

Client/Server Architecture in the ADAMS Parallel, Object-Oriented Database System

Russell F. Haddleton

John L. Pfaltz

Department of Computer Science, University of Virginia
Charlottesville, VA 22903

Abstract

This paper describes issues encountered in the design and implementation of a parallel object-oriented database system. In particular, we find that the design of a client/server interface, that is whether to use a page server or query server architecture, depends greatly on the expected application environment. We believe that the query server model is more appropriate for most scientific database applications. This paper develops the reasons for this assertion. Then we discuss the implementation of a working parallel, object-oriented, database system, called ADAMS, that has been developed at the University of Virginia. ADAMS is best known as the database system underlying the “Oracle of Bacon” website which is currently responding to over 5 million queries per year.

We show that set operations, such as union and intersection, can be performed in a completely data parallel fashion, and that implicit join queries can be performed with very few inter-processor messages in a way that scales well. All data access in the system can be completely monitored, so these observations are supported by experimental results. Finally, we compare our approach with an equivalent page server version.

Keywords: database, object-oriented, parallel, distributed, ADAMS, client/server

1. Introduction

The design of the client/server interface in object-oriented database systems has received considerable recent attention [DFMV90, KoFr95]. In creating ADAMS, a parallel object-oriented database system, the choice of an appropriate client/server architecture was a major design decision. This design has been affected by hardware considerations, application considerations, and the expected level of user activity. Considerable effort was devoted to implementing an effective client/server architecture.

ADAMS employs a query server architecture. In this paper, we explain the reasoning behind this design decision. Analytic modeling and actual performance results provide further support for our choice. But we do not argue that this architecture is best in all situations. The factors we considered for anticipated ADAMS applications drove us to our particular design.

2. Client/Server Architectures

By a page server model we mean a configuration in which one, or more, server processors directly access secondary storage at the behest of other processes which are their clients. Based on its own internal logic, a client process requests a page denoted by some persistent storage address. The server locates the page in storage and transmits it to the client which then extracts the relevant portions. A page server has no comprehension of the contents of a page, whether it consists of objects, tuples, or

perhaps code fragments. Page server architectures have been a frequent choice in object-oriented database implementations [Fran96]. The server code is very simple; and features such as page level locking and transaction rollback add little complexity. According to Carey, Franklin, and Zaharioudakis [CFZ94]: “data-shipping offloads DBMS function from the server to the client workstations, enabling the plentiful and relatively inexpensive resources of the workstations to be exploited.” Data-shipping is another name for the page server, and simple object server, architectures.

The object server model is similar, except that the client requests individual objects which the server extracts from the appropriate pages and returns. The server must know about object placement in pages. According to DeWitt, *et al.* [DFMV90], in an object server architecture object methods can be applied at either the client or server, allowing for some variation between implementations. Besides reducing the amount of data that must be shipped between the server and its clients, recovery and concurrency control can be handled at a finer granularity.

The query server model adds further server complexity. A client process sends operations, commands, or code fragments to the server along with any necessary operands. The server executes the operations and may, or may not, return the results of the operation to the client. The server not only employs object-level logic, it can exploit indices and may make optimization decisions.

An OODB implementation need not fit precisely into one of the above categories. As Wilcox [Wil94] indicates, “the variety displayed in OODB implementations is at once confusing and encouraging”. Yet whatever the label, some complexity is provided by the processor directly used by the client, and some by one or more server processors. The above labels capture this range of choices well.

How does one choose between these architectures? We contend that the choice should be governed by the nature of the intended applications. We will argue further that for many scientific database application the query server architecture is optimal.

3. Scientific Database Applications

There are at least three fundamental differences between the way scientific data and business data are processed. Commercial processing is characterized by many transactions originated by many clients that examine or update the current state of the database. In contrast, a scientific measurement, once inserted into the database, is rarely changed. We do not see the posting of small transactions by many users as the major scientific activity. Rather we expect a handful of researchers to execute fairly complex queries in search of phenomena of interest.

