



## Spring Simulation MultiConference

### Design and Implementation of A Data Server using a Peer-to-Peer Storage System

Journal:	<i>Spring Simulation Multiconference</i>
Manuscript ID:	SMC-2005-HPC-045.R1
Topic Area Conference:	High Performance Computing Symposium
Submission Type:	Final Paper
Date Submitted by the Author:	02-Feb-2005
Complete List of Authors:	Kumar, Vijay; Ohio State University, Biomedical Informatics Department Kurc, Tahsin; Ohio State University, Biomedical Informatics Catalyurek, Umit; Ohio State University, Biomedical Informatics Saltz, Joel; Ohio State University, Biomedical Informatics
Keywords:	peer-to-peer systems, distributed storage, data server, multi-dimensional data

# Design and Implementation of A Data Server using a Peer-to-Peer Storage System \*

Vijay Shiv Kumar, Tahsin Kurc,  
Umit Catalyurek, Joel Saltz

Department of Biomedical Informatics, The Ohio State University  
Columbus, OH, 43210

{vijayskumar,kurc,umit,jsaltz}@bmi.osu.edu

## Abstract

*This paper describes the implementation of a distributed data server system to support queries against multi-dimensional scientific datasets. The server system builds on a distributed, peer-to-peer object storage middleware. It is an extensible server in the sense that new user-defined filtering and processing functions can be added to the system. We investigate and experimentally evaluate several query scheduling policies within this system.*

**Keywords:** *Data servers, peer-to-peer, storage, data querying.*

## 1 INTRODUCTION

Analysis of datasets from simulations is key to gaining insight into the problem under study. Scientists are increasingly able to generate data at high rates and at high resolutions, with the help of high end computing platforms. Data items in a scientific dataset can often be associated with points in a multi-dimensional attribute space [1]. Applications that process and analyze multi-dimensional datasets use only a subset of all the data available in input datasets. Access to data items is described by a *range query*, which is a multi-dimensional bounding box in the underlying multi-dimensional attribute space of the dataset. Only the data items whose associated coordinates fall within the multi-dimensional box are retrieved. Queries can involve user-defined filtering and processing operations on the data.

In recent years, peer-to-peer systems have emerged as a viable technology to support storage and sharing of content and files across disparate data sources and data con-

sumers (clients). These systems provide support for location independent routing and data replication in a distributed environment. In this paper, we describe the implementation of a distributed and extensible data server system for efficient querying and subsetting of multi-dimensional datasets. The system is extensible in that user-defined filtering and processing functions can be integrated into the system. The server system is built using Pond [2], a distributed object storage system. We have implemented several query scheduling strategies to improve query response times when multiple queries are submitted to the system. We experimentally evaluate these strategies using a distributed collection of PC clusters.

## 2 RELATED WORK

Many peer-to-peer middleware systems have been developed for content delivery and file sharing [3, 4, 5]. Several research projects investigate methods for sharing of different data types and processing of complex queries. Gulbenden and Sahin [6] develop techniques for efficient sharing of multi-dimensional data in the Chord system [3]. Kalnis et. al. [7] have proposed a PeerOLAP architecture that supports On-Line Analytical Processing Queries. Like Pond, the system is decentralized, and results are cached arbitrarily over all the peers in the network. Harren et al. [8] have developed an architecture that supports the processing of complex queries in DHT-based peer-to-peer networks. They do not address multidimensional data and replication of data. PeerDB [9] is a P2P-based system for distributed data sharing. PeerDB is unique in that, it adopts mobile agents to assist in query processing. Agents can perform operations at peers' sites, thereby better utilizing the network bandwidth. PeerDB also caches responses for queries to improve performance on subsequent queries. Demirbas et al. [10] have developed a peer-to-peer indexing structure to efficiently process spatial queries in sensor networks. They present a query processing model that optimizes the number

---

\*This research was supported in part by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0203846, #ACI-0130437, #ANI-0330612, #ACI-9982087, #CCF-0342615, #CNS-0406386, #CNS-0426241, Lawrence Livermore National Laboratory under Grant #B517095 (UC Subcontract #10184497), NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003.

of peers contacted to service a query. Schmidt et al. [11] have designed Squid, a P2P information-discovery system that supports complex queries including partial keywords, wildcards and ranges. Using Space-filling curves, they map the multidimensional space onto the physical peers. Queries are optimized by generating only useful clusters of the SFC-based index. Their work does not specifically address the issue of serving multi-dimensional datasets and query scheduling techniques.

