing. Our primary viewpoint is that not only data distribution but also computation or query processing distribution should be considered to achieve both geographic scalability and load scalability.

Heterogeneity. In our applications, individual heterogeneous data sources may have its own specific interface, and may not be designed to interact with other databases. Most research prototypes for heterogeneous database systems use a three-tier architecture where clients connect to a mediator, a mediator maintains a global schema and parses a query and executes some of the operations of a query, and a wrapper for every component database translates query request of mediator to the component's API and translates result back [13]. It remains an open direction to distribute the mediator and have a fully decentralized architecture to support heterogeneous data sources. Query processing also needs to dynamically estimate the cost associated with querying heterogeneous data sources through wrappers. In addition, a query infrastructure requires a unified data representation as well as an intuitive query interface for the heterogeneous data sources. An important and related challenge that we will not focus on in this paper is the semantic challenge of integrating data sources with heterogeneous schemas. We refer readers to a survey of the issues [24] and a few recent proposals [6, 2] that focus on schema mediation in distributed systems that will complement the solution we propose in this paper.

Network and Resource Dynamics. Another characteristics of the large scale data applications in the wide area networks is the dynamic network conditions (such as node availability and network latency) and resource conditions (such as load of the nodes). This has a few implications on the distribution of query processing. Conventional query processing strategies [13] will not work well because of their assumption of constant throughput and homogeneous or constant network delay (e.g. they only consider data size to reduce data shipping). In order to guarantee the usage of resources in the most efficient way to achieve the desired scalability, query processing must adapt to the network and resource dynamics. In addition to node joins and departures, interconnection network between nodes is dynamic and often changes. When some nodes become harder to reach, query execution process has to be changed accordingly and needs to avoid those locations. Second, the computational power or load of different nodes constantly changes. Query executor and optimizer have to be able to notice such changes and respond with proper action in order to achieve load scalability.

Contributions. In this paper we focus on the architectural and query processing issues for such applications and present our approach for supporting querying and operating large scale distributed and heterogeneous data sources. We propose DObjects, an easy-to-use and scalable distributed data objects framework based on a novel metacomputing architecture without centralized services. By hiding the details of resource sharing, using object/relational mapping, and dynamically adapting query optimization based on network and resource conditions, DObjects simplifies the development of efficient and scalable integrated distributed data networks. The main contributions can be summarized as follows.

First, the system employs a novel *distributed* and *decentralized* architecture that uses a *metacomputing* platform as a resource sharing substrate. The metacomputing platform consists of independent nodes which form a virtual abstraction of a single database management system. Each node can either be responsible for retrieving data from data sources through data adapters (wrappers) or simply provide computational power for query processing. The system scales well with the size as well as the load of the network.

Second, the system provides an integrated and transparent querying mechanisms for data objects. Front-end users build queries and get back an array of data objects called *persistent entities*. Persistent entities can be modified, deleted or created easily. DObjects' query language strictly follows *object-oriented fashion* of data representation. Data from multiple heterogeneous sources are joined transparently from user's point of view. The system supports various options of *query types*, such as ad-hoc queries, batch queries and continuous queries for various applications.

Third, the system includes an iterative and dynamic distributed *query execution and optimization* scheme. It optimizes both throughput and response time to maximize resource sharing and achieve scalability. Queries or subqueries are dynamically 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 manner (right before the execution of each query operator). Such an approach guarantees the best reaction to network and resource dynamics.

Finally, we have a full implementation and perform extensive experimental results validating our approach through simulations and the implementation. Our solution, as we believe, can be used in many classes of applications as an underlying database framework. It also provides an extensible platform where new ideas can be evaluated.

Organization. Section 2 provides a review of related work. Next, Section 3 presents an overview of our framework, including its novel architecture, data presentation, data operations, and query language. Section 4 presents details of

our novel query execution and optimization engine. Section 5 presents an evaluation of the system through simulations and real implementation. Finally, Section 6 provides a brief summary and discussion of future work.

2 Related work

Our work on DObjects was inspired and informed by a number of research areas. We attempt to provide a rough overview of related work in this section.

Distributed Databases and Distributed Query Processing. Distributed databases has been a subject of research for quite a long time [19, 12]. 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.

