An Implementation of a Parallel Object Oriented Database System

Russell F. Haddleton

CS-95-49
December 20, 1995

Computer Science Department
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22901

This research was supported in part by
DOE Grant # 95ER25254

An Implementation of a Parallel Object Oriented Database System

Technical Report CS-95-49, Computer Science Department,
University of Virginia

Abstract

In the following document we describe the current implementation of ADAMS, a parallel object oriented database system developed at the University of Virginia. The parallel data structures employed by ADAMS are discussed, as is the client/server architecture. We list a number of sources of parallel speed-up found in typical ADAMS programs, and explain how these opportunities are exploited. Several potential future research projects related to this work are given.

1. Introduction

This paper describes the implementation and use of ADAMS, a multi-user parallel object oriented database system developed at the University of Virginia. While students at U.Va have worked with a version of ADAMS in undergraduate and graduate database courses, the version employed in those courses has been a single-user version employing several of the features which will be discussed in this document. It is important to note that programs developed and debugged with the single-user version of ADAMS can be run using the parallel version without source code changes. Further references to ADAMS in this document will mean the parallel version.

ADAMS can mean different things to different people. An ADAMS programmer would see ADAMS as an object-oriented/functional database language to embed in his/her C++ or Fortran programs. A systems administrator would see ADAMS as a group of server processes running on his systems. A parallel programming authority would see ADAMS as an application running on top of Mentat, a parallel programming system developed by a related research group at U. Va. An expert in data structures would see ADAMS as a vehicle for O-trees, the storage structure employed by ADAMS.

The ADAMS system is an aggregate of design choices made to tackle the challenges of maintaining very large quantities of complex data in a user-friendly manner. In particular, we are interested in supporting scientific applications, which are often responsible for very large quantities of data while having the conflicting task of managing complex and flexible object structures. We see these application environments as write once/read many, where providing an ADAMS user with the means to quickly isolate data of interest from a vast collection is of primary importance. Our query processing goals and ADAMS query processing methods are discussed further in [HaPf96].

This paper focuses on ADAMS parallel features. Other documents discuss the ADAMS language [Pfal95] and the theory behind O-trees [OrPf88]. Future documents may discuss the O-tree implementation, the ADAMS storage manager, and stream processing within ADAMS.

Recent work has centered on the exploitation of large grained data parallelism in set operations over a partitioned database. A secondary theme is the creation of large

granularity operations (from smaller ones) whenever possible, treating long sequences of set insertions, attribute assignments, and similar ADAMS operations as single data parallel operations. To this we have added aggressive pre-fetching strategies for iterative processing. All of this is within the context of providing a sophisticated object oriented database language to the user.

2. Architectural Overview

In a parallel database system there are a number of tasks to be performed and a number of processes available to perform them. In many systems these processes are divided into clients (each handling the vagaries of a user program and interactions with a specific user) and servers (providing a fixed set of operations in a specified manner, generally to multiple clients but also to other servers in some cases). In many cases each processor in a system configuration is either dedicated to a single server process or to one or more client processes. The design of the interface between client and server has great implications for performance, and as has been seen in this research different designs are better adapted to different workloads.

The server designs we have seen have fallen into three categories: page server, object server, and query server (we include file server in the page server category). In page server systems, the server only knows about pages of storage. The server can be asked to retrieve a page of data from a location, and can be asked to store a page at a location. Object server systems employ a more complex server, the server knows about objects or clusters of objects and can retrieve and store them. Query servers are a further step up, they can be asked to perform complex queries over collections of objects. Useful references in this area include [DFMV90], [ChWi93], and [Vers93] (which describes one of the few object oriented query server systems).

The bottleneck limiting database system performance has traditionally been disk I/O. The main purpose of parallel server systems has been to address this problem. A difficulty we have seen is that while the use of parallel servers removes the disk I/O burden from the client processor, it replaces it with a significant message processing load. This message processing cost at the client has dominated performance in some of our testing. A

concern has been that with a page server architecture, which is prevalent in commercial object oriented systems, the message burden on the client would become more of an issue as the number of servers increased.

Some of the justifications for a page server architecture include the fact that in a configuration serving many client processors the message burden on any single client processor is likely to be small. A further justification is that the client processors are generally powerful workstations while the database servers often have substantially less capacity. Both of these are true for many system configurations and many application environments. But our anticipated workload consists of only a few clients running large queries, and the database servers we have employed have had cpu cycles to spare (disk I/O and occasionally client processing costs dominating performance). For that reason one of our goals has been to move the processing burden to the servers. To accomplish this we have adopted a query server architecture .

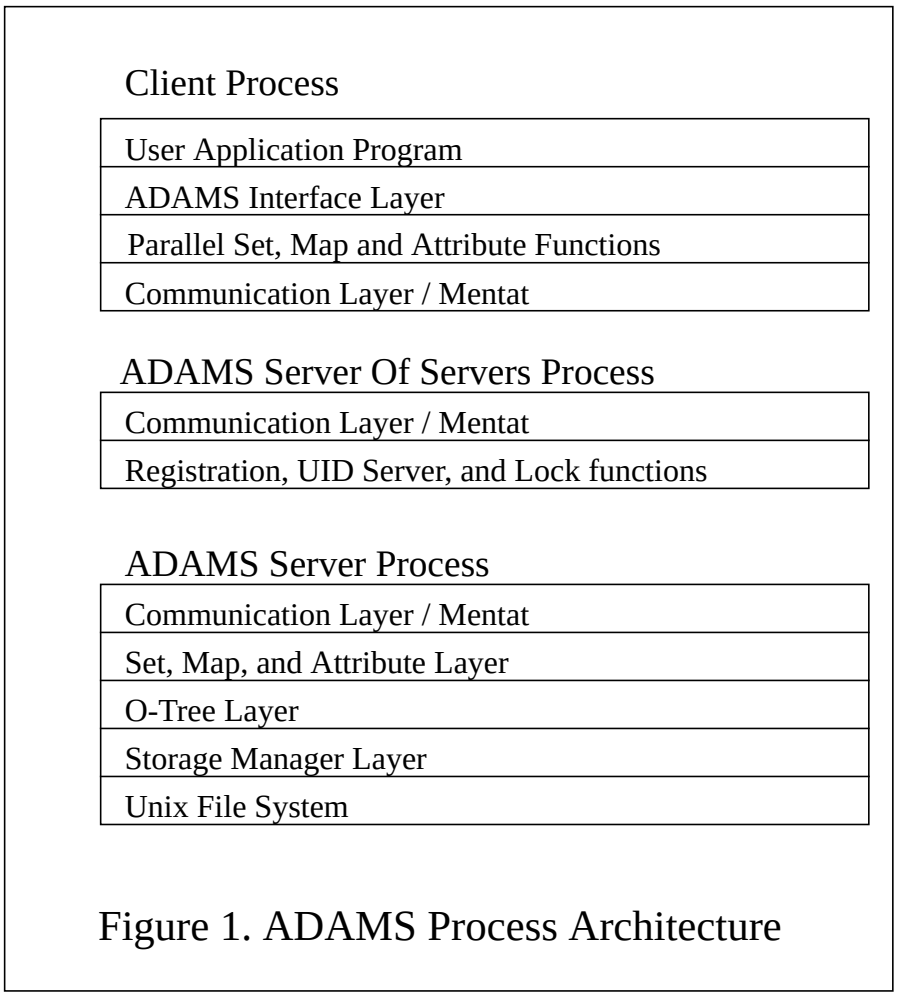

Figure 1. ADAMS Process Architecture

A running ADAMS program involves at least three executing processes: the process executing the client side of the application, a Server Of Servers process which the client process contacts to receive references to the appropriate Server process(es), and one or more ADAMS Server processes (Figure 1).

2.1 Client Side - Application Program

An ADAMS application program initially consists of a program written in C++ with embedded ADAMS statements. The application code contains operations such as "`insert X into S1`", or "`S3 <- {X | X in S1 or X in S2}`" or "`| text_buffer char * | <- X.name`", where `X` is an object, `S1`, `S2`, and `S3` are sets, `name` is an attribute, and "`<-`" is the ADAMS assignment operator. This program is compiled by the ADAMS preprocessor, which translates named references (such as `S1` or `name`) into *uids* (unique identifiers, also called *oids* or object identifiers in the literature) and ADAMS code into calls to ADAMS interface routines (see Appendix A).