Parallel and distributed database systems have been a major topic in the database community [12] for a long time. Much attention has been devoted to query execution and implementation of joins on parallel systems. The proposed methods leverage parallelism by effectively partitioning the data and workload among the processors. Our system could leverage those techniques for the execution of parallel queries. In this paper, we target scientific datasets and settings, where data is stored as objects in a distributed object storage system, and investigate query scheduling algorithms in such an environment. The large aggregate memory of a PC or workstation cluster has attracted the attention of many researchers, including work on *cooperative caching* and prefetching [13, 14, 15]. Our approach is to deploy a peer-to-peer storage system so that distributed storage resources can be utilized for data replication and caching.

### 3 POND

Pond [2] is a prototype of the OceanStore framework<sup>1</sup>, which is designed to provide support for location-independent access to distributed persistent data in a uniform and universally available fashion. A Pond instance is made up of a number of virtual interactive resources (Pond clients or nodes) that make up an overlay network. Both resources and data are named using 160-bit GUIDs (Globally Unique Identifiers) from the same namespace. The Pond system uses Tapestry [16] as the underlying means of communication between the overlay resources that constitute a Pond instance.

The basic unit of data storage in Pond is a data object, identified uniquely by its AGUID (Active Globally Unique Identifier). Pond clients can create data objects, request to read data from the objects, and update the data objects. Objects are automatically and dynamically replicated in the system for greater availability and faster data access. One copy of a data object is identified as its primary copy, while the others are its secondary copies. The consistency of object replicas is also managed by Pond. At the basic level, the actual data is stored in the form of byte arrays in read-only blocks, identified uniquely by their block GUIDs. These blocks of data are cached throughout the system. The main entities present in a Pond system include: (1) Inner Ring

(Tier 1), which is a set of powerful and well-connected servers. Any one of the inner ring servers can store the primary copy of a data object, while the other servers automatically become replicas for that object. (2) Pond clients (Tier 2), which are applications that access Pond through local processes. When a new object is created in Pond, it is assigned an Inner Ring server. In order to retrieve a data object, a Pond client must provide a read request containing the following information: (1) the AGUID of the data object, (2) Version Predicate: A mechanism to request specific (possibly older) versions of the object based on its VGUID (Version GUID). (3) Selection: The specific portion of data that needs to be retrieved from within the object.

## 4 A DATA SERVER SYSTEM FOR MULTI-DIMENSIONAL DATASETS

Our main aim is to develop a multi-dimensional data server, utilizing Pond as its persistent storage middleware. The data server system provides support for partitioning of input datasets into chunks, indexing of these chunks, execution of queries with user-defined operations, and optimizations to speed up execution of multiple queries submitted to the system. This section describes in detail the architecture of our multi-dimensional data server system, the functions of each component of our system, and the different query execution strategies implemented in the system.

### 4.1 Overall System Architecture

The data server system is composed of a set of specialized Pond nodes that implement a typical server-client system. These nodes use Pond's rich interfaces to create, store, and read data (objects). Based on their functions, these nodes are classified as one of the following.

**Master Server:** A master server is a server that creates Pond objects in the system. Master servers have multi-dimensional datasets stored on local disks. On entering the system, they partition their dataset into data chunks and create Pond objects for each of these chunks. They also create indexes for efficient retrieval of these objects. The object creation phase is a one-time operation for a master server because objects in Pond remain forever, as long as the Pond instance is up and running. Once a master server is done creating data, it functions as a normal server that receives requests from clients, performs index lookup, retrieves data, and sends the results back to the client. A master server can also redirect some of its queries to other servers (called slave servers) if it becomes overloaded.