A number of distributed query systems were targeted at Internet-scale in recent years. HyperQueries framework [11] 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[1]. Instead of data objects, response to user queries is XML which has references (active links) to Web services providing given information. PIER [9, 7] is one of the first general-purpose relational query processor 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. Open issues remain for optimizing complex, multi-operator queries. The initial work in Seaweed [16] 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).

Another related body of work such as Piazza [5] and PeerDB [17] 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 [18]. A number of works are focused on distributed query evaluation on semi-structured data or XML using techniques such as query decomposition [26] and partial evaluation [27]. These research agendas complement our research.

Commercial systems provide support for distributed heterogeneous databases to some extent. For instance, Oracle Distributed Data System (incorporated in Oracle Database) allows cooperation between Oracle Database and non-Oracle systems. However, it can use only predefined data sources, such as databases supporting ODBC. Another notable example is Objectivity/DB[2], an object-oriented database system.It employs a novel architecture for computation and data distribution and uses object-oriented fashion to store data. However, the main limitation is that objects are stored in data sources in a system-specific way and one cannot use it with heterogeneous databases.

Data Streams and Continuous Queries. Quite large effort was put into the area of continuous query processing. Several research prototypes (such as TAG [15], Astrolabe [28], SDIMS [32]) were focused on supporting single distributed aggregation queries. There are also recent works [27, 8] focused on multi-query optimization. The vision of DObjects is to support ad hoc queries as well as continuous queries. At this stage we support basic continuous query processing and it is on our research agenda to adopt the more sophisticated continuous query optimization techniques into our framework.

The query optimization in DObject is most closely related to SBON [21]. It 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 local query migration and explicitly examine the relative importance of network latency and system load in the performance.

Distributed Systems Architectures. Distributed metacomputing paradigm offers great scalability at relatively low price by sharing distributed resources to solve large-scale computing problems. Grid systems, such as Globus Toolkit [4] and UNICORE [22], provide general frameworks for running software on grid architecture using different approaches such as component-based model [1] and distributed objects paradigm [31]. Recently, we can observe the emergence of applications that use distributed and grid architectures for query processing. A novel mediator-based database architecture for grids was proposed in [29]. The grid community has also explored the idea of relocating data preprocessing closer towards data locations [30]. However, grid systems require centralized services for authentication or job submission which limit these solutions. Our framework is built on top of a decentralized metacomputing platform H2O [14, 10] that avoids the administrative burden related to using grid systems and makes resource sharing easier for providers, in the spirit of the P2P model.

[1] http://activexml.net/

[2] http://www.objectivity.com/

3 System Overview

In this section we present an overview of DObjects framework. We describe its novel system architecture, present the concept of persistent entities as data wrappers, discuss data operations and types of queries supported by the system, and describe details of the query language. Along the way, we present relevant implementation details of the framework.

3.1 System Architecture

Figure 1 presents our vision of deployed DObjects framework. The system has no centralized services and thus allows system administrators to avoid all the burden in this area. It uses the *metacomputing* paradigm as a resource sharing substrate. Each node in the system provides its *computational power* that can be used by others during query execution. In addition, nodes can run *data adapters* which pull data from external data sources and transform it to a uniform format that is expected while building query responses. Front-end 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 database system consisting of all the participating nodes.

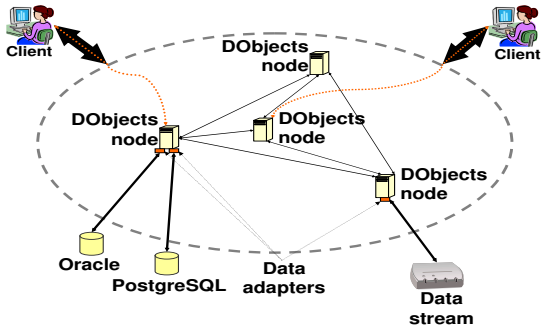

Figure 1: System Architecture

Our current implementation builds on top of a Java metacomputing platform, H2O[10, 14], that provides lightweight, decentralized and peer-to-peer resource sharing and communication. Data adapters are implemented using a Java object/relational mapping API, Hibernate[3], to pull data from heterogeneous relational data sources such as MySQL, PostgreSQL, Oracle etc. Adapter interface can be also implemented for any abstract sources providing data (for example, data file, network device or any system device driver).