ADAMS data structures include sets, maps (inter-object references), and attributes. Our storage model is similar to the Decomposition Storage Model discussed by Copeland and Khoshafian [CoKh85]. Rather than store an object's attribute and map values contiguously, all values for an attribute or map are stored in a (parallel) tree structure associated with the name of the attribute or map. These tree structures are indexed by object identifier. We partition our database by object identifier, thus all the attribute and map values of a particular object exist in the same partition but in different trees.

```
class _A_para_uid
{
   public:
          _A_uid          self;
          _A_NDX_TYPE     type;
          _A_uid          sub_uids[MAX_NODES];
          _A_uid          storage_uid;
          int             Persistent;
          int             has_inverse;
          _A_para_uid(_A_NDX_TYPE new_type);
};
```

Figure 2. The Para_uid C++ Class Definition

At run time, for each *uid* representing a parallel structure (a set, map, or attribute) a *para_uid* (Figure 2) is employed. The client process maintains a list of the parallel set, map, and attribute *para_uids* it accesses as it executes. Each *para_uid* structure keeps track of the N sets, maps, or attributes which make up a parallel object in an N-partitioned database.

For each operation requested by the program, the client process will look up the *para_uid* of the parallel sets, maps, and attributes referenced and send the operation to the ADAMS Server(s) handling the relevant partition(s) of the database. In an N-partitioned database, a parallel set union operation would require the notification of all N ADAMS Servers, while for the insertion of a single object into a parallel set only one server would need to be notified.

If the *para_uid* representing the parallel set, map, or attribute referenced is not found in the client processes list, the client will request it from the appropriate ADAMS Server (hashing the parallel objects *uid* to determine which server is responsible for storing the *para_uid*). If the requested *para_uid* is not found by the ADAMS Server, the ADAMS Server is responsible for creating a new one and returning it.

As our database is partitioned by object identifier a number of operations involved in query processing, such as set intersection, union, difference, and range search over an attribute, can be performed in a completely data parallel manner. For example, the client process executes an intersection operation by sending a message to each ADAMS Server containing the the intersection instruction and the appropriate *sub_uids* (from the three involved *para_uids),* and the ADAMS Servers complete the operation without further interaction with the client process (and without any interaction with each other). Employing data parallel techniques when possible was an early priority in the design of ADAMS, as a query consisting only of data parallel operations would be very likely to exhibit good parallel performance, and it was expected that a design allowing excessive message traffic would doom large system configurations.

Many of the instructions to be sent to the ADAMS Server(s) require no response, as with set insertion or map entry deletion. As long as these no-response operations are executed prior to a further use of the set or map (which would require their execution prior to logically following operations for program correctness), the client need not incur the expense of a message send for each tiny operation. To exploit this situation, the client

process maintains a buffer of outgoing instructions for each database partition. When a buffer fills, or an instruction requiring some response must be sent, the buffer is shipped to the appropriate ADAMS server, which unpacks the buffer and processes the operations as if they had been sent one at a time. Adding this instruction buffering capability quartered the time it took to populate a large database.

In other cases a response is required, but from all the ADAMS Servers at once. Requests for set cardinality, for example, are made in a loop. The client does not begin waiting for the first response until all requests have been sent. This same looping is done with first/next operations over sets. The expectation is that in these cases much of the work required is being done simultaneously by multiple servers.

We've seen a larger opportunity to exploit parallelism and reduce message overhead when performing these operations. First/next operations generally occur within "for_each" ADAMS program constructs. Within such a loop it is usually true than a number of attribute (or map) values are requested for each element. Previously each of the return attribute value operations executed required a response from an ADAMS server before further processing by the client was possible, as there may have been user code which branched or took other action depending on the value returned. We accelerate this processing by keeping track of the attributes requested for the "first" element and requesting that the ADAMS server provide these values automatically when we perform "next" operations within the loop. And when we request a next element from the servers we actually request the next 10 elements from each server.

When performing a attribute value retrieval (or assignment, for coherency), the client code checks a "next element" write through cache to determine if the value needed is already present, and if so (on retrieval) proceeds without contacting an ADAMS Server and incurring a lengthy delay. Despite the cache overhead, the attribute value/set element pre-fetching strategy as implemented greatly speeds the "sequential" processing done after many queries.

While data parallel execution of many query operations helps to ensure good parallel query performance, the iterative looping, message buffer and pre-fetching strategies discussed above have helped to ensure that adding data to a database and examining the contents of a retrieved dataset also benefit from parallelism.

2.2 Server Side - ADAMS Server

As mentioned earlier in this section, ADAMS servers are query servers. They currently accept approximately 80 different instructions (see Appendix B). Many of these instructions are frequently encountered by a server as part of a block of instructions, saving message sending costs at the client and receiving costs at the server. Each server can accept instructions from several users, and processes them in the order in which they are received. Note that Mentat guarantees that instructions from one client to one server will be received by the server in the order that they were sent by the client.

Each ADAMS server maintains a list of the maps, sets, and attributes it accesses as it executes (similar to the list the client maintains, only in this case these entries correspond to the *sub_uids* in the *para_uids* maintained in the client list). If an operation requires a set, map, or attribute not in the list then a persistent tree structure known as the "header map" is consulted to resolve the reference.

Aside from providing tree/index operations on these sets, maps, and attributes, the stream processing associated with ADAMS queries, and persistent storage for *para_uids* used by the client, ADAMS Servers handle most of the complex processing involving inverse maps (which involves inter-server transfer of *uids*, generally as part of a query involving inter-object references).

An inverse map ADAMS Server operation of particular importance in query processing specifies a set of items and the name of an inverse map. The result is to be a set comprised of the inverse map applied to the set passed in. The difficulty is that this result set consists of elements that are unlikely to belong in the partition, as the links between objects do not respect partition boundaries. Each ADAMS Server can communicate directly with other ADAMS Servers in the ADAMS configuration, and with very minor coordination provided by the client process the result sets are automatically re-partitioned.

The instructions sent by the client to the ADAMS servers take the database from one "file consistent" state to another. The client process does not know about O-trees and therefore cannot damage them if terminated unexpectedly (in the middle of processing an O-tree split the database is quite vulnerable). If a client program is terminated in the middle of processing, the data in the database may not be "application consistent". But a skillful

ADAMS programmer could work with such a database (if needed), possibly rerunning his program from the beginning without major modifications.

Tests of the system verify that terminating one client program abruptly does not interfere with an ADAMS Servers communications with other clients running ADAMS programs at the same time as the termination.

2.3 Server Side - ADAMS Server Of Servers

ADAMS employs a second type of server called an ADAMS Server of Servers. The name Server of Servers lacks the clarity of meaning associated with names such as “lock manager” or “uid server” used in some systems, but the ADAMS Server of Servers has multiple functions and this design has helped to keep the implementation simple while costing nothing in performance. The ADAMS Server of Servers serves *uids* to the ADAMS Servers, thus it is a server of servers. It also provides references to ADAMS Servers when requested to by client processes, or serves servers to client processes. With two of its functions described by the name, Server of Servers was the most logical choice. The ADAMS Server of Servers has four responsibilities.

The first responsibility of the Server Of Servers is to prevent a problem associated with sequential ADAMS. When two sequential ADAMS programs access the same database, chaos in the O-tree structures or block allocation bitmaps can be the general result. The Server Of Server ensures that only one ADAMS Server is assigned to one directory/system combination.

A client process is not permitted to create ADAMS Servers. Instead it requests references to the appropriate ADAMS Server(s) from the Server Server, which maintains a list in memory of all the active ADAMS Servers in the Mentat configuration, and which directory/system combination each ADAMS server is handling. If an ADAMS Server is requested which is not in the list the Server Of Servers creates it.

The Server Of Servers keeps track of which user process is using which ADAMS Server(s), and is notified when an ADAMS program shuts down, so that it can shut down ADAMS Servers not currently being used, limiting the risk to a database from system/CPU failure.

A second responsibility of the ADAMS Server Of Servers is as a server of element *uids* to all processes in the Mentat configuration (mainly the client processes, an ADAMS Server only uses them when it creates new *para_uids*). If the Server Of Servers provided element *uids* one at a time it would be a horrible bottleneck. Instead the requestor indicates how many it needs and the Server Of Servers returns an appropriate begin *uid* and end *uid*, leaving the requestor to use the range within as needed.

A third responsibility of the Server Of Servers is to manage a lock table. While it is unclear whether the features will actually be employed generally within ADAMS, the ADAMS Server Of Servers supports *uid* level locking/unlocking/lock interrogation functionality, including the ability to unlock all locks owned by a specified user process.

The last responsibility of the Server Of Servers is to maintain information on the activities of the ADAMS Servers. Each ADAMS Server keeps track of the number and types of instructions it has processed, and periodically sends this information to the Server Of Servers. The Server of Servers can then be queried by a client program for a summary of ADAMS Server activity.