**Slave Server:** As the name suggests, the slave server acts as a slave to master servers. Excess queries can be redirected to slave servers to ease the burden on the master. A slave server can receive queries directly from a client. It can also redirect queries to other slave servers. The presence of many slave servers can also lead to greater caching

<sup>1</sup><http://oceanstore.cs.berkeley.edu/>

and replication of data.

**Resource Manager:** In some of the query execution strategies, we employ a centralized entity called a resource manager, which keeps track of the number of servers in the system and their loads. Servers and clients contact the resource manager to find out the least loaded server and redirect their queries to that server.

**Client:** A client is an application that sends queries to the servers requesting data. In our implementation, clients do not provide disk storage for the data blocks distributed throughout the system. Clients can send requests to master servers or slave servers.

## 4.2 Data Storage

In most cases only a small portion of the data is of interest to the scientist. Hence, it is a common optimization to partition the dataset into a set of chunks [1]. Each chunk contains a disjoint subset of data items in the dataset and is associated with a bounding box that encompasses the coordinates of all data items in the chunk. When the client requests for a portion of the data, only the relevant data chunks need to be retrieved and processed, not the entire dataset. In a distributed setting, data chunks are declustered across the storage nodes in the environment to achieve higher I/O bandwidths. Applying the notion of data chunks in the context of Pond, a dataset is divided into multiple chunks. Each chunk is stored as a data object in Pond. In our design, we rely on the underlying runtime system of Pond to provide efficient access to data across the nodes of the system through object replication.

## 4.3 Indexing

We use an Rtree index [17, 18] to rapidly search for data that intersects the query bounding box. The index contains the VGUID of the object as part of metadata for the object<sup>2</sup>. This is an optimization to speed up data object retrieval. Every time a Pond node issues a read request for a new object, it needs to provide version predicate. If it requests the latest version, the request is forwarded all the way to the inner ring for that object. Only when the node knows the VGUID of an object can it issue a read request for the object. Since scientific datasets are usually read-only (updates to data are in the form of adding new data elements to the dataset or new datasets to the system), there exists only a single version of every data object in the system. By including the VGUID of the object as part of the metadata at the time of index creation, the node can know the VGUID by just performing the index lookup, and thus avoid sending the VGUID request to the inner ring. Like data chunks in the dataset, the index is also serialized into a byte array and stored as

<sup>2</sup>We should note that more than one R-tree index file can be created if a single index for the dataset will be very big

a Pond object. This makes it possible for one server node to perform an index lookup on the data and answer queries into data created by another server node, thereby potentially reducing the workload on the latter.

## 4.4 User-defined Filtering and Processing of Data

User-defined operations, such as value-based filtering and data aggregation, are often a part of the query submitted against a scientific dataset. A user-defined operation is implemented as an object with well defined callback methods in the data server. Currently, the object is expressed using a Java language binding by subclassing a base class. Drawing from the notion of filters and aggregation objects in our earlier work [1], the interface of method callbacks consists of an initialization function, a processing function, and a finalization function. The initialization function is called when the query is received to initialize internal data structures. The process function is executed for each data chunk (represented as a byte array) that intersects the query bounding box. The finalize function is called after the last data chunk is processed.

In the data server system, user-defined objects, compiled as a Java class, are stored as Pond objects. This allows any server in the system to be able to execute queries with user defined processing operations. In our implementation, each user-defined operation is assigned a unique name and the AGUID of the operation is derived from its name by using the hash function employed in Pond. When a query is received by a server, the server retrieves the corresponding Pond object, extracts the Java class, and creates a Java object using the `ClassLoader` object provided by Java.