3.2 Persistent Entities

When a user starts a query, DObjects returns persistent entities which are data represented as objects. From user's perspective, query responses are objects of desired type. Each data object has a set of *attributes*, divided into two groups: *simple* and *referential*. Simple attributes represent simple types, such as numbers or strings. Referential attributes follow an object-oriented idiom and allow the definition of *association*, *composition* or *collection* relations between data objects. Thus, when a referential attribute is accessed, another persistent entity, or collection of persistent entities, is obtained.

A set of available data types in the system along with their attributes is defined in the system configuration. Each configuration entry has a full description of an object, i.e. its type name and a list of simple attributes and referential attributes. When a referential attribute is defined, one has to specify the foreign key information that is required to join the referencing object and referenced object. It also specifies a list of nodes (sources) where given objects can be found. Each source is specified with: 1) name of the node, 2) remote data object name, and 3) attribute mappings that define the semantic mappings between the remote data object and the current object. There is no centralized copy of the global configuration but rather it is replicated and synchronized at every node. The design choice of a decentralized global configuration is appropriate and efficient in our system as we focus on load scalability and the distribution of query execution.

Listing 1: Persistent Entity Configuration

```
<persistent-entity name="NetworkDevice">
 <definition>
  <key><attribute name="id" type="Integer"/></key>
  <attribute name="name" type="String"/>
  <list name="lAlarms" type="Alarm"
       local-key="id" remote-key="r_nd_id"/>
  <list name="lMEvents" type="MaintenanceEvent"
       local-key="id" remote-key="r_nd_id"/>
 </definition>
 <sources>
  <source name="place_a" remote-object="X">
   <attribute-mapping local-name="id" remote-attr="id"/>
   <attribute-mapping local-name="name" remote-attr="dn"/>
  </source>
 </sources>
</persistent-entity>
```

Our implementation uses XML for the configuration of persistent entities and an example configuration is provided in listing 1. It defines a persistent entity of NetworkDevice (representing any network equipment) that has 2 referential attributes: list of Alarms (representing alarm information from device), and a list of MaintenanceEvents (representing maintenance entries created after device maintenance). In addition, Alarm object has a list of Action objects (representing action performed in answer to Alarm) and their configurations were omitted due to space constraints.

3.3 Data Operations

DObjects supports all standard data operations. Users can query, create, delete and update persistent entities. All persistent entities can be queried, but depending on other allowable data operations, they can be classified into 4 groups: 1) *editable/not-pinned* that allows update, delete,

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

and create as well as change of location, 2) *editable/pinned* that allows update, delete, and create but no change of location, 3) *not-editable/not-pinned* that allows no update, delete or create but allows change of location, and 4) *not-editable/pinned* that allows no update, delete, create or change of location. DObjects implements a *three-phase commit protocol* [25] to support the above data operations with transaction semantics. In this paper, however, we will only focus on query operations and the associated query processing issues.

To support various applications, DObjects supports *ad-hoc queries*, *batch queries* and *continuous queries*. In case of an *ad-hoc query*, the system provides response immediately after its completion and execution of user's code is blocked until the query is finished. As the opposite, batch queries are executed in background. When a *batch query* is executed, user's code is not blocked and notifications about results are provided on a given listener. What is important, user can get results *incrementally* and operate on partial results while the query is executed. DObjects also supports basic *continuous queries* that remain in the system until explicit termination. Whenever new data comes from adapter, system can determine whether it is potential answer to the standing queries. If so, the response is prepared and sent to appropriate users.

3.4 Query Language

The DObjects query language strictly follows the object-oriented fashion of the data representation of persistent entities. A user creates a query by building a *hierarchy of objects*. Each query is created for a given persistent entity type and specifies which simple or referential attributes should be *populated*. In case of referential attribute, the user builds the referenced object which again specifies simple and referential attributes to be populated. The process can be continued until desired level in the hierarchy is reached. In addition, the query could specify *conditions* for objects in the hierarchy, both for simple attributes and recursively for referential attributes (objects).