2.4 Hardware Configuration

While ADAMS can be configured to run with a number of ADAMS Servers on a single processor, all accessing different files on a single disk drive, and delivering less than spectacular performance, there are two standard modes of operation. The first is to have a single ADAMS Server maintain the database. This is useful when the datasets being manipulated are not particularly large and the primary benefit gained from employing the server is support for multi-user access. The second involves setting up multiple ADAMS servers over multiple processors, each server having it's own disk drive and processor (see Figure 3). This is appropriate for larger datasets (tens or hundreds of thousands of objects or more), where the benefits of parallel operation outweigh the message cost.

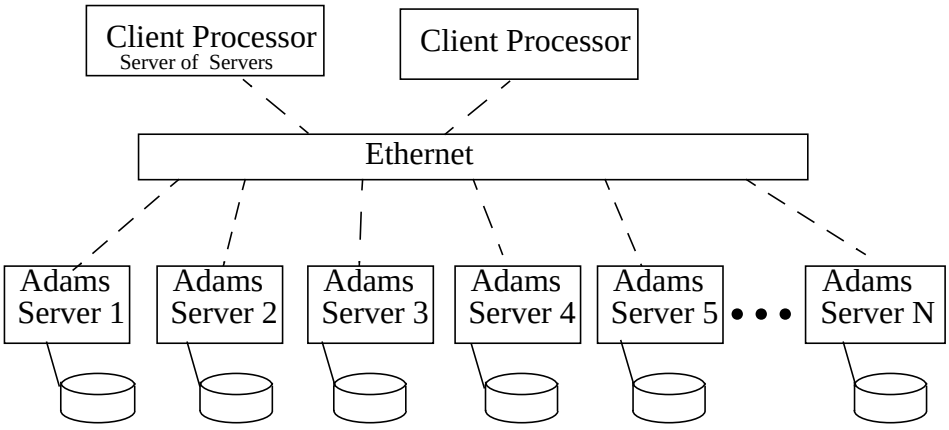

Figure 3. Typical ADAMS Hardware Configuration

3.0 Parallelism

In discussing the client side process, we have identified three potential sources of parallel execution. All three forms are exploited in this implementation.To summarize:

3.1 Data Parallelism

Data parallelism involves performing the same operation on multiple data items concurrently. With our parallel sets, for example, we can perform a union on all database nodes simultaneously on the three N subsets (two source sets and one result set in each partition) which comprise the parallel sets being unioned. No inter server communication is required for this and similar operations (intersection, difference, range search over an attribute are other examples).

3.2 No Return Parallelism

Many operations do not require a return value, only that they be completed before related operations begin. We can bundle these by destination partition (saving on message cost) and send the resulting larger granularity bundles to be completed in parallel on the N servers (without waiting for them to complete).

3.3 Loop Parallelism

Some ADAMS processing requires responses from all the ADAMS servers in an ADAMS configuration. Determining the cardinality of a parallel set is one example. Here we can send our N requests to the N ADAMS Servers, and only when the requests are sent do we wait to retrieve our responses. The loop of processing thus results in parallel execution at the ADAMS Servers (although at a low granularity). Iterating through sets (with first/next operations) is done similarly. See [McEl91] for a different form of loop parallelism explored by the ADAMS research group.

4.0 Running ADAMS

It would be ideal if the ADAMS programmer could be kept completely away from the parallel details of ADAMS. This is nearly the case. But not quite.

The ADAMS Server and Server Of Server processes are Mentat objects, and can only be run within a Mentat configuration. The ADAMS programmer will not have to program with any knowledge of writing code for Mentat, but it would be useful if he/she could determine whether the appropriate Mentat configuration was up and running. The ability to use Mentat utilities such as *list_objects* would be valuable.

There are other tools of interest as well. These include the ADAMS preprocessor, the Mentat/C++ compiler and linker (*dcci*), a set of programs to display and manipulate dictionary information, and a utility to observe the running ADAMS configuration.

4.1 Mentat

The current version of ADAMS relies upon Mentat [Ment95] for inter-process communication. The use of Mentat in parallel ADAMS is an interesting choice. Mentat is very good at compiling computationally intensive programs, creating (with some help from the programmer) a dependency graph using Mentat class instances, and then efficiently scheduling these objects based on current processor loads.

These techniques are not of great value when applied to an ADAMS program. The first difficulty is, of course, that the data dictates where an operation must be performed. If we desire full data parallel execution for set operations, and if we want to use locally mounted disks on some or all of our nodes, and if we need to reduce the number of messages whenever possible, then we are left with few scheduling options.

A second difficulty is that dependencies are hard for the Mentat compiler to find. The code generated by the ADAMS preprocessor re-uses variables frequently, as it must (having a unique C++ variable for each database item would fail quickly for large databases). Running the ADAMS preprocessed code through the Mentat compiler is only done for linking purposes and to acquire a few subroutine calls needed to initialize the Mentat client side. Parallel ADAMS knows where the potential parallel activity is (see section 3.0), leading the Mentat compiler “horse” to that “water” and forcing it to drink was

somewhat tricky.

The third difficulty is expressing consistency rules about our data. The cache coherency issue addressed in the first/next operations, for example. Or the need to complete all insertions into a set before performing a union operation with that set. The success of parallel ADAMS depended very much on our code maintaining strict control.

All of that being said, using Mentat did allow us to create our architecture with the sources of parallel execution as we envisioned them. Additionally, we were able to avoid coding at the socket level, which saved development time. The Mentat/C++ class structure helped to keep the system code modular, while the Mentat "sequential" guarantee of properly ordered messages between any two objects saved additional work. When ADAMS is ported to other hardware, it is likely that Mentat will have already been ported to it, which is a further advantage, as is support for heterogeneous configurations. And there is an existing suite of Mentat utilities which we have found valuable.

At present there must be a Mentat "configuration" running before an ADAMS program can be successfully run. The configuration consists of a collection of Mentat processes (instantiation managers, transaction management units, thermometer processes, along with another process or two for good measure) which are needed for Mentat to run properly. The configuration specification file currently used in the ADAMS group is shown in Figure 4.

```
HOST niv  { }
HOST juliet  { }
HOST jade  { }
HOST opal  { }
HOST viper  { }
HOST mamba { }
CLUSTER A {jade viper mamba niv juliet opal}
IM_PORT 6767
```

Figure 4. A Mentat Configuration File (config.db)

Unfortunately there is no way for an ADAMS program to determine if a configuration is running and the result is, rather than a message indicating that "Mentat is currently not running" and a quick exit by the program, a program that sits and does nothing and must be interrupted or otherwise terminated.

To bring up Mentat (using current Mentat tools) one must specify the desired

system configuration, i.e. a set of systems one may wish to be able to run one's programs on. The specification is stored in a *config.db* file, and at present an exact copy of the file must reside on every directory on which one will be running ADAMS programs. With the configuration specified, one can run the Mentat program *phoenix* which will bring up the desired configuration (with any luck). To bring Mentat back down one simply interrupts the *phoenix* program (which will continue running until it is killed, spitting out periodic diagnostic reports).

In the future any ADAMS program may be able to tell if Mentat is up and if not then it would bring it up without user intervention, using a *config.db* file stored in a single known location. Tools to enable these capabilities have been requested from the Mentat group.

4.2 The ADAMS Configuration

As Mentat must know what processors one may be using, ADAMS must be told where one's database files are and how many servers one would like on which processor. This information is to be specified in an ADAMS.CONFIG file (Figure 5) maintained on the directory where the ADAMS program is being run (*/users/adams/ADAMS.CONFIG* is used as a default).

```
This is the first line in the ADAMS.CONFIG file
--- #nodes follows
1
--- node 1 name
viper
--- directory 1
/users/adams/storage/server_storage
```

Figure 5. An ADAMS Configuration File (ADAMS.CONFIG)

It is easy to switch configurations, but it is important to be sure that all the files in one's configuration are in synch. If one is beginning a new configuration one must be sure that initialized database files are installed in each directory in the configuration. Also, it is

possible to specify within a configuration that more than one server is to have a particular directory. In the cluster, for example, duplicate directory names will be seen because the locally mounted disk on one node has the same name as the locally mounted disk on another. In this case there is no conflict. But point two servers at the same NFS mounted shared files and one will quickly be rewarded with a degraded database.

4.3 The Parallel Preprocessor

As mentioned, the ADAMS preprocessor converts the names of sets, maps, and attributes mentioned in an ADAMS program into *uids*. To do this, it makes use of dictionary utilities written in ADAMS, and data structures created and maintained within the ADAMS database. It generates C++ code, including calls into the ADAMS Interface Layer (see figure 1). In addition to converting an ADAMS program into a C++ program, an important function of the preprocessor is error checking. With such a complex object oriented language [Pfal95] there are opportunities for mistakes.