This introduces the second fundamental difference which concerns the usual patterns of data access. A typical commercial transaction deals with only a few data items that are well identified by an account number, a social security number, a flight number, or the like. Relatively few elements of scientific data are uniquely identified in this manner. Sometimes a scientist will be interested in a particular pixel in a particular image, but frequently he will want *all* the pixels of a specified region to serve as input to some computation, possibly statistical. Other common queries are “volumetric”; that is they denote all elements satisfying some specified range constraints. The following query expression is a good example.

```
S <- { x | p1 < x.pressure < p2      OR
        t1 < x.temperature < t2 AND
        x.instrument.elevation > 1000 }
```

Here, we are looking for all objects, *x*, whose observed *pressure* falls between the indicated limits, or whose *temperature* lies in its specified interval and the observations were recorded on a sensing *instrument* sited above 1000 meters. We will consider this example more fully in Section 4.3.

This query also illustrates the third fundamental difference; it may access thousands of data items, whereas a single transaction will normally access fewer than ten.

These three differences alone are sufficient to suggest that query servers are more appropriate for scientific work. First, the convenience of page granularity for transaction atomicity and concurrency control is much less important. Second, complex query resolution, or traversal of multi-link access paths, is more easily accomplished at a server which is “close” to the data than by a client which is well “removed” from it. And finally, exchange of very large quantities of data, whether pages or objects, between the server and client can swamp any communications network. These three considerations alone were sufficient to incline us towards a query server architecture.

We also expect the nature of scientific data to be somewhat different. The value of the object model in scientific database work is a frequent theme. Reasons for this include OODBMS support for complex data structures [MaHa94], and the adaptability of object oriented code [NRC95]. According to Bukhres and Elmagarmid [BuEl96], “the OO paradigm is particularly adept at capturing the semantics of other data models and representing it in ways that are easy to integrate.” Thus the task of scientific application design, and the further task of assimilating data from other sources into that design, can be more easily accomplished when an object-oriented approach is used.

But object oriented databases have been seen as poor performers when handling very large data bases [OzVa91]. And scientific databases can be enormous. It seems certain that high performance, scientific database implementations will have to be highly parallel. Our focus has been on shared-nothing implementations, as, according to DeWitt and Gray [DG92], “shared-memory and shared-disk do not scale well on database applications.”

These parallel object oriented database systems introduce two new considerations. First, when a system is distributed over *n* processors, it becomes likely that the aggregate of server memory and CPU resources will exceed those of active clients. In a single server system that accomodates multiple clients, a page server approach makes sense because the client, often a powerful workstation, can easily provide the cache storage to receive pages, and the cycles needed to transform these pages into objects at a rate fast enough to satisfy the user’s needs [KoFr95]. But, with multiple servers feeding only a few clients, even relatively small query results from a large database can swamp the client cache. Simple load balancing suggests that when the number of servers exceeds the number of clients, the processing load should be redistributed.

The second consequence of implementing a parallel system is that as the number, *n*, of processors increase, so to must the inter-processor message traffic. Because it does not scale as *n* becomes large, message passing inherently limits the potential of parallel systems. The key to effective parallel code is to minimize inter-process message passing. Either page servers which transfer entire pages when only a few objects will be used or

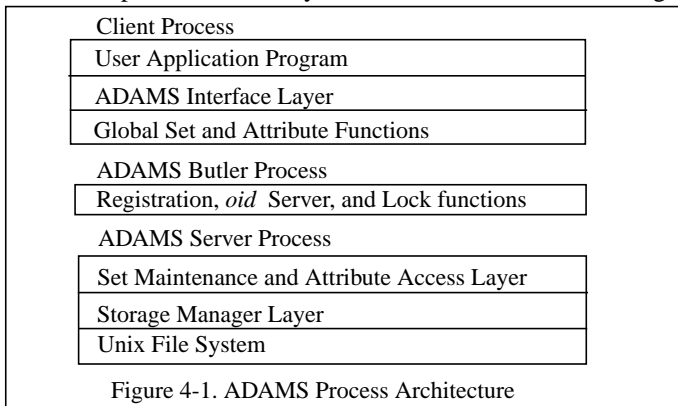
algorithms that rely on extensive inter-server data transfer can cause severe bottlenecks. Therefore we chose a query server architecture and an object distribution methodology which minimizes inter-server communication.

4. ADAMS Architectural Overview

In this section we flesh out some of the more important details associated with the ADAMS implementation. ADAMS [PfFr93] was initially designed to support highly concurrent, scientific computation [GPFS90] on shared nothing, message passing processors. The performance reported in this paper has been observed in a system running on a cluster of 8 SPARC2 processors each with an attached Seagate 1.2 GB disk and Ethernet connection network. It is object-oriented, with user defined data classes. Each instantiated object within one of these classes is assigned a persistent, unique object identifier, or *oid*. These *oids* serve as object surrogates throughout the system and play a prominent role in the way we distribute objects and minimize inter-processor communication.

