DObjects: Enabling Distributed Data Services for Metacomputing Platforms

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. The key constraints for building a distributed data query infrastructure for such applications are: scalability, consistency, heterogeneity, and network and resource dynamics. We designed and developed DObjects, a general-purpose query and data operations infrastructure that can be integrated with metacomputing middleware. This demo proposal describes the architecture and the dynamic query processing functionalities of our data services and shows how they are integrated with a metacomputing framework offering users an open platform for building distributed applications that require access to data integrated from multiple data sources.

1. INTRODUCTION

Many distributed data intensive applications rely on large scale distributed and heterogeneous data sources; examples are enterprise end-system management and large-scale scientific data management. Consider a national IT network provider who owns hundreds of thousands of network devices across the country and utilizes hundreds of small servers with potentially heterogeneous database systems to connect to these local devices and store their information. In order to develop applications such as an enterprise-scale device management system or a report generation tool, data from distributed and heterogeneous sources must be operated. Such applications are characterized by a number of features and requirements. First, the scale of the applications can vary from a handful to several hundreds of nodes and requires good *scalability* as well as data query and update functionalities with *data consistency*. Second, the *heterogenity* of data sources requires a unified and seamless data representation and query interface for the applications. Lastly, the *dynamics* of resource and network conditions requires applications to adapt dynamically in both query processing and transaction management in order to achieve scalability and data consistency.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '08, August 24-30, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

In this demonstration we present DObjects [9], a general-purpose infrastructure for querying and operating data from distributed and heterogeneous data sources. DObjects illustrates a number of research contributions. First, the system extends the *metacomputing* paradigm with a *distributed* mediator-wrapper based architecture. Metacomputing [13] is a paradigm in which distributed nodes share resources and form a networked virtual supercomputer, or metacomputer. Our data services extend this metacomputer to offer a scalable way for accessing and operating distributed and heterogeneous data sources. Second, the system includes a distributed query processing engine that deploys and executes (sub)queries 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 to optimize both response time and throughput.

Related work. It is important to distinguish DObjects from the many existing distributed database systems. At the first glance, distributed database systems have been extensively studied and many systems have been proposed over the years. Earlier distributed database systems [10], such as R* and SDD-1, share modest targets for network scalability (a handful of distributed sites) and assume homogeneous databases. The focus is on encapsulating distribution with ACID guarantees. Later distributed database or middleware systems, such as Garlic [3], DISCO [14], HERMES[2], TSIMMIS [4], Pegasus[1], target large-scale heterogeneous data sources. Many of them employ a *centralized* mediator-wrapper based architecture (see Figure 1) 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. Most recently, Internet scale query systems, such as Astrolabe [15] and PIER [8], target thousands or millions of massively distributed homogeneous data sources with a peer-to-peer (P2P) or hierarchical network architecture and focus on efficient query routing schemes for network scalability. However they sacrifice on functionalities of complex queries and data updates and typically relax the consistency guarantee. While it is not the aim of DObjects to be superior to any of 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 scala-

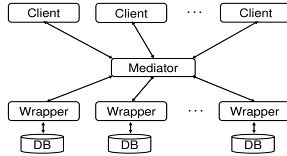

Figure 1: Typical Mediator-Based Architecture

bility as well as transaction semantics[1]. In spirit, DObjects is a *distributed* mediator-based system in which a federation of mediators and wrappers forms a virtual system in a P2P fashion. In addition to cost-based query optimization, the query processing engine of DObjects focuses on dynamically placing (sub)queries on the mediators for query-load balancing and scalability.

It is also important to position DObjects among the existing distributed system frameworks. Distributed systems technologies such as DCOM[2], CORBA[3], and RMI[4] support distributed objects paradigm and can be used to build distributed applications across heterogeneous networks. Our system builds on top of these technologies and offers a general platform for metacomputing with support for distributed data access and operation services. In particular, current implementation of DObjects builds on top of a resource sharing platform H2O [11] that builds on top of RMIX (an extension of RMI). The data services provided by DObjects offer query processing and transaction management substrates fully integrated with the metacomputing middleware and can be used easily and transparently in distributed applications for scalable data operations.