As a parallel application, the ADAMS preprocessor is not very efficient. For example, many singleton sets are created and manipulated. In many cases the attempts to achieve good parallel performance through data parallelism are thwarted due to a lack of parallel data, while attempts to pre-fetch also yield little success. We are then left with the overhead of message passing for low granularity operations, with the main benefit not being speed but multi-user access. It may be that preprocessor performance will improve as the rest of the system evolves. But improving the performance of the preprocessor is not a goal in itself at this time.

4.4 System Applications

A number of system tools are available. In describing these tools we use the terms “name space” and “task”. The terms have to do with specifying how the information in the database is shared and viewed and how the scope of any data element or name is determined. The terms are discussed in a more complete way in [Pfal95]. Here we simplify greatly and say that a name space of an ADAMS user consists of all the data names his programs can access, and that the user can indicate whether one of his data names can be

seen by only his programs, by the programs of everyone in his task group, or by the programs of everyone accessing the database. By data names we are refering to programmer supplied names maintained in the ADAMS database dictionary, and not *uids*. The number of names in the dictionary (such as "`S1`" in section 2.1) is likely to be small, while ADAMS programs can access vast quantities of unnamed data.

The tools include *dnewuser* (to establish a user's name space in the dictionary), *dnewtask* (to establish a task in the dictionary), *daddtask* (to add a user into a task in the dictionary), *dnamespace* (to view dictionary information), *dviewpath* (to display the data viewable from a specified scope), *dwarmstart* (to initialize selectable portions of a user's namespace), and *monitor* (which, unlike the other tools listed, does not have a counterpart in sequential ADAMS).

Parallel users may be particularly interested in *monitor*, which provides information on ADAMS server activity, listing which operations each server has performed and indicating how many calls have been processed (allowing a calculation of operations per client message). It can be valuable in determining what an application program does from a server's perspective.

The following *monitor* output (Figure 6) was generated on a single server configuration as a result of the preprocessing of a small ADAMS program (the sample database schema program given in Appendix C).

```
viper$ monitor

 ==========  ADAMS Manager Menu ==============

 1 - Show Server Information
 2 - Update Monitor Rate
 3 - Lock A UID
 4 - Unlock a UID
 5 - There isn't a five yet
 Q - Quits

And that's all there is for now...

Select Option - : 1

Server Reports:
        No Error
        1 users
        0 locks
        1 data servers

Users:
```

```
        0000000001cb0064

Locks:
        (nothing locked)

Disk Server Identification:
       viper
      /users/adams/storage/server_storage


Disk Server Processing Log:

            Call count = 3600
            Monitor Rate = 50
      441 - Set Creation Operation
      558 - Set Empty Operation
        4 - Set Delete Operation
      117 - Set Insert Element Operation
        9 - Set Remove Element Operation
      119 - Set Cardinality Operation
        9 - Set is Element Member? Operation
       54 - Set Union Operation
       11 - Set Intersection Operation
      429 - Set Copy Operation
      393 - Set First Element Operation
        9 - Set Change Persistence Operation
      441 - Set Get Parallel Data Structure Header
        2 - Set Parallel Header Deletion Operation
        9 - Set Parallel Header Change Persistence Operation
     361 - Set Get Several Next Elements AND some of their attributes too
       13 - Map Creation Operation
      365 - Map Store Element Operation
     1298 - Map Get Element Value Operation
       60 - Map Store forward map (one element)
       14 - Map Get Parallel Data Structure Header
        9 - Attribute Creation Operation
      195 - Attribute Assign Element Value Operation
      236 - Attribute Get Element Value Operation
        2 - Attribute Get Inverse Set Operation
        9 - Attribute Get Parallel Data Structure Header
        1 - System Initialization Operation
        1 - System Database Initialization Operation
        5 - System Request Acknowledgement Message Operation
        1 - System Update Monitor Rate
     5175   total operations
```

Figure 6. monitor output from preprocessing

As was indicated in section 4.3, the database operations required by the ADAMS preprocessor are generally not of high granularity. The operation receiving the largest number of requests is for the one object grained return of a Map value. As the server reports an operation to message ratio of 5175/3600 or 1.44 there has also been little of the message bundling discussed in section 3.1.

We cannot determine whether an ADAMS program has executed efficiently using *monitor* alone. The output resulting from *monitor* does not tell us whether, for example, one

of the two attribute inverse retrievals returned a set of two million elements, dwarfing the processing time of the all the other operations and providing an excellent opportunity for data parallel speedup (in a multi-server configuration).

But we know from experience that the sets manipulated by the preprocessor are very small. This knowledge coupled with the output from monitor supports the contention that the preprocessor is not going to be aided substantially by parallel execution at the level employed in ADAMS.

We can contrast this behavior with that of a very different program. In Figure 7 we show the *monitor* processing log output resulting from running a program which loads an ADAMS database with 25,000 objects (see appendix D).

```
Disk Server Processing Log:

           Call count = 1950
           Monitor Rate = 50
      21 - Set Creation Operation
      17 - Set Empty Operation
       9 - Set Delete Operation
   25002 - Set Insert Element Operation
       8 - Set Cardinality Operation
       1 - Set Union Operation
       7 - Set Intersection Operation
       1 - Set Copy Operation
       4 - Set First Element Operation
       2 - Set Change Persistence Operation
      25 - Set Get Parallel Data Structure Header
       9 - Set Parallel Header Deletion Operation
       2 - Set Parallel Header Change Persistence Operation
     1 - Set Get Several Next Elements AND some of their attributes too
   25101 - Map Store Element Operation
       1 - Map Get Element Value Operation
      51 - Map Load buffers with elements for new inverse map
      51 - Map Transfer buffers with elements for new inverse map
      50 - Map Receive buffers with elements for new inverse map
       1 - Map Transfer buffers with elements from map inverse of set
       1 - Map Receive buffers with elements from map inverse of set
   25002 - Map Store forward map (one element)
       5 - Map Get Parallel Data Structure Header
       1 - Map Parallel Header Change Has Inverse Operation
       1 - Map Load buffers with elements from map inverse of element
  125601 - Attribute Assign Element Value Operation
       4 - Attribute Get Inverse Set Operation
       3 - Attribute Get Inverse Range Set Operation
      13 - Attribute Get Parallel Data Structure Header
       1 - System Initialization Operation
       1 - System Database Initialization Operation
       5 - System Request Acknowledgement Message Operation
      53 - Synchronize Inter-Server Messages - High to Lower servers
      53 - Synchronize Inter-Server Messages - Lower to Higher servers
       1 - System Update Monitor Rate
  201109   total operations
```

Figure 7. monitor log output from database loading

The database load caused the server to process over 200,000 operations, but only 1950 messages were required (recall that the output of Figure 6 listed 3600). The operation to message ratio is 103 operations per message, much better than the 1.44 previously discussed. We want to limit the number of messages not because it could make the interconnection network a bottleneck (which is not something we expect to see), but because each message consumes substantial cpu on both the client and server processors. The implementation of message bundling quartered database load times in an earlier version of ADAMS, and as shown above it can be very effective.

The output from the running of another program is associated with a third behavior. Figure 8 shows the *monitor* log output resulting from running a query over the database established by the previous two ADAMS programs (see appendix E).

```
Disk Server Processing Log:

          Call count = 107
          Monitor Rate = 1
       19 - Set Creation Operation
       15 - Set Empty Operation
       16 - Set Delete Operation
        2 - Set Insert Element Operation
        3 - Set Cardinality Operation
        2 - Set Union Operation
        5 - Set Intersection Operation
        1 - Set Copy Operation
        4 - Set First Element Operation
        2 - Set Change Persistence Operation
       23 - Set Get Parallel Data Structure Header
       16 - Set Parallel Header Deletion Operation
        2 - Set Parallel Header Change Persistence Operation
      1 - Set Get Several Next Elements AND some of their attributes too
        1 - Map Store Element Operation
        1 - Map Get Element Value Operation
        1 - Map Transfer buffers with elements from map inverse of set
        1 - Map Receive buffers with elements from map inverse of set
        2 - Map Store forward map (one element)
        5 - Map Get Parallel Data Structure Header
        1 - Map Load buffers with elements from map inverse of element
        1 - Attribute Assign Element Value Operation
        3 - Attribute Get Inverse Set Operation
        3 - Attribute Get Inverse Range Set Operation
        6 - Attribute Get Parallel Data Structure Header
        1 - System Initialization Operation
        1 - System Database Initialization Operation
        6 - System Request Acknowledgement Message Operation
        1 - Enable Stream Processing
        1 - Synchronize Inter-Server Messages - High to Lower servers
       1 - Synchronize Inter-Server Messages - Lower to Higher servers
        6 - System Update Monitor Rate
      153   total operations
```

Figure 8. monitor log output from database querying

Figure 8 lists relatively few operations, and unlike the output from Figure 7 it is difficult to determine what, if anything of consequence, the program responsible for the operations was doing. We know that operations on sets of substantial size were performed, yet the monitor output would be nearly the same if the database contained 50 or 500,000 elements rather than 25,000 (`Map Transfer`, `Map Receive`, and `Synchronize Inter-Server` counts would be higher for 500,000). But this is in line with our goal of limiting messages and performing operations in a data parallel way. The client process does not need to know about query partial results, or to manage their creation in an involved way. During the query the time is spent on processing required to get the requested result, and this is as it should be.