4.1 ADAMS Client/Server Details

An ADAMS server is essentially a query server. It executes set and index level operations. But at times it also functions as a very fine-grained data or object server in that it can honor requests not for entire objects, but individual attribute values of objects. This handles an object server problem mentioned by DeWitt, *et al.*: “since the software on a workstation simply requests an object from the server without generally being aware of the size of the object, large, multipage objects may move in their entirety, even if the application needed to access only a few bytes” [DFMV90]. Both client and server processes have been implemented in a layered fashion, as illustrated in Figure 4-1.



The user application program of the client process is written in some host language, such as C, C++, or Fortran, with embedded ADAMS statements. These ADAMS statements are translated by a preprocessor into appropriate function calls to ADAMS runtime routines residing in the ADAMS interface layer. The database operands of these runtime procedures are always denoted by object *oid*. But, neither the user, nor the ADAMS preprocessor, need be aware of the way sets of objects, and their attributes may have been distributed. We have deliberately created a seamless, parallel

implementation. “Global” *oid*'s denoting sets and attributes that are distributed throughout the system are decomposed into their n distributed components by the ADAMS runtime routines within the client process.

The set maintenance and attribute access level of the servers constitutes the heart of the ADAMS system. ADAMS employs a “decomposed storage model”, or DSM [Khos93], in which the attribute values of an object are stored as (*oid*, *value*) pairs. All references to attribute values for a named attribute, for example “pressure” above, are stored in a B-tree associated with that attribute. Thus there is a “pressure” B-tree, indexed by *oid*. The inverses of these attribute trees, indexed by value, are created as needed. We describe how this model has been employed in a parallel form in the next section. The use of DSM makes schema evolution fast and efficient, while making the storage and access of data associated with complex or large objects a simple task.

All sets and all attributes are maintained as separate trees in order to facilitate access. This is the set maintenance and attribute access layer of Figure 4-1. It is in this level that all retrievals and set manipulations are performed. Each server maintains its own set of trees to manage the objects assigned to it. Below this level, we indicate a storage manager level which was implemented on top of the Unix file system.

In between the client and server processes we have indicated a “Butler” process. It is a single, well-known system process. When a new client enters the system, it must be connected to the appropriate distributed servers; we call this “registration”. From then on, there is a direct connection between the client and its assigned servers with no intervention by the butler. The butler process also allocates new *oid*'s as needed and oversees concurrency control. Further implementation details are available [Hadd95].

One secret for minimizing message traffic lies in the nature of the messages themselves. For example, in sequences of low granularity operations, such as set insertions or object attribute assignments, we are frequently able to bundle many operations for a particular server into a single message. As these messages do not require a response from the servers, the message burden is kept low when loading large quantities of data, such as that from a remote sensor stream.

4.2 Partitioning by *oid*

Processing very large sets of scientific data requires the distribution of these sets over multiple disks and processors. It seems to us that a horizontal partitioning of sets by *oid* is the natural way to address what for some systems has been seen as a difficult issue [GhDe90]. We have found several addressing schemes in the literature that employ a simple mapping between object identifier and partition, although only a few claims of direct database support for data parallel set operations have been encountered. The SHORE system is one such example, providing support for a parallel structure called the ParSet [DNSV94].

We use the *oid* of an object to uniquely determine the partition P_j , that is the processor and storage, on which the object, and all its attribute values, resides. We use the low order bits to implement a modulo operator which partitions every *oid* to a unique P_j . A hash function could work, as well. A set is an object. Based on its *oid*, it is assigned to a partition, P_j . But, the members of a set are objects which may have been assigned to many different partitions. Consequently, each set, A , is also subdivided into

n subsets, A_1, \dots, A_n , where each subset, residing on a different node in the system, is comprised of just the objects assigned to partition, P_j . If a particular object x is a member of two ADAMS sets, say $x \in A$ and $x \in B$, it will always be represented as a member in two subsets on the same partition. That is, if $x \in B_j$, $1 \leq j \leq n$, and $x \in A$, then $x \in A_j$, as shown in Figure 4-2.