2. DOBJECTS DATA SERVICES

In this section we will give an overview of DObjects framework and describe its main features and functionalities.

System architecture. Figure 2 presents our vision of deployed DObjects framework. The system has no centralized services and uses the *metacomputing* paradigm as a resource sharing substrate. Each node in the system serves as a *mediator* and provides its *computational power* that can be used by others during query execution. Nodes can also serve as a data adapter or *data wrapper* that can pull data from external data sources and transform 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.

Our current implementation builds on top of a Java metacomputing platform, H2O[11], that provides lightweight, decentralized and peer-to-peer resource sharing and communication. Data adapters are implemented using a Java object/relational mapping API, Hibernate[5], to pull data from relational data sources such as MySQL, PostgreSQL, etc. Adapter interface can be also implemented for any abstract data sources such as data file, network device or any system device driver.

[1]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 readers to a survey of the issues [12] and a few recent proposals [7, 5] focusing on schema mediation in distributed systems.

[2]http://msdn2.microsoft.com/en-us/library/ms809340.aspx

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

[4]http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

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

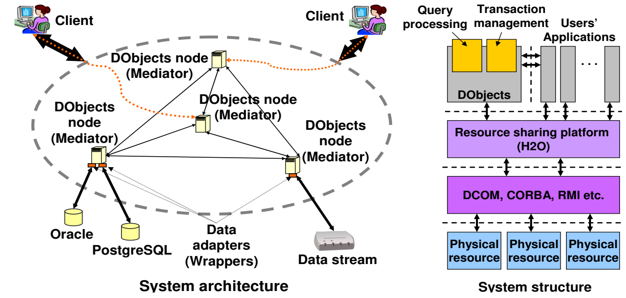

Figure 2: DObjects System Architecture

Data model. DObjects uses *persistent entities* as its data model 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 a 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 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. For systems with a handful DObjects nodes (the number of data sources can be still large), the configuration can be replicated and synchronized at every node as the cost of synchronization will be relatively small. For larger scale systems with more DObjects nodes, the global schema can be replicated at a subset of the DObjects nodes such as landmark nodes.

Data operations and query languages. DObjects supports all standard data operations. Users can query, create, delete and update persistent entities. Both synchronous and asynchronous queries are supported. For synchronous queries, the system provides response immediately after its completion and execution of user's code is blocked during the query execution. For asynchronous queries, user can get results *incrementally* and operate on partial results while the query is being executed. In order to support the various data operations with transaction semantics, a variation of three-phase commit protocol is implemented.

The query language for our system could be implemented using any language that allows one to specify attributes or conditions for a given attribute in objects hierarchy. XPath or XQuery as well as OQL-like language are all valid approaches. DObjects also provides its internal *query language API* which strictly follows the object-oriented fashion of the data representation of persistent entities. A user creates queries 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*.

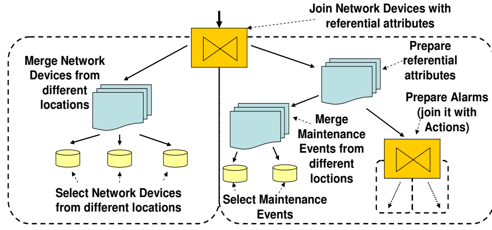

Figure 3: Example of high-level query plan.

3. QUERY PROCESSING

In this section, we present our query processing engine and highlight a few contributions. While adapting traditional distributed query processing techniques such as distributed join algorithms and the learning curve approach for keeping statistics about data wrappers, 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 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. 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 optimal placement of those sub-queries on the system nodes (mediators) to minimize the query response time and maximize system throughput.