Listing 2: Example Query

```
Query query = system.getQuery("NetworkDevice");
//Populated attributes in NetworkDevice
AttributePopulator populator =
    query.createAttributePopulator();
populator.addPopulatedAttribute("name");
//Populated attributes in each Alarm object
AttributePopulator aPop =
    populator.createRefPopulator("lAlarms");
aPop.addPopulatedAttribute("message");
populator.addPopulatedRefAttribute(aPop);
//Prepare constraints for results
QueryCondition condition =
    query.cerateQueryCondition();
condition.addCondition("id", new Integer(50),
    QueryCondition.SMALLER);
QueryCondition aCondition =
    condition.createQueryCondition("lAlarms");
aCondition.addCondition("severity", new Integer(1),
    QueryCondition.EQUAL);
condition.addCondition(aCondition);
```

Because our data objects are hierarchical, query language for our system could be implemented using any language that allows one to specify populated attributes or conditions for given attribute in objects hierarchy. Thus, XPath or XQuery as well as SQL-like (or OQL-like) language could be used. Our current implementation provides a Java API that allows users to create queries. An example of an ad hoc query is presented in listing 2. It defines a query for NetworkDevice (as defined in listing 1) with populated simple attribute of "id" and referential attribute of Alarm list. Each Alarm has populated "severity" attribute.

4 Query Execution and Optimization

In this section, we focus on the query processing issues, 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. We present relevant implementation details for the presented strategies.

4.1 Overview

As we have discussed, the key to query processing in our metacomputing framework is to have a decentralized and distributed query execution engine that dynamically adapts to network and resource conditions. While adapting "textbook" distributed query processing techniques such as distributed join algorithms and the learning curve approach for keeping statistics about data adapters, our novel 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 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 best reaction to changing load or latency conditions in the system.

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 2 that corresponds to the query example in listing 2. The query plan contains such elements as *joins*, *horizontal* and *vertical data merges*, and *select* operations that are performed on data adapters. Each type of elements 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.

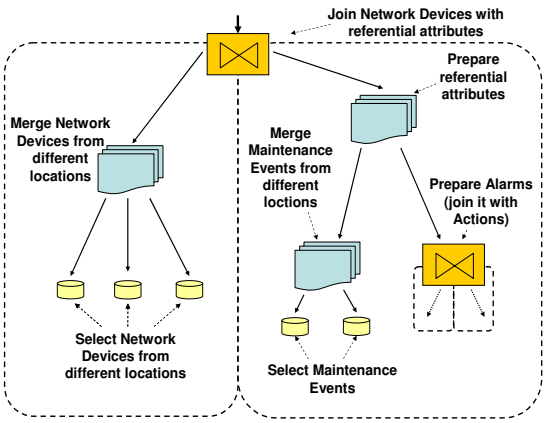

Figure 2: High-level Query Plan Illustration

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 was delegated to a remote node, the remote node has a full control over the execution of any child steps that are required. The process works recursively and iteratively, therefore remote node could decide about moving 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.

Algorithm 1 Local Algorithm for Query Processing

```
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; return
6: end if
7: execute active element;
8: for all child nodes of active element do
9:     go to step 2
10: end for
11: return result to parent element
```

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's important to note that optimization for each element in the query plan is performed iteratively, just before given element is going to be executed. We describe the optimization strategies of each type of operators below.

Join. Join operator is created when user issues a query for an object with populated referential attributes (objects). In this case, join between main objects and the referenced objects have to be performed. The optimization is focused on finding appropriate join algorithm and the order and place of branch executions. Our current implementation uses semi-join algorithm and standard techniques for result size estimations. We also plan to implement bloom-join algorithm and expect further enhancement in this area.

Data merge. Data merge operator is created when data objects are split among multiple nodes (horizontal data split) or when attributes of object are located on multiple nodes (vertical data split). Since an intention of data merge operation is to merge data from multiple input streams, this operation needs to execute its child operations before it is finished. Therefore, our optimization approach for this operator tries to maximize the parallelization of subbranches execution. This goal is achieved by executing each subquery 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 needs 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 faster response time. For instance, enforcing an order (sort) to queries allows us to use merge-joins in later operations. Next, optimal response chunks are built in order to support queries returning large datasets. Specifically, in case of heavy queries, we are implementing an iterative process of providing users with smaller pieces of final response. In addition to helping to maintain considerable node load level in terms of memory consumption, such a feature is especially useful when building a user interface that needs to accommodate long query execution.