5.0 Potential Future Work

There are a number of different projects which would enhance the current version of ADAMS and in some cases could produce results of more general interest.

5.1 Query Optimization

Currently the pre-processor generates query operations in an order directly corresponding to that specified by the ADAMS programmer, and the system at run time follows orders in much the same manner. In some cases there are more efficient orderings of operations (possibly allowing removal of some operations), and in others the query speed could be improved by accessing individual object attributes instead of intersecting with the results of a range search (this would be effective when the set of objects of interest is very small and a range search over the attribute of interest would result in a very large set).

Optimizing ADAMS queries would require developing methods to recognize and respond to the two situations above. Fortunately the only substantial complexity added by ADAMS parallelism is in the added cost to implicit joins (transfer of OIDs resulting from inverse map operations). We are not particularly interested in supporting explicit joins (queries involving operations such as `{x,y | x.attr1 = y.attr2}`) through server to server transfers or other extraordinary means as the object oriented model renders them far less important than in the relational model. There are often better ways to get the same or equivalent information.

There are two places where optimization techniques could be employed. The first is in the preprocessor, which supports optimization in that a flexible two-phase design is employed. The lexical analyzer and parser first generate a tree structured intermediate language which is then passed to the code generator. This tree structure could be manipulated and optimized. The second place is as part of the executing client process. The client sees a query as a list of operations. Analyzing this list along with information about the current state of the database could provide substantial benefits. As the executing client process will have a more current view of the database than the preprocessor (perhaps much more as a query may be rerun repeatedly over the life of a database without recompilation),

it may be the more aggressive optimization site. We have devoted little effort towards implementing either of these approaches.

The server structure of ADAMS may make optimization of ADAMS queries very different than anything currently existing. As mentioned, we don't have much use for explicit joins (support for implicit joins involving inter-server transfers of *uids* is provided) which are a focus of relational database processing, and we don't employ page servers (most existing OO systems rely on them).

Of particular interest in the optimization area is work by Graefe and others [GCDM94, Grae94, Grae93]. It should be noted that while there are a number of papers on parallel relational query processing and optimization, and some on object oriented query optimization, there is very little published work on parallel object oriented (non navigational) query processing. And these rely on data models, partitionings, or algorithms very different from ours. When results have been presented, they have generally been from simulations. A paper of particular interest is by Khoshafian, Valduriez, and Copeland [KhVC88], which discusses the decomposed storage model (similar to that employed by ADAMS, although we specify partitioning by object identifier). Other papers of interest include [LeTa95], [JWKL90], [HaSS88] and [ThSu94].

5.2 Navigational Query Support

Khoshafian defines navigational access as follows: "In this type of access various 'reachable' objects are accessed through attribute values or elements of referencing or parent objects. For instance, in intelligent office applications users can access a folder and then navigate to the elements of the folder. If a folder contains, say, another folder, the navigation can proceed with the elements of this folder and so on." ([Khos93], p.306). The status of navigational access is such that traversals and navigational-type queries are featured in current object oriented database benchmarks such as 007 [CaDN93], while complex conjunctive and disjunctive set queries such as those found in the relational set query benchmark [ONei93] are not well represented.

An example of a navigational query from 007 is: "Find all base assemblies that use a composite part with a build date later than the build date of the base assembly. Also, report

the number of qualifying base assemblies found"([CaDN94], p.26). Such a query would require multiple lines of ADAMS code, and would require accessing the build date attributes of all the base assemblies (an index could not be gainfully used to prune the base assembly access, as there is no information on which to initiate a worthwhile search). A page server based system, particularly one featuring clustering of objects on the same page, could perform relatively well on iterative search queries. Such queries certainly have great value, but one could ask whether focusing on them (rather than on complex set oriented queries) unduly rewards the simplicity of the page server approach.

The ADAMS task here is to examine navigational queries and determine whether the ADAMS system could reasonably adopt a multi-threaded approach towards their resolution. Currently, in the above 007 query, parallel pre-fetching would take place over the set of base assemblies, yielding good performance for that portion of the query. But the remainder of the task is sequential in nature for each set element, although performed over the entire set of objects (yielding an opportunity for parallel execution).

This area of inquiry is very much linked with the question of server executed object methods, yet the operations we would want for navigational query parallel/multi-threaded execution need not be user specified (and ideally in typical navigational queries should not have to be). Part of the problem is to determine, if the methods are not user specified, what the server functions needed to handle such processing are and how we can use the pre-processor and run time system to translate an ADAMS navigational query into the correct and appropriate server functions. There are further complications offered by navigational queries which follow several layers of map references, forcing a multi-threaded scheme to handle inter-server requests, and to avoid deadlock situations while attempting to provide good performance.

One ADAMS design goal has been to move the code to the relevant data, and this has yielded good results with complex conjunctive/disjunctive queries. A problem with navigational queries is that where the data is within an iterative thread can change rapidly. Relevant work in this area includes research related to the University of Wisconsin's SHORE project ([DNSV94]), Michael Kilian's work with parallel sets ([Kili92]), and query optimization work by Zhuoan Jiao and Peter Gray [JiGr91].

5.3 Support for User Supplied Methods

All the commercial object oriented database systems that we know of support user defined object methods. But the choice not to pursue such support within ADAMS, based on a need to devote resources to other areas of interest, and on a desire to maintain the ADAMS language in it's current elegant form, is quite reasonable given that ADAMS is a research system. As ADAMS applications development is usually done with C++ as the host language, enhancing an ADAMS class in a virtual way with C++ methods should not be difficult.

In the future user method support may be provided. Naturally the execution of these methods would be done at the object's server processor, particularly in the case of complex queries. Query optimization under such circumstances would be more difficult, and inter-server processing issues (such as those raised in the previous section) would need to be handled for those methods accessing objects through maps. It is likely that queries involving both indices and methods would be well served by processing the method portion last, as the method execution cost per object could be substantial. The language addition would require careful design, as would any changes to the ADAMS Server interface. An approach similar to that used by SHORE [CDFH94], which apparently allows a "value added server" residing on the SHORE server's processor to handle custom methods, could be implemented using Mentat without great difficulty (and with little risk to the ADAMS Servers). But we would prefer to do something radically different if a reasonable alternative could be found.

Other relevant references (aside from those in the previous section) including a very detailed article by Jonathan Wilcox [Wilc94].

5.4 Object Clustering

In page server database systems multiple objects are often stored on a single page. Should the user have the ability to forecast his access patterns well, a system allowing the him/her to specify a page co-location preference for his/her objects can reduce both the number of messages from the client to the server and the amount of disk I/O at the server.

As ADAMS does not store data for an object contiguously (see section 2.1), page

co-location of entire objects in ADAMS is not possible. But we expect parallel servers to be able to fill an ADAMS client processes needs for object data without difficulty (determining the correct objects to ship to the client, i.e. resolving the query, being the time consuming part of typical ADAMS programs over large data sets).

There are two potentially beneficial ways in which the current version of ADAMS could employ the clustering notion. But not as page co-location (assuming the file system does a reasonable job of limiting fragmentation within O-tree nodes, and that we are reading a page/node at a time from disk), rather as partition co-location.

The two ways require the user to specify desired partition co-location patterns for objects (how the user should best specify "put this object on the same server as that object" is unclear). For the ADAMS run time system this would only slightly complicate *uid* assignment. The first benefit would be that in inverse map querying over colocated objects fewer *uids* would need to be sent between partitions. The second benefit would be that in iterative data pre-fetching the set first/next code could be made even more aggressive, pre-fetching not just attribute and map values for objects in the set, but data from the objects referenced by the map values (as this data could be on that server).

We are not very optimistic about this strategy yielding significant gains in performance. The co-location pattern in the data needs to be such that it is achievable when the data is N-partitioned, but if many objects have the same co-location target achieving the co-location pattern would cause a data imbalance in the partitions and hurt performance. Should an imbalance not occur, the degree to which co-location would help is likely to be small. As mentioned, we do not expect the transfer of object data from the servers to a client process to be a bottleneck, and in ADAMS query processing we feel that the inter-server communication cost associated with inverse maps is only a minor part of processing a complex query.