## 4.5 Query Scheduling and Execution

Queries by the clients take the form  $\{D, LB, UB, F, P\}$  where  $D$  is the dataset being queried,  $LB = \{lb_1, lb_2, \dots, lb_d\}$  is an array containing the lower bounds of the range query along each of the  $d$  dimensions, similarly  $UB = \{ub_1, ub_2, \dots, ub_d\}$  contains the upper bounds along the  $d$  dimensions,  $F$  is the name of the object corresponding to the processing operation to be applied to data, and  $P$  is the input parameter string to  $F$ . Once a server receives a query, it looks for a copy of the dataset object  $D$  on local disk. If it is not present, the server retrieves the corresponding object from Pond and stores it locally. The dataset object contains the list of indexes (i.e., index objects/files) associated with the dataset. The server then retrieves the Pond objects corresponding to those indices and stores them on local disk as index files for the R-tree index [18]. This is followed by an index lookup on each of these indices. The objects corresponding to the chunks intersecting the query are retrieved from Pond by the server, and results are sent back to the client using the Tapestry [16] layer in Pond, i.e., result data is routed back to the client as a set of chunks via

Tapestry's routing support. Since any node can perform an index lookup and retrieve data in Pond, client applications can send their multi-dimensional queries to any of these nodes. We have investigated several strategies, one static and two dynamic scheduling schemes, in our implementation. The objective of dynamic query scheduling strategies is to route the queries to appropriate nodes in the system so that the workloads on the nodes are evenly balanced at all times and the queries are answered as quickly as possible. The first dynamic strategy uses a centralized server-side entity, while the second dynamic strategy adopts a client-side distributed approach.

**Static Partitioning (Static):** This strategy evenly distributes clients among servers (both master and slave servers). Each client is assigned to a server statically and submits all its queries to that server. Each query is answered by the server to which the client is assigned.

**Resource Manager:** Every server, master and slave, registers itself with the resource manager. The resource manager keeps track of the load on each server. We have implemented two variations of this strategy. In the *Resource Manager Server Only (RMS)* strategy, clients direct all their queries to any one server. When a server receives a query, it notifies the resource manager so that the resource manager increments the count of queries being handled by the server. If the server detects that the number of queries exceeds the excess load threshold, the server sends a request to the resource manager for the least loaded server. Once the least loaded server is known, the server forwards the excess queries to it. It then notifies the resource manager to decrement the count of queries assigned to this server accordingly. In the *Resource Manager Client-Server (RMS+RMC)* strategy, a client first contacts the resource manager to find out the least loaded server and sends its queries to that server. Once the query is received by a server, the RMS strategy is employed, if a server becomes overloaded. The goal of using the RMS strategy in addition to client's contacting the resource manager is to allow servers to dynamically change their threshold for excess load or apply a different method to compute their load (instead of just counting the number of queries waiting in the queued).

**Client-side Dynamic Selection (Client-Dynamic):** The previous strategies use the number of queries assigned to a server as the criterion for choosing a server to delegate queries. However, that may not always yield a good result. For instance, if the server chosen is faraway from the client, the advantage gained by servicing the queries as early as possible is lost due to the time it takes to route the data back to the client across a large distance. The client-dynamic strategy uses the query response time as the criterion for choosing who executes queries. The client initially sends a small 'test' query to each server. It records the response times from each server. The next query is sent to the server

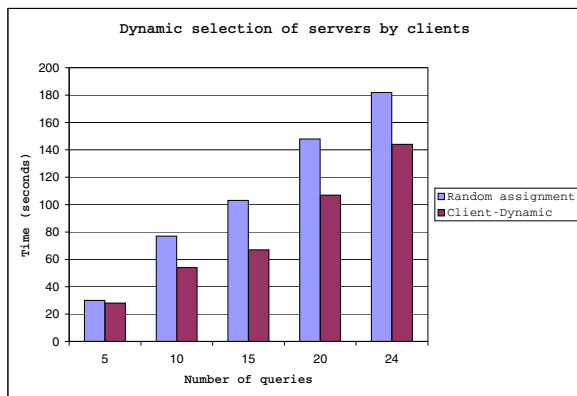
that provides best results for this client. The client updates its records every time it receives a response. If a previously 'fast' server yields poor response times, then this may be reflective of the increased load on the server, or increased traffic in the network. The client-side dynamic selection strategy is most beneficial when a client will submit many queries to the data server system over the client's lifetime or when there is a proxy located near a group of clients, which handles submission of queries to the system and routing of results back to clients on behalf of the group of clients.