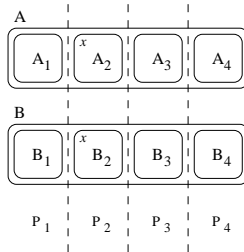


Figure 4-2. Distribution of sets A and B

Using such a scheme, large granularity set operations, such as union, intersection, and difference, can be performed in a data parallel manner, as can selection by attribute value, as in $p1 < x.pressure < p2$. ADAMS makes abundant use of these data parallel set operators.

But, an essential characteristic of the object-oriented model is that attributes may be “object valued”, that is reference another object. The term $x.instrument.elevation < 1000$ is representative. The attribute *instrument* denotes a sensor object y with *elevation* attribute greater than 1000 meters on which the observation x was made. Queries involving object valued attributes are somewhat more complex to process because the object y may very well have been assigned to a different partition than x . The presence of object valued attributes, or implicit joins, forces inter-server communication. It is essential to reduce their impact on performance.

4.3 Query Processing Details

We consider queries such as

```
S ← { x | p1 < x.pressure < p2      OR
      t1 < x.temperature < t2    AND
      x.instrument.elevation > 1000 }
```

that we had seen in Section 3. The logical connectives, *and* and *or*, can be naturally implemented as set operators, *intersect* and *union*. Consequently the approach we take is to find all x satisfying the first term, $p1 < x.pressure < p2$, and all x satisfying the second term $t1 < x.temperature < t2$. These two sets are unioned. This will then be intersected with the last term.

Parallel selection with respect to an attribute is well understood; each processor searches the indexes for the objects on its partition that satisfy the criteria. It is a completely data parallel operation, with no cross talk between the processors. The conjunct which involves an object valued attribute is a bit different.

Like all attributes, an object valued attribute is a single valued function. But, when applied to an object, the attribute function returns a reference to another object instead of a value. Also like any other attribute, the system indexes these object valued attributes using the referenced *oids* as the key values and the referencing *oids* as the entries. When

partitioning data by *oid*, we partition the attribute index trees by the referenced *oids*. That means that if object oid_k on P_k references oid_j on P_j , a selection in P_j will retrieve oid_k .

To process the term $x.instrument.elevation > 1000$, the system first selects all sensor objects y with $elevation > 1000$. In our database there were approximately 60 of these on each partition. In an iterative loop over each object, actually each *oid* y , all the measurements x are retrieved that have that *oid* as instrument value. Let X_j denote all x such that $x.instrument = y_j$, where y_j is a sensor object on partition P_j . These *oid*'s comprising X_j must now be distributed to their proper partitions. They are separated into batches, e.g. all oid_k bound for P_k , and then transmitted¹ to their appropriate processors. Server P_j in turn receives its subsets from the other servers, assembles them all into a single set, which it then intersects with the result of the first two terms.

Table 4-4 summarizes the data accesses involved in executing the query in a million object database. Observe that there are many more random, than sequential, disk reads.

	P ₁	P ₂	P ₃	P ₄	Sum
p1 < pressure < p2	7,466	7,499	7,503	7,439	29,907
t1 < temperature < t2	47,631	47,657	47,371	47,591	190,250
y.elevation > 1000	61	71	70	49	251
result	19,179	19,079	19,120	19,052	76,430
random disk reads	1,872	1,868	1,800	1,673	7,213
random disk writes	25	26	25	26	102
sequential disk reads	643	373	268	265	1,549
sequential disk writes	1,173	1,167	1,161	1,173	4,674
data messages sent	54	65	63	45	227
data messages received	57	54	55	61	227

Table 4-3. Data accesses in the query statement

When an inverse attribute index tree is created, it is almost impossible to cluster the nodes; so virtually all disk access requires random head movement. In this query, nearly all of the sequential disk operations arise from the sorting of *oids*. To facilitate the set operations, all sets are kept in lexicographic *oid* order. But, the *oids* returned from selection over an attribute are not in sort order; they must be sorted. With a bit of cache management, described in [Hadd97], we can sort surprisingly large sets completely in cache; but, for safety, we always sequentially flush the entire cache to disk on completion. Only occasionally must portions of these sets be sequentially re-read from disk.

Note that, although 76,430 observation objects were eventually retrieved from 1,000,000 observations, most of the processing was data parallel. At most 65 inter-server data messages were sent by any server. In addition the client sent control messages initiating the query and there were a handful of short control messages between servers that coordinated the data exchange. We have been quite successful in minimizing inter-processor message traffic.