Should ADAMS be enhanced with user defined methods or navigational query support (see sections 5.2 and 5.3) the performance implications of partition co-location change. Reducing the need for small granularity threads to generate inter-server messages to resolve map references could improve their performance substantially. Determining how great this gain could be (or whether the granularity of the operations could be increased in some other way) remains to be seen. An adjunct to this issue has to do with the replication

of objects for performance reasons, requiring not only a co-locate language directive but a locate in partition directive. This would lead us away from maintaining ADAMS as configuration independent, which is undesirable. The problem would be to determine when such drastic changes would yield a significant benefit. Papers on object clustering include (disk clustering) [BeDe90] and (partition placement) [GWLZ94].

5.6 Skew Management

It would be a simple task to modify the *uid* generation algorithm to favor object identifier assignments to or away from a particular partition. Determining when such modifications should be made or employed should also be simple, involving measuring response times for various actions over all partitions. Whether there is already too much literature on skew in parallel databases is not clear. One paper regarding skew is [LaYu88].

5.7 Repartitioning Support

If an ADAMS user wishes to create a database and expects that more processors will be available at a later date it is easy to assign multiple servers to a single processor and then, when more processors become available, move a partition's files and processor assignment to a new processor. If there already exists a one-to-one relationship between processors and partitions and more processors become available the current solution would be to download the entire database and reload it from the beginning. At some point it may be worthwhile to support automatic database re-partitioning, and avoid the total download/ reload solution.

On systems where data is continually added, a non-disruptive method would involve simply modifying the partitioning function to assign the partition using a function based on a series of *uid* sub-ranges (as *uid*s increase with time, early *uid*s would be associated with a function using a small number of partitions, while later *uid*s would be larger). A difficulty is that old *para_uid*s would have to be dynamically modified to contain more non-null *sub_uid*s, but this could be done.

5.8 Distributed Database Functionality

We see the main issue for ADAMS here as supporting two (or possibly more) *para_uid* types. In the case of strictly local sets, maps, or attributes, *para_uids* would contain N *uids* where N is the number of partitions in the local configuration. Global *para_uids* would contain more than N *uids*. The *uid* allocation algorithm would use information about the overall database configuration.

A similar use of multiple para_uid versions could allow for selective querying based on age or other data characteristic. If we assigned older, perhaps infrequently desired or accessed, data to one set of partitions (perhaps supported by slower hardware) it would be possible to process queries on other portions of the database rather than on the entire configuration. For systems supporting a great deal of historical data this capability could be very beneficial.

5.9 Attribute Value Compression

There is a disk storage penalty incurred when maintaining object data in a non-contiguous form. A single tree structure is replaced with many. One way the ADAMS parallel server system could reclaim disk storage is by maintaining attribute values in a compressed form. As we've indicated, the server processors are generally involved with I/O and the cpu is underutilized. The reduced storage size resulting from attribute compression may also reduce I/O costs due to increased file cache effectiveness and fewer blocks transferred to and from disk.

On a related note, this could particularly improve ADAMS performance when working with multimedia information. The system has not been tested with very large attribute values and compression will be even more important when such applications are created.

6.0 Other Systems

An active research system we have found of interest is the SHORE project at the University of Wisconsin ([CDFH94],*http://www.cs.wisc.edu/shore/shore.home.html*). While there is relatively little literature on parallel object oriented database systems, they have been very productive. Several of the papers referenced in this document are the resılt of their research.

We have found it difficult to obtain up to date technical information on activity in the commercial sector. There appears to be great interest in distributed servers, such that data can be retrieved over a large distributed configuration, but little interest in manipulating this information in a parallel way. The Versant product ([Vers93], *http:// www.versant.com:80/welcome/*) employs an interesting server architecture, but page servers apparently dominate the field. This may change rapidly.

While both of the above groups will continue to make great strides towards their respective goals, it’s possible that the small size of the ADAMS group may make it particularly effective. The “small sharp team” concept discussed by Brooks (p.30,[Broo75]) may well be at work. The use of Mentat reduces the size of the development task, while providing useful (and when manipulated sufficiently, unrestrictive) parallel development tools. The ADAMS language has been in existence for some time. Thus the team is generally able to focus on parallel database issues without the burdens of user language design or low level socket coding.

7.0 Conclusions

We have described some features of ADAMS (particularly those related to parallelism) and their implementation. The interested reader is strongly encouraged to review the ADAMS language [Pfal95] to gain a broader view of what the ADAMS system has to offer.

In section five we have discussed features not yet part of ADAMS. This section may have been overly stressed. We have been known to see a glass at half capacity as not half full, or even half empty, but instead as unaccountably missing. To put the work presented in context, with a small research team and a very limited amount of time we have designed and developed a database system such that we are able to consider this list of issues as possible next steps in development work. This question is not whether these and other issues can be dealt with, but which ones best advance the research.

8.0 Bibliography

[BeDe90] Veronique Benzaken and Claude Delobel, "Enhancing Performance in a Persistent Object Store: Clustering Strategies in O_2", *Implementing Persistent Object Bases: Principles and Practice, Proceedings of the Fourth International Workshop on Persistent Object Systems,* Martha's Vineyard, 1990. pp. 403-412.

[Broo75] Frederick P. Brooks, Jr., The Mythical Man-Month, Addison-Wesley, Reading, Mass. 1975.

[CaDN94] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton, The 007 Benchmark, Technical Report, Computer Sciences Department, University of Wisconsin-Madison, file://ftp.cs.wisc.edu/oo7/.

[CDFH94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, et al., "Shoring Up Persistent Applications",*Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data,* Minneapolis, Minnesota, May 1994. SIGMOD Record, Volume 23, Issue 2, June 1994, pp. 383-394.

[CoKh85] George P. Copeland and Setrag N. Khoshafian, "A Decomposition Storage Model",*Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data,* Austin, Texas, May, 1985 pp. 268-279.

[DNSV94] David J. DeWitt, Jeffrey F. Naughton, John C. Shafer, and Shivakumar Venkataraman, "ParSets for Pallelizing OODBMS Traversals: Implementation and Performance", http://www.cs.wisc.edu/shore/shore.papers.html.

[GCDM94] Goetz Graefe,Richard L. Cole, Diane L. Davison, William J. McKenna, and Richard H. Wolniewicz, "Extensible Query Optimization and Parallel Execution in Volcano", Query Processing For Advanced *Database Systems,* Johann Freytag, David Maier, and Gottfried Vossen, editors, Morgan Kaufmann, San Mateo 1994.

[GWLZ94] Shahram Ghandeharizadeh, David Wilhite, Kaming Lin, and Xiaoming Zao, "Object placement in Parallel Object-Oriented Database Systems", *Proceedings of the 10th International Conference on Data Engineering,* Houston, Texas 1994, pp. 253-262.

[Grae93] Goetz Graefe, "Query Evaluation Techniques for Large Databases", ACM Computing Surveys, Volume 25, Number 2. June, 1993

[Grae94] Goetz Graefe,"Volcano - An Extensible and Parallel Query Evaluation System", IEEE Transactions on Knowledge and Data Engineering, Volume 6, No. 1, February 1994

[HaPf96] Russell F. Haddleton and John L. Pfaltz, “Parallelism in Scientific Database Queries”, *Proceedings of the Eighth International Working Conference on Scientific and Statistical Database Management*, Stockholm 1996. Submitted.

[HaSS88] T. Harder, H. Schoning, and A. Sikeler, “Parallelism in Processing Queries on Complex Objects”, *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems,* Austin, 1988, pp. 131-143.

[JiGr91] Zhuoan Jiao and Peter M. D. Gray, “Optimisation of Methods in a Navigational Query Language”,*Proceedings of the Second International Conference on Deductive and Object-Oriented Databases, DOOD ‘91*. Munich, 1991, pp. 22-42.

[JWKL90] B. Paul Jenq, Darrell Woelk, Won Kim, and Wan-Lik Lee, “Query Processing in Distributed ORION”, *Proceedings of the International Conference on Extending Database Technology, EDBT ‘90,* Venice, 1990, pp. 169-187.

[Khos93] Setrag Khoshafian, Object Oriented Databases, John Wiley & Sons, New York, 1993.

[KhVC88] Setrag Khoshafian, Patrick Valduriez, and George Copeland, “Parallel Query Processing for Complex Objects”, *Proceedings of the 4th International Conference on Data Engineering,* Los Angeles, 1988, pp. 202-209.

[Kili92] Michael Francis Kilian, *Parallel Sets: An Object-Oriented Methodology for Massively Parallel Programming*,The Division of Applied Sciences, Harvard University, Cambridge, MA, 1992 (Ph.D Dissertation)

[LaYu88] M. Seetha Lakshmi and Philip S. Yu, “Effect of Skew on Join Performance in Parallel Architectures”, *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*. Austin, Texas. December 5-7, 1988. pp. 107-120.