Another client-side strategy we implemented is the *Random Server Selection* strategy. In this strategy, the client randomly chooses a server and sends its query to that server. This is a simple strategy that does not require keeping track of response times or the number of queries assigned to servers. By randomizing selection of servers, this strategy tries to balance distribution of queries to servers and at the same time, to reduce the chances of two clients submitting queries to the same server concurrently.

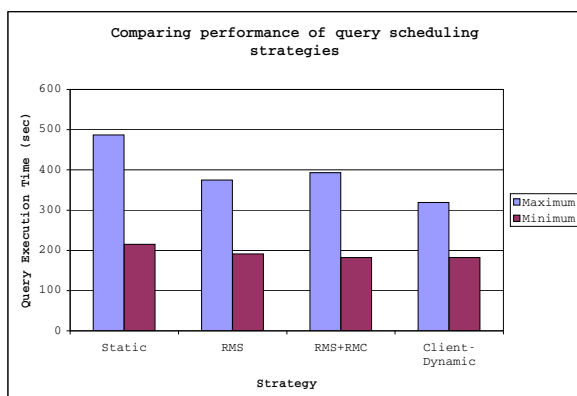
## 5 RESULTS

For the experimental evaluation, we used a heterogeneous platform consisting of two inter-connected clusters. The first cluster, **OSUMED**, is made up of 24 Linux PCs. Each node has a PIII 933MHz CPU, 512 MB main memory, and three 100 GB IDE disks. The nodes in this cluster are inter-connected via Switched Fast Ethernet. The second cluster, **MOB**, contains 8 nodes. Each node of the cluster consists of two AMD Opteron processors with 8 GB of memory, 1.5 TB of disk storage in RAID 5 SATA disk arrays. The nodes are connected to each other via a Gigabit switch. MOB is connected to OSUMED over a shared 100 Mbit network. For our experiments, master servers and inner ring servers were run on nodes in the MOB cluster. Slave servers and clients were executed on nodes in the OSUMED cluster. We ran experiments to evaluate the performance of the various scheduling strategies. Our main objective is to reduce the overall time taken to serve queries from all clients in the system. In the experiments, the bounding boxes of input data chunks are randomly created using uniform distribution within a 2-dimensional  $[(0.0,1.0) \times (0.0,1.0)]$  rectangle and a client requests subsets of chunks via 2-dimensional queries.

The first set of experiments compares the performance of the client-dynamic strategy to that of the client-side random server selection strategy. The experimental setup consists of a master server that creates 120 MB of data, six slave servers, and a single client that sends thirty randomly-generated queries. In the client-dynamic strategy, the client first sends 'test' queries, each of which returns a single data chunk, to all the servers in the system, and uses the observed bandwidth to rank the servers initially. In our experiments, the size of the data chunk returned as response to a test



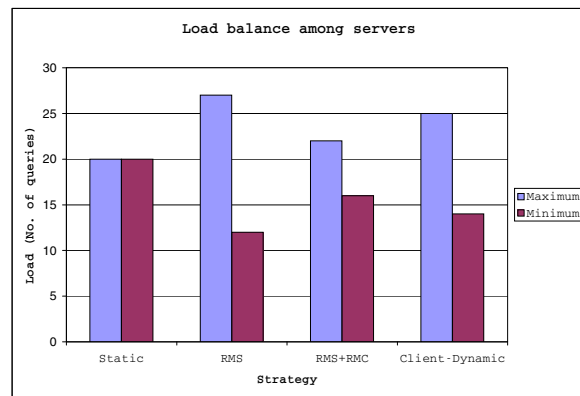
**Figure 1. Performance comparison of the client-dynamic strategy vs. random assignment of queries to servers.**



**Figure 2. Performance comparison of query scheduling strategies.**

query is 512KB. The timing results in the graphs do not include the time for the test queries, which are submitted only once when a client joins the system. In the current implementation, the client waits for all responses to come back, so that it can build its local table of response times for each server. We observed that the overhead from a test query is about 6 seconds in our experimental configuration. Figure 1 gives us the breakup of times after every 5 queries are executed. We observed that random assignment achieved a more balanced distribution of load in terms of the number of queries assigned to servers. However, in terms of response time, the client-dynamic strategy performs better since it takes into account bandwidth between the client and a server.