5. Performance

1. In ADAMS, each data message can contain up to 1,000 *oids*

Because the set operators are completely data parallel, they should exhibit linear speed up, which is defined t_n (task of size k) = t_1 (task of size k) / n where t_n denotes execution time in an n processor system. In our work with ADAMS they have. The time to execute a specific task on n processors is one n^{th} the single processor time. The complex query, on the other hand, exhibits *super-linear* speed up, as shown in Figure 5-1, where theoretical linear speed up is indicated by the dashed line.

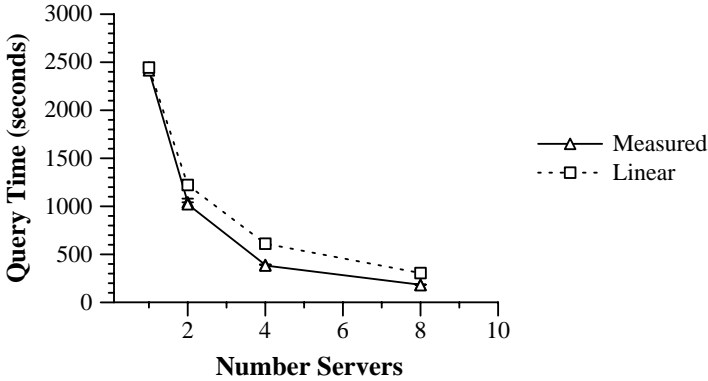


Figure 5-1. Super-linear speed up of volumetric query

It is not hard to demonstrate that asymptotically, such a query cannot exhibit linear speed up because (1) its inter-server message traffic must increase as the number, n , of processors increases, (2) the necessary sort operations are $O(n \lg n)$, and (3) the analytic cost expressions contain constants that do not scale at all [Hadd97]. Nevertheless, on such small sets as these, we get super-linear behavior because both searching inverse attribute trees and sorting are very cache dependent and when we double the number of processors, we are more importantly doubling the available cache. For these reasons, scale up is regarded as a much more accurate measure of parallel performance, even if it is harder to derive.

A system exhibits linear scale up if t_n (task of size $n*k$) = t_1 (task of size k), that is, if the number of processors is doubled the size of the task is doubled. The problem

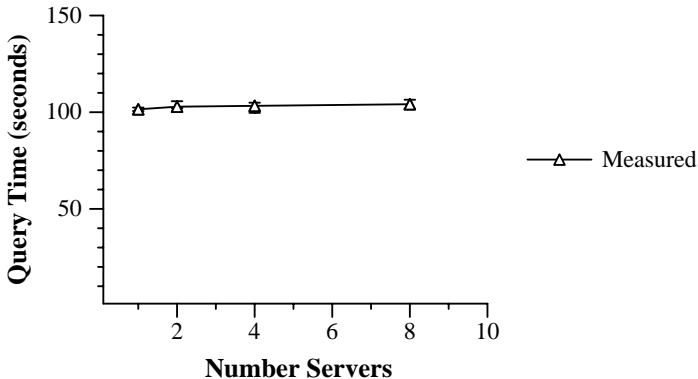


Figure 5-2. Scale up of the query

is that to fairly double the size of a task, one must really double the size of the database

in which it is embedded as well. This is possible only with experimental databases such as ours. We created databases of 125000, 250000, 500000, and 1000000 observation objects respectively and ran the same queries. Each query, or operator, was repeated 10 times, always using a “cold” system. Figure 5-2 illustrates the scale up performance of the query. It is not quite linear, although the rise is almost imperceptible. We believe this kind of performance is due to our success in minimizing inter-processor message traffic, between client and servers and between the servers themselves.

6. Conclusions

In a scientific environment in which complex volumetric queries are the norm, a query server architecture seems desirable. We have demonstrated that this architecture scales well in our parallel implementations.

We wish to make a comparison with an equivalent page server architecture. This is a bit difficult since, in a real page server system, a different algorithmic approach may be adopted. Nevertheless some rough comparisons can be made.

The objects in our test database were approximately 100 bytes in length.² So, given our 8K pages, we can expect about 81 objects per page, or approximately 12,345 pages in all. The retrieved set of 76,430 objects in our running query comprises about 0.076 of the entire million object database. There is no reason to expect any significant clustering of these retrieved objects, so if we assume a reasonably uniform distribution of these objects there will be approximately 6.1 desired objects per page. Just to return the results of this query in the form of pages will require transmission of nearly all 12,000 pages! This is why in Section 3, we indicated that the sheer size of query results in scientific database applications constituted a reason for considering a different client/server architecture.