4.3 Query Migration

A key of our query processing is a 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 manner (just before the execution of each element in the query plan). In order to determine the best (remote) node for possible query migration and execution for a query element, 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) transmission delay or communication cost between nodes, and 2) 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:

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

where DS is the size of the necessary data to be migrated for query execution, $latency_{i,j}$ is the network latency between node 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 terms give normalized values between 0 and 1 and the resulting cost metric also gives a normalized value between 0 and 1.

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 in the list, it calculates the cost of migration and compares the minimum (best candidate) 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 is moved to the best candidate. Otherwise, the execution will be performed at the current node. Formally, the decision of a migration is done 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 nodes in its 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 local execution only considers the load of the current node.

The above cost metric consists of two cost features, namely, the *network latency* and the *load* of each node. Below we present techniques for computing each of them efficiently.

Network Latency. 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 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 [3] 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 response. After the response is obtained, initiating node calculates the latency with remote nodes. As latency changes, 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 stable state for the most recent springs configuration.

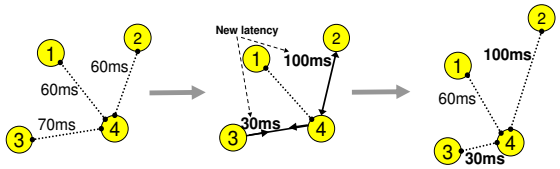

Figure 3: Illustration of Virtual Coordinates Computation for Network Latency

Figure 3 presents an example iteration of the Vivaldi algorithm. The first graph on the left presents current state of the system. New latency information is obtained (values of 30ms between nodes 3 and 4 and 100ms between nodes 2 and 4) in the middle graph. The rest length of spring between nodes 3 and 4 is shortened and between nodes 2 and 4 is made wider. As the answer to new forces in the system, new coordinates are calculated. The new configuration is presented in the rightmost graph. Important fact about this algorithm is that, it has a great scalability which was proven by its implementation in some P2P solutions (e.g. in OpenDHT project [23]).

Load of Nodes. The second feature of our cost metric is the *load* of the nodes. Given our desired feature 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 heterogeneous environment. The main idea is based on time measurement of execution of predefined test programs that considers computing and multithreading capabilities of machines [20].

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.

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 and decentralized event framework, REVENTS[4], that is integrated with our metacomputing platform for efficient and effective asynchronous communication among the nodes.

[4]http://dcl.mathcs.emory.edu/revents/index.php

5 Experimental Evaluation

Our framework is fully implemented. In this section we present an evaluation through simulations as well as a real deployment of the implementation.

5.1 Simulation Results

We run our framework on discrete event simulator that gives us an easy way to test the system against different settings. The configuration of data objects was identical for all the experiments below and relates to the persistent entities and configuration mentioned in Section 3. The configuration of data sources for persistent entities is as follows: object NetworkDevice was provided by node1 and node2, object Alarm was provided by node3 and node4, object MaintenanceEvent by node1 and finally object Action by node2. All nodes with numbers greater than 4 were used as computing nodes. Table 1 gives a summary of important system and algorithmic parameters for different experiments.

Table 1: Experiment Setup Parameters. * - variable parameter

Test Case	Figure(s)	Nodes#	Clients#	α
α vs. Query Workloads	5	6	14	*
α vs. Nodes#	6	*	32	*
α vs. Clients#	7	6	*	*
Comparison of Query Optimization Strategies	8	6	14	0.33
System Scalability	9, 10	6	*	0.33
Impact of Computational Resources	11	*	32	0.33
Impact of Network Latencies	12	6	14	0.33

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 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.

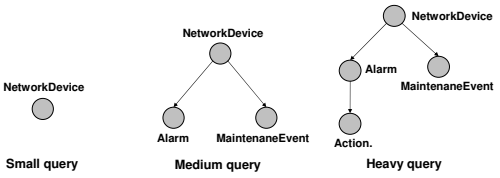

Figure 4: Illustration of Query Workloads

For the first case, we tested three query workloads: 1) small query for NetworkDevice objects without referential attributes (therefore, no join operation was required), 2) medium query for NetworkDevice objects with two referential attributes (list of Alarms and MaintenanceEvents), and 3) heavy query with two referential attributes of NetworkDevice of which Alarm also referential attribute. The queries are illustrated in figure 4. The second case varied the number of computational nodes and used medium query which was submitted by 32 clients simultaneously. The last