The second set of experiments compare the performance of different query scheduling strategies. In these experiments, there is one master server, 6 slave servers, and 12 clients. Each client submits a total of 10 randomly generated queries one by one to the system. The master server stores 250 data chunks, each of which is 512KB in size, in

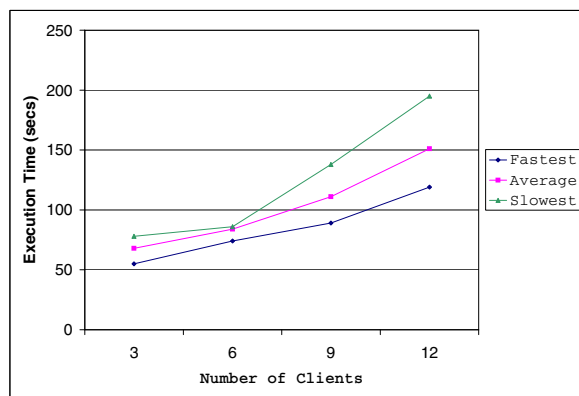


**Figure 3. Distribution of queries across servers with different scheduling strategies.**

Pond. Figure 2 shows the maximum and minimum response times observed by the clients when 10 queries from each client are answered. As is seen from the figure, the client-dynamic strategy achieves a better performance among all the strategies. The static strategy aims to achieve load balance across servers by evenly partitioning clients among servers. The RMS and RMS+RMC targets load balance at the query level and assigns queries to least loaded servers. The client-side client-dynamic strategy, on the other hand, aims to minimize query response time as observed by the client. Our results show that the metric used by the client-dynamic strategy is a better approximation of load distribution and network overhead. Figure 3 shows the maximum and minimum number of queries handled by servers in the system. In the dynamic strategies, unlike the static strategy or the random server selection strategy, a particular server may yield fast query execution times and response times for the client, and as a result, may have more queries directed toward it.

In order to look at the variation in execution time when the number of clients is varied, we carried out a set of experiments using a fixed number of 6 servers. The number of clients was 3, 6, 9, and 12. Figure 4 shows the total execution times for different number of clients. In these experiments, we used the client-dynamic strategy. The execution times in the graph are the response times observed by the client for 10 queries. *Fastest*, *Average*, and *Slowest* denote the minimum, average, and maximum response times among clients. As expected the execution time rises as the number of clients is increased, since more clients have to be served by the given set of servers. We observe that the increase in the execution time is linearly proportional to the number of clients.

We also conducted a set of experiments to measure the relative performance of the various strategies for slicing queries using the same configurations as in our original experiments. Randomly generated slicing queries that ex-



**Figure 4. Change in the execution time of queries when the number of clients is varied.**

tracted row-wise and column-wise thin slices were submitted to the system by 12 clients. Each client submitted 10 queries. We observed performance trends between algorithms which were similar to those in our original range query experiments.

## 6 CONCLUSIONS

In this paper we presented the implementation of a data server for multi-dimensional datasets. A salient feature of this server system is that it uses a distributed object store as its underlying storage substrate. This allows the server system to leverage data replication and location-independent routing of messages that are provided by the underlying substrate. We also described several query scheduling strategies to improve response times. Our preliminary experimental evaluation shows that the client-side dynamic scheduling strategy that implicitly takes both server load and network overhead into account achieves better performance than the other strategies.