Orenstein, *et al.* describe query processing in a page server system. They indicate that “... queries execute on the client. This does not mean that the entire contents of a collection, or index, is sent to the client in order to evaluate a query. As with other objects, only those pages containing referenced addresses are fetched from the server” [OHMS92]. Just transmitting the relevant index pages to the client in a sequential fashion so it can conduct the search is expensive. Our running query required search on three attributes. This is why, in Section 3, we asserted that selections should be conducted in the server “close” to the data.

We have shown how, in the case of the ADAMS object oriented database system, the choice of scientific database as an application area has resulted in a number of design and implementation decisions reflected throughout the system. It is possible that focussing on other application areas would have yielded different design choices. Yet the current design has been effective, as supported by a wide range of performance tests.

7. Bibliography

[BuEl96] Omran A. Bukhres and Ahmed K. Elmagarmid, *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*, Prentice Hall, 1996.

2. Two string attributes, *observer* and *description*, account for the variability.

- [CFZ94] Michael J. Carey, Michael J. Franklin, and Markos Zaharioudakis, "Fine-Grained Sharing in a Page Server OODBMS", *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, 1994.
- [DFMV90] David J. DeWitt, Philippe Futersack, David Maier, and Fernando Velez. "A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems", *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, 1990, pp. 107-121.
- [DG92] David DeWitt and Jim Gray. *Parallel Database Systems: The Future of High Performance Database Systems*, *CACM*, 35(6):85-98, June 1992.
- [DNSV94] David J. DeWitt, Jeffrey F. Naughton, John C. Shafer, and Shivakumar Venkataraman. "ParSets for Parallelizing OODBMS Traversals: Implementation and Performance", *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, Austin, TX 1994 Texas, 1994.
- [Fran96] Michael J. Franklin, *Client Data Caching: A Foundation for High Performance Object Data Systems*, Kluwer Academic Publishers. Boston, MA. 1996
- [GhDe90] Shahram Ghandeharizadeh and David J. DeWitt. "Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines", *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, 1990, pp. 107-121.
- [GPFS90] Andrew S. Grimshaw, John L. Pfaltz, James C. French, Sang H. Son, Exploiting Coarse Grained Parallelism in Database Applications, PARBASE-90 Intern'l Conf. on Databases, Parallel Architectures and their Applications, Miami Beach, FL, 1990, pp. 510-512.
- [Hadd95] Russell F. Haddleton, An Implementation of a Parallel Object Oriented Database System, Technical Report CS-95-49, Department of Computer Science, University of Virginia, 1995.
- [Hadd97] Russell F. Haddleton, Parallel Set Operations in Complex Object-Oriented Queries, Ph.D. Dissertation, Univ. of Virginia, Aug. 1997.
- [Khos93] Setrag Khoshafian, *Object Oriented Databases*, John Wiley & Sons, New York, 1993.
- [KoFr95] Donald Kossman and Michael J. Franklin, "A Study of Query Execution Strategies for Client-Server Database Systems", Technical Report CS-TR-3512 and UMIACS-TR-95-85, Department of Computer science and UMIACS, University of Maryland, August 1995.
- [MaHa94] David Maier and David M. Hansen, "Bambi Meets Godzilla: Object Databases for Scientific Computing", *Proceedings of the Seventh International Working Conference on Scientific and Statistical Database Management*, Charlottesville, VA 1994, pp. 176-184.
- [NRC95] Committee for a Pilot Study on Database Interfaces, Finding the forest in the trees: the challenge of combining diverse environmental data: selected case studies, National Academy Press, Washington, DC, 1995.
- [OHMS92] Jack Orenstein, Sam Haradhvala, Benson Margulies, Don Sakahara. Query Processing in the ObjectStore Database System, Proc. 1992 ACM SIGMOD Conf., San Diego, CA, June 1992.
- [OzVa91] M. Tamer Ozsu and Patrick Valduriez, *Principles of Distributed Database Systems*, Prentice Hall, Englewood Cliffs, NY. 1991.
- [PpFr93] John Pfaltz and James French, Scientific Database Management with ADAMS, *Data Engineering*, 16, 1 March 1993, pp. 14-18.
- [Wilc94] Jonathan Wilcox, Object Databases: Object methods in distributed computing, *Dr. Dobbs's Journal*, #222, November, 1994. pp. 26-32.