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 3. 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 for optimization. Next, the node chooses active elements from the query plan one by one in a top-down manner for execution. Execution of an 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). If the active element is delegated to a remote node, the remote node has full control over the execution of any child steps that are required. The process works recursively and iteratively, thus a 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.

Query migration. The 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 (just before the execution of each element in the query plan) manner. 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. Equation 1 defines the cost, denoted as $c_{i,j}$, associated with migrating a query element from node i to a remote node j:

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:    move 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
```

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

where DS is the size of the necessary data to be migrated for query execution (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 give normalized values between 0 and 1 considering the potential variances between them.

To perform query migration, each node 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. To prevent a (sub)query being migrated back and forth between nodes, we require each node to execute at least one operator from migrated query plan before further migration. Alternatively, a Time-To-Live (TTL) strategy can be also implemented to limit the number of migrations for the same (sub)query. Formally, 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 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 local execution only considers the load of the current node.

The above cost metric consists of two cost features, namely, the *network 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 the two cost features we are using.

Communication between nodes. To compute the network latency between each pair of nodes efficiently, each DObjects node maintains a virtual coordinate [6], such that the Euclidean distance between two coordinates is an estimate for communication latency. 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.

Load of nodes. The second feature of our cost metric is *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 program that considers computing and multi-threading capabilities of machines.

4. DEMONSTRATION

In the demonstration, we will show the functionalities of our implemented system (a current implementation is available for download[6]), highlight a few key features, and show some of the under-the-hood details for interested audience.

The demonstration setup will use a client machine in the demonstration room and will connect to a set of physical machines (5-10) located at Emory University. All the remote nodes will have the H2O platform deployed and will act as DObjects nodes (data mediators). A subset of the nodes will have data stored in their local databases and will act as data wrappers. Through a user interface installed on the client machine, we will issue queries and update requests. The data setup will follow the example scenario of the IT network device management system and will include the following object types: NetworkDevice, Alarm, MaintenanceEvent and Action. NetworkDevice will have two referential attributes: list of Alarms, and list of MaintenanceEvents; Alarm objects will have one referential attribute: list of Action objects. Alarm objects will be horizontally split among few nodes.

The demonstration will focus on the following aspects: 1) DObjects configuration and setup showing how the system is configured and started, 2) basic data operations of the system including query building, results retrieval and various data updates, and 3) performance of the system in terms of query response time and throughput in different settings. During the runtime, some of the physical nodes will be dynamically enabled or disabled to illustrate the impact of node availability on system performance. Importantly, DObjects provides a logging option during data operations. The demonstration interface will offer a look at the logging information so the audience will be able to see under-the-hood how the query migration and optimization proceed.

In case the demonstration room does not provide connectivity with our system deployed at Emory University, we also have a backup plan and will follow alternative path for the demonstration. In order to present how DObjects works, we will start a few system nodes on the local machine and our demonstration application will connect to these local nodes. Such a setting will be sufficient to present how DObjects is configured, deployed and used. To better present the impact of query execution optimization, we will run simulations of various settings to demonstrate the results of our dynamic query processing and placement.

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

5. REFERENCES

[1] R. Ahmed, P. D. Smedt, W. Du, W. Kent, M. A. Ketabchi, W. A. Litwin, A. Rafii, and M.-C. Shan. The Pegasus Heterogeneous Multidatabase System. *Computer*, 24(12), 1991.

[2] A. Brink, S. Marcus, and V. s. Subrahmanian. Heterogeneous Multimedia Reasoning. *Computer*, 28(9), 1995.

[3] 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.

[4] 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*, Japan, 1994.

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

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

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

[8] 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.

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

[10] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 2000.

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

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

[13] L. Smarr and C. E. Catlett. Metacomputing. *Commun. ACM*, 35(6), 1992.

[14] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of disco. In *Proc. of the ICDCS*, 1996.

[15] 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.