[LeTa95] Leung, C.H.C, and Taniar, D., “Parallel Query Processing in Object-Oriented Databases Systems”, *Proceedings of the Sixth Australasian Database Conference (ADC’95),* Adelaide, Australia, (Australian Computer Science Communications, vol 17, no 2, 1995), January 1995, pp. 119-131.

[McEl91] McElrath, Rodney, *A Look at Two Persistant Storage Models*, Technical Report IPC-91-11, Department of Computer Science, University of Virginia, 1991.

[Ment95] The Mentat Research Group,*Mentat 2.8 System Reference*, Department of Computer Science,University of Virginia. http://www.cs.virginia.edu/~mentat/.

[ONei93] Patrick E. O'Neil, "The Set Query Benchmark", *The Benchmark Handbook For Database and Transaction Processing Systems",* Jim Gray, editor, Morgan Kaufmann, San Mateo 1993. pp.359-395.

[OrPf88] Ratko Orlandic and John L. Pfaltz, "Compact 0-Complete Trees", *Proceedings of the 14th VLDB Conference,* Los Angeles, CA 1988, pp. 372-381.

[Pfal95] John L. Pfaltz, *The ADAMS Language: A Tutorial and Reference Manual,* Technical Report IPC-93-003, Department of Computer Science, University of Virginia, 1995.

[ThSu94] Arun K. Thakore and Stanley Y. W. Su, "Performance Analysis of Parallel Object-Oriented Query Processing Algorithms", *Distributed and Parallel Databases 2* (1994). Kluwer Academic Publishers, Boston. 1994. pp. 59-100.

[Vers93] Versant ODBMS, A Technical Overview for Software Developers Versant Object Technology Menlo Park, CA 1993

[Wilc94] Jonathan Wilcox, "Object Databases, Object methods in distributed computing", Dr. Dobb's Journal, Volume 19, Issue 13, November, 1994, pp. 26-34.