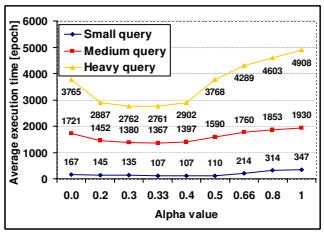

Figure 5: Parameter Tuning - Query Workloads

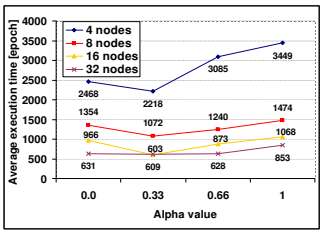

Figure 6: Parameter Tuning - Number of Nodes

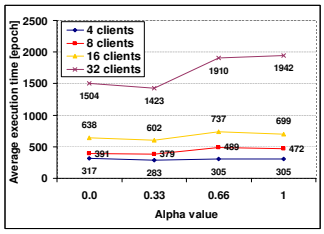

Figure 7: Parameter Tuning - Number of Clients

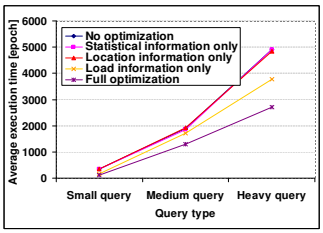

Figure 8: Comparison of Different Query Optimization Strategies

case used a system consisting of 6 nodes and varied number of clients submitting medium queries. The latency between nodes (which has impact on nodes placement in virtual coordinates system) was randomly selected every time the test was started.

Figure 5, 6 and 7 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 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 ($\alpha = 1.0$[5]). For all further experiments we use the best α value which was determined 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 (naive query execution where children of current query operator are executed one by one from left to right), 2) statistical information only (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 8. They clearly show

[5]For this case we had to modify equation 1 and remove alpha factor.

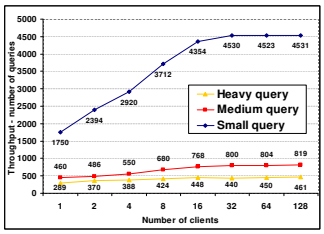

Figure 9: System Scalability (Throughput) - Number of Clients

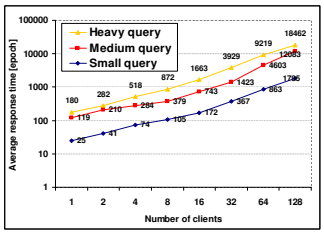

Figure 10: System Scalability (Response Time) - Number of Clients

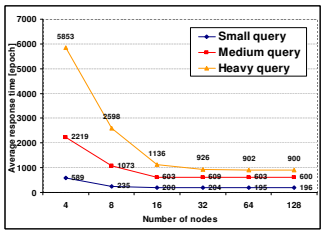

Figure 11: Impact of Computational Resources

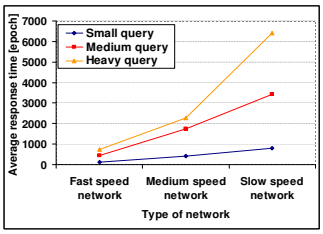

Figure 12: Impact of Network Latency

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 approaches. The performance improvements are most manifested in the heavy query workload which requires more join processing and hence computation.

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.

Figures 9 and 10 present the throughput and average response time for different number of clients respectively. Figures 9 shows average number of queries that our system was capable to handle for given number of clients (during specified time frame). It shows quite well how the system reaches its maximal throughput - for small query, average value around 4500 queries per specified time frame was the maximum. As can be noticed, the number of queries increases as number of clients increases. This phenomenon is expected, as with the increase of number of clients, saturation of nodes with workload also increases in the system. However, when it reaches critical level (each node is heavily loaded), new clients cannot obtain any new resources. Thus, the throughput will not increase. For small and medium queries the critical value is 32 nodes (when we use more than 32 nodes throughput does not change significantly). Figure 10 reports average response time and shows a good scalability. Please note that the figure uses logarithmic scale for y-axis for a better clarity.

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 different number of nodes with 32 clients simultaneously querying the system. The results are provided in Figure 11. Our query processing effectively reduces average response time when more nodes are available. For 32 clients and small queries, 8 nodes appears to be enough as an increase to 16 nodes does not improve response time significantly. Consequently, for medium size queries 16 nodes appears to be enough (no improvement in average response time if 32 nodes were used). 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 can benefit from larger number of available nodes.