## References

- [1] M. Beynon, C. Chang, U. Catalyurek, T. Kurc, A. Sussman, H. Andrade, R. Ferreira, and J. Saltz, "Processing large-scale multidimensional data in parallel and distributed environments," *Parallel Computing*, vol. 28, pp. 827–859, May 2002. Special issue on Data Intensive Computing.
- [2] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: the OceanStore Prototype," in *Proceedings of the 2nd Usenix Conference on File and Storage Technologies*, (San Francisco, CA), March 2003.
- [3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 149–160, ACM Press, 2001.
- [4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 161–172, ACM Press, 2001.
- [5] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Middleware'2001*, (Germany), November 2001.
- [6] O. Sahin and A. Gulbeden, "Sharing Multi-dimensional Data over Peer-to-Peer Networks.." <http://www.cs.ucsb.edu/gulbeden/multimed>.
- [7] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias, and K.-L. Tan, "An adaptive peer-to-peer network for distributed caching of OLAP results," in *SIGMOD Conference*, 2002.
- [8] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica, "Complex queries in DHT-based peer-to-peer networks," in *In Proceedings of IPTPS02, Cambridge, USA*, 2002.
- [9] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou, "PeerDB: A P2P-based System for Distributed Data Sharing," in *Proceedings of the 19th International Conference on Data Engineering (ICDE 2003)*, pp. 633–644, 2003.
- [10] M. Demirbas and H. Ferhatosmanoglu, "Peer-to-peer spatial queries in sensor networks," in *In 3rd IEEE International Conference on Peer-to-Peer Computing, (P2P '03). Linköping, Sweden, September 2003.*, 2003.
- [11] C. Schmidt and M. Parashar, "Enabling flexible queries with guarantees in p2p systems," *IEEE Network Computing, Special issue on Information Dissemination on the Web*, IEEE Computer Society Press, vol. 8, no. 3, pp. 19–26, 2004.
- [12] D. Kossmann, "The state of the art in distributed query processing," *ACM Computing Surveys*, vol. 32, pp. 422–469, Dec. 2000.
- [13] M. Dahlin, R. Wang, T. Anderson, and D. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *Proc. Symp. on Operating Systems Design and Implementation*, pp. 267–280, ACM Press, 1994.
- [14] M. J. Franklin, M. J. Carey, and M. Livny, "Global memory management in client-server DBMS architectures," Tech. Rep. CS-TR-1992-1094, University of Wisconsin, 1992.
- [15] L. Shrira and B. Yoder, "Trust but check: Mutable objects in untrusted cooperative caches," in *Advances in Persistent Object Systems, Proceedings of the 8th International Workshop on Persistent Object Systems (POS8)*, (Tiburon, California), pp. 29–36, Morgan Kaufmann Publishers, 1999.
- [16] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment..," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, 2004.
- [17] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of SIGMOD'84*, pp. 47–57, ACM Press, May 1984.
- [18] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized search trees for database systems," in *Proc. of 21st International Conference on Very Large Data Bases, Zurich*, pp. 562–573, September 1995.

## Biographies

**Vijay Shiv Kumar** is currently pursuing his PhD at the Department of Computer Science and Engineering at The Ohio State University. His research interests include developing techniques and middleware for efficient storage, retrieval and analysis of large datasets in distributed environments. He received his B.E. in Computer Science and M.Sc. in Chemistry from the Birla Institute of Technology and Science, Pilani, India in 2003.

**Tahsin Kurc** is an Assistant Professor in the Department of Biomedical Informatics at the Ohio State University. His research interests include runtime systems for data-intensive computing in parallel and distributed environments, and scientific visualization on parallel computers. He received his PhD in computer science from Bilkent University, Turkey, in 1997 and his B.S. in electrical and electronics engineering from Middle East Technical University, Turkey, in 1989.

**Umit Catalyurek** is an Assistant Professor in the Department of Biomedical Informatics at The Ohio State University. His research interests include graph and hypergraph partitioning algorithms, grid computing, and runtime systems and algorithms for high-performance and data-intensive computing. He received his PhD, M.S. and B.S. in Computer Engineering and Information Science from Bilkent University, Turkey, in 2000, 1994 and 1992, respectively.

**Joel Saltz** is Professor and Chair of the Department of Biomedical Informatics, Professor in the Department of Computer and Information Systems and a Senior Fellow of the Ohio Supercomputer Center. Prior to coming to Ohio State, Dr. Saltz was Professor of Pathology and Informatics in the Department of Pathology at Johns Hopkins Medical School and Professor in the Department of Computer Science at the University of Maryland. He received his M.D. and PhD in computer science from Duke University in 1985 and 1986, respectively. He earned his B.S. in mathematics and physics from University of Michigan in 1978. His research interests are in the development of systems software, databases and compilers for the management, processing and exploration of very large datasets.