Appendix A - Client Interface

```
                /* interface.i */

/************************************************/
/*          _A_Set Operations                   */
/************************************************/

void    _A_newset (_A_UID_PTR set_uid, _A_UID_PTR
                   class_uid,_A_BOOLEAN persistence);
        /*
        **  Create an initially empty set that will be
        **  denoted by 'set_uid',
        **  which may be either persistent or non-
        **  persistent.
        */

void    _A_set_make_empty (_A_UID_PTR set);
        /*
        **  Make an existing 'set' the empty set (i.e.
        **    NULL set)
        */

void    _A_set_display_header (_A_UID_PTR set);
        /*
        **  Display the header of set.
        */

void    _A_set_delete (_A_UID_PTR set);
        /*
        **  Delete existing 'set'.
        **   (i.e. completely remove its representation)
        */

void    _A_set_insert (_A_UID_PTR element, _A_UID_PTR
                       set);
        /*
        ** Inserts 'element' (actually just its uid)
        ** into 'set'.
        */

int     _A_set_card (_A_UID_PTR set);
        /*
        **  Returns the cardinality of 'set'.
        */

void    _A_set_remove (_A_UID_PTR element, _A_UID_PTR
                       set);
        /*
        **  Remove 'element' from 'set' (i.e. delete
        **  its uid)
        */

int     _A_set_member (_A_UID_PTR element, _A_UID_PTR
                       set);
        /*
        **  Is 'element' a member of 'set'? (I.e. set
        **  membership test)
        **  Return  1 if true
        **          0 if false.
        */
```

```
void    _A_set_union (_A_UID_PTR result , _A_UID_PTR
                      set_1, _A_UID_PTR set_2);
        /*
         **  A new ‘result’ set with is the UNION of
         ** ‘set_1’ and ‘set_2’ is
         **  created.
         */

void    _A_set_intersect (_A_UID_PTR result , _A_UID_PTR
                         set_1, _A_UID_PTR se);
        /*
        **  A new ‘result’ set with is the INTERSECTION
        ** of ‘set_1’ and ‘set_2’
        **  is created.
        */

void    _A_set_complement (_A_UID_PTR result, _A_UID_PTR
                          set_1, _A_UID_PTR set_2);
        /*
        **  A ‘result’ set which is the COMPLEMENT of
        ** ‘set_1’ WITH RESPECT
        **  to ‘set_2’ is created.
        **  Note that this is a relative complement.
        */

void    _A_set_copy (_A_UID_PTR dest_set, _A_UID_PTR
                     source_set);
        /*
        **  ‘dest_set’ is made to be a SHALLOW COPY of
        **  ‘source_set’
        **   That is, ‘dest_set’ is first emptied, then
        **   every element uid in
        **  ‘source_set’ is inserted into ‘dest_set’.
        **   or equivalently
        **              dest_set <- source_set
        **  ‘source-set’ remains unchanged.
        */

int     _A_set_first_element (_A_UID_PTR element_1,
                              _A_UID_PTR set);
        /*
        **  Assign to ‘element_1’ the first element of
        ** ‘set’.
        **  return: int 1 if ‘set’ is non-empty;
        **          int 0 if ‘set’ is empty.
        */

int     _A_set_next_element (_A_UID_PTR next_element,
                             _A_UID_PTR set);
        /*
        **  Set ‘next_element’ to be the next element
        **  uid in the ‘set’
        **  Return: 1 if there was a ‘next_element’;
        **          0 if the elements of ‘set’ have
        **          been exhausted.
        **
        **  Note: An invocation of _A_set_first_element
        **  MUST precede
        **        any use of this procedure.
        */

void    _A_Enable_Streams();
        /*
            Allows programmer willing to “play fair” with
```

```
           temporary
            sets to use streams.
            Example of playing fair : not expecting a
            temporary set
            to contain anything if it's contents were
            copied to another
            set. Once a stream is empty, it's empty.
        */


/*********************************************/
/*            _A_map operations                */
/*********************************************/

void    _A_newmap (_A_UID_PTR map_uid, _A_UID_PTR
                   class_uid,_A_BOOLEAN persistence);
        /*
        **  Create an initially empty map that will be
        **  denoted by 'map_uid',
        **  which may be either persistent or non-
        **  persistent.
        */

void    _A_map_assign (_A_UID_PTR map, _A_UID_PTR
                       element, _A_UID_PTR image);
        /*
        **  Makes the assignment 'element'.'map' <-
        **  'image'.
        **  Note that the image of a map is always an
        **  element uid.
        */

void    _A_map_assign (_A_UID_PTR map, _A_UID_PTR
                       element, _A_UID_PTR image);
        /*
        **  Makes the assignment 'element'.'map' <-
        ** 'image'.
        **  Note that the image of a map is always an
        **  element uid.
        */

void    _A_map_image(_A_UID_PTR map, _A_UID_PTR element,
                     _A_UID_PTR image);
        /*
        **  Retrieve the 'image' uid of the 'map'
        **  applied to 'element'
        **  that is,  'element'.'map'
        */

void    _A_map_remove (_A_UID_PTR map, _A_UID_PTR
                       element);
        /*
        **  Set the value of 'element'.'map' to NULL
         **  I.e. make the 'map' be undefined on 'element'.
        */

_A_UID_PTR  _A_map_inverse (_A_UID_PTR map, _A_UID_PTR
                            image);
        /*
        **  Create a 'pre_image_set' of all uid's for
        **  which uid.'map' = 'image'
        */
```

```
_A_UID_PTR  _A_map_set_inverse (_A_UID_PTR map,
                                _A_UID_PTR set_image);
        /*
        **  Create a 'pre_image_set' of all uid's for
        **  which uid.'map' = some
        **  uid in 'set_image'
        */

/***********************************************/
/*           _A_attr operations                */
/***********************************************/

void    _A_newattr (_A_UID_PTR attr_uid, _A_UID_PTR
                    class_uid,_A_BOOLEAN persistence);
        /*
        **  Create an initially empty attribute that will
        **  be denoted by
        **  'attr_uid',
        **  which may be either persistent or non-
        **  persistent.
        */

void    _A_newattr (_A_UID_PTR attr_uid, _A_UID_PTR
                    class_uid,_A_BOOLEAN persistence,
                    char *filter_name);
        /*
        **  Create an initially empty attribute that
        **     will be denoted by
        **  'attr_uid',
        **  which may be either persistent or
        **  non-persistent.
        */

void    _A_attr_assign(_A_UID_PTR attribute, _A_UID_PTR
                       element, char* value,
                       int nbr_bytes);
        /*
        **  Make assignment 'element'.'attribute' <-
        **  'value'.
        **  'value' is a string representation of the
        **  desired value.
        */


int     _A_attr_value (_A_UID_PTR attribute, _A_UID_PTR
                       element,
                       char* buffer, int buf_size);
        /*
        **  Copies the value of 'element'.'attribute'
        **  in 'buffer'.
        **  Returns:  Number of bytes written to buffer.
        */

void    _A_attr_remove(_A_UID_PTR attribute, _A_UID_PTR
                       element);
        /*
        **  Function:  Sets the value of
        ** 'element'.'attribute' to NULL
        **  Parameters are both _A_UID_PTRs.
        */

void    _A_attr_copy (_A_UID_PTR attribute1, _A_UID_PTR
                      element1,
```

```
                    _A_UID_PTR attribute2, _A_UID_PTR
                    element2 );
        /*
        ** Function: Performs the attribute assignment:
        **      'element1'.'attribute1' <-
        **'element2'.'attribute2'
        ** NOTE:  This function is not currently used,
        **        but could be
        **        helpful for optimization.
        */

_A_UID_PTR _A_attr_inverse (_A_UID_PTR attribute,
                            char* searchval, int
                            searchval_len);
        /*
        **  Create the 'pre_image_set' of all elements
        **  whose 'attribute' values
        */

_A_UID_PTR  _A_attr_range_inverse (_A_UID_PTR attribute,
                        char* lowval, int lowval_len,
                        _A_ENDPOINT lowval_incl,
                        char* hival,  int hival_len,
                        _A_ENDPOINT hival_incl);
        /*
        ** Function: Find all elements whose 'attribute'
        **           value is between
        **            'lowval' and 'hival.'  These elements
        **           are returned
        **           as a SET function value.
        ** Parameters: _A_UID_PTR attribute;  char rep
        **                           of attr uid.
        **               char*     lowval;     char pointer
        **                       to first search val.
        **               char*     hival;      char pointer
        **                       to second search val.
        **               int       lowval_len; length
        **                           (bytes) of low val.
        **               int       hival_len;  length
        **                          (bytes) of high val.
        **               _A_ENDPOINT lowi_incl;closed
        **                              (include lowval)
        **               _A_ENDPOINT hi_incl;  open (do
        **                            not include hival)
        */

/*************************************************/
/*          _A_subscript operations              */
/*************************************************/

void    _A_newsubscript (_A_UID_PTR subscr_uid,
                         _A_BOOLEAN persistence);
        /*
        **  Create an initially empty subscript that will
        **  be denoted by
        ** 'subscr_uid', which may be either persistent
        **  or non-persistent.
        */

void    _A_subscript_assign (_A_UID_PTR subscr_uid,
                             unsigned long key,
                            _A_UID_PTR elem_uid);
        /*
        **  Makes 'subscr_uid'['key'] denote 'elem_uid'.
```

```
        **  Here ‘subscr_uid’ denotes the subscript
        **  O-tree identified by
        **  a subscripted name in the dictionary.
        **  When the particular subscripts are combined
        **  to form a single
        **  integer ‘key’, then the subscripted name
        **  ‘subscr_uid’[‘key’]
        **  denotes this particular subscripted instance,
        **  i.e. ‘elem_uid’.
        */

void    _A_subscript_remove (_A_UID_PTR subscr_uid,
                             unsigned long key);
        /*
        **  Remove ‘key’ as a defined argument in the
        ** ‘subscr_uid’ O-tree.
        **  That is, ‘subscr_uid’[‘key’] is no longer
        **  defined.
        **  NOTE: The reference counter of the current
        **  element denoted
        **  by subscr_uid[key] SHOULD be decremented,
        **  and possibly
        **  the element REMOVED.
        */

void    _A_subscript_get_val (_A_UID_PTR subscr_uid,
                              unsigned long key,
                             _A_UID_PTR elem_uid);
        /*
        **  Gets the ‘elem_uid’ denoted by the
        **  subscripted expression
        **  ‘subscr_uid’[‘key’].
        */

unsigned long   _A_eval_subscript (int n, int
                                   subscript_list[]);
        /*
        **  Return a single long integer corresponding
        **  to the
        **  diagonal evaluation of the ‘n’ integer
        **  subscript
        **  values in ‘subscript_list[]’.
        */

/************************************************/
/*          _A_uid Operations                   */
/************************************************/

void    _A_uid_getuid (_A_UID_PTR uid);
        /*
        **  Get a new ‘uid’.
        **  Note: invocation would normally be
        **      _A_uid_getuid (_A_UID_PTR uid)
        **  but ‘uid’ argument must be declared (char *)
        **  for compatibility
        **  with the runtime system.
        */

void    _A_change_persistence (_A_UID_PTR uid,
                               _A_BOOLEAN persistence);
        /*
        **  Change the ‘persistence’ of ‘uid’.
        */
```

```
void    _A_record_elem_class (_A_UID_PTR elem_uid,
                              _A_UID_PTR class_uid);
        /*
        **  Record in the _A_class_map that 'elem_uid'
        **  belongs to
        **  'class_uid'.
        **  NOTE:
        **(1) only persistent elements should be entered
        **  into the
        ** _A_class_map until a protocol for removing
        **  non-persistent elements is developed.
        */

int     _A_class_of_uid (_A_UID_PTR elem_uid,
                         _A_UID_PTR class_uid);
        /*
        ** Returns TRUE if elem_uid is an instantiated
        ** element, and
        ** puts its associated class uid in 'class_uid';
        ** returns FALSE
        ** if 'elem_uid' is not an instantiated element,
        ** and puts
        ** _A_NULLUID in 'class_uid'.
        */

int _A_setuid_is_Persistent (_A_UID_PTR set_uid);
        // This function returns 0 if the set designated
        // by set_uid is not
        // persistent, and 1 if it is.  The programmer
        // is responsible for ensuring
        // that set_uid designates a set.

int _A_get_nbr_blockreads (_A_UID_PTR a_uid, char*
                           a_type);
        //  This function is intended for use in the
        //  C-code of an adams program.
        //  It is used by passing an A_UID_PTR and the
        //  type of otree to which
        //  it refers, either "ATTR", "MAP", or "SET",
        //  as parameters.  It returns
        //  the total number of blockreads since the
        //  creation of that otree.

int _A_get_nbr_blockwrites (_A_UID_PTR a_uid, char*
                            a_type);
        //  This function is intended for use in the
        //  C-code of an adams program.
        //  It is used by passing an A_UID_PTR and the
        //  type of otree to which
        //  it refers, either "ATTR", "MAP", or "SET",
        //  as parameters.  It returns
        //  the total number of blockwrites since the
        //  creation of that otree.

int _A_get_total_blocks (_A_UID_PTR a_uid, char*
                         a_type);
        //  This function is intended for use in the
        //  C-code of an adams program.
        //  It is used by passing an A_UID_PTR and the
        //  type of otree to which
        //  it refers, either "ATTR", "MAP", or "SET",
        //  as parameters.  It returns
         //  the total number of blocks used by that otree.
        //  If the otree is a
```

```
        //  small set, it assumes 1.


#ifdef SERVER

void _A_set_lock_list_entry(_A_UID_PTR a_uid);
        //  This function is intended for use in the
        //  C-code of an adams program.
        //  It is used by passing an A_UID_PTR. If
        //  running under the multi-user version
        //  The associated uid is inserted into the
        //  allocator's lock_list.

void _A_remove_lock_list_entry(_A_UID_PTR a_uid);
        //  This function is intended for use in the
        //  C-code of an adams program.
        //  It is used by passing an A_UID_PTR. If
        //  running under the multi-user version
        //  The associated uid is removed from the
        //  allocator's lock_list.

void _A_clear_lock_list();
        //  This function is intended for use in the
        //  C-code of an adams program.
        //  If running under the multi-user version
        //  all entries are removed from the allocator's
        //  lock_list.

int  _A_attempt_locks(_A_UID_PTR a_uid);
        //  1 on success, 0 on failure. If under multi-
        //  user version, sends
        //  allocator's lock list to the lock server to
        //  see if all uids on the
        //  list can be locked.

void _A_set_unlock_list_entry(_A_UID_PTR a_uid);
        //  This function is intended for use in the
        //  C-code of an adams program.
        //  It is used by passing an A_UID_PTR. If
        //  running under the multi-user version
        //  The associated uid is inserted into the
        //  allocator's unlock_list.

void _A_remove_unlock_list_entry(_A_UID_PTR a_uid);
        //  This function is intended for use in the
        //  C-code of an adams program.
        //  It is used by passing an A_UID_PTR. If
        //  running under the multi-user version
        //  The associated uid is removed from the
        //  allocator's unlock_list.

void _A_clear_unlock_list();
        //  This function is intended for use in the
        //  C-code of an adams program.
        //  If running under the multi-user version
        //  all entries are removed from the allocator's
        //  unlock_list.

int  _A_attempt_unlocks();
        //  1 on success, 0 on failure. Shouldn't fail.
        //  If under multi-user version, sends
        //  allocator's unlock list to the lock server
        