Impact of Network Latency. Our last experiment using simulation was aimed at finding the impact of the average network latency on the performance. The configuration setup involved again 6 nodes and 14 clients. We report results for three network speeds: fast network that relates to FastEthernet network offering speed of 100MBit/s, medium speed network that can be compared with Ethernet network of speed 10MBit/s, and finally slow network which represents 1MBit/s connection speed.

The result is reported in Figure 12. 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 query is larger than the medium and small queries, and therefore the time it takes to transfer this data in slower network will have much larger impact on overall efficiency.

5.2 Testing of a Real Implementation

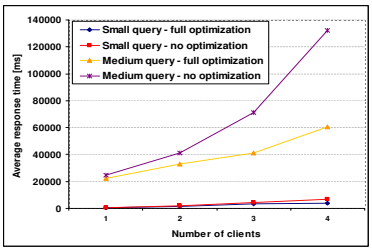

Figure 13: Average Response Time in Real System

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, NetworkDevice (provided by node 1), Alarm (provided by nodes 2 and 3) and MaintenanceEvent (provided by node 3). Node 4 was used

only for computational purposes. We ran experiments in which we measured response time for different query workloads. Figure 13 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).

6 Conclusion

We have presented DObjects, a distributed data objects framework that facilitates integration of data from large scale heterogeneous sources. We have presented the novel architecture of the system based on a metacomputing platform for addressing both geographic and load scalability, and discussed in detail the dynamic query processing engine with local query migrations that dynamically adjusts to the network and resource conditions. Our approach was validated in different settings through simulations as well as real implementation and deployment. We believe that initial results of our work are quite promising. It can be used in many classes of applications as an underlying database framework. A current implementation is available for download [6]. The framework also provides an extensible platform for future research. The ongoing and future efforts include further enhancement for query optimization with a broader set of cost features, advanced optimization for continuous queries, and efficient transaction support for data modification and replication.

References

[1] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, and T. Dahlgren. The cca component model for high-performance scientific computing. *Concurrency and Computation: Practice & Experience*, 18(2), 2006.

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

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

[4] I. Foster. Globus toolkit version 4: Software for service-oriented systems. *Lecture Notes in Computer Science*, 3779, 2006.

[5] 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.*, 16(7), 2004.

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

[7] 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.

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

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

[10] P. Hwang, D. Kurzyniec, and V. Sunderam. Heterogeneous parallel computing across multidomain clusters. In *Proceedings of 11th European PVM/MPI Users' Group Meeting*, LNCS. Springer-Verlag, 2004.

[11] A. Kemper and C. Wiesner. Hyperqueries: Dynamic distributed query processing on the internet. In *The VLDB Journal*, 2001.

[12] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32, 2000.

[13] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4), 2000.

[14] D. Kurzyniec, T. Wrzosek, D. Drzewiecki, and V. Sunderam. Towards self-organizing distributed computing frameworks: The H2O approach. *Parallel Processing Letters*, 13(2), 2003.

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

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

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

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

[19] T. M. Ozsu and P. Valduriez. *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 1999.

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

[21] 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.

[22] M. Riedel and D. Mallmann. Standardization processes of the unicore grid system. In *Proceedings of 1st Austrian Grid Symposium 2005*, 2006.

[23] 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.

[24] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3), 1990.

[25] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. 1987.

[26] D. Suciu. Distributed query evaluation on semistructured data. *ACM Trans. Database Syst.*, 27(1), 2002.

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

[28] 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.*, 21(2), 2003.

[29] A. Whrer, P. Brezany, and A. M. Tjoa. Novel mediator architectures for grid information systems. In *Future Generation Computer Systems*, 2005.

[30] A. Wohrer, P. Brezany, L. Novakov, and A. M. Tjoa. D3G: Novel approaches to data statistics, understanding and preprocessing on the grid. In *AINA*, 2006.

[31] Y.-J. Woo and C.-S. Jeong. Distributed object-oriented parallel programming environment on grid. *Lecture Notes in Computer Science*, 2668, 2003.

[32] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM*, 2004.

[6] http://www.mathcs.emory.edu/Research/Area/datainfo/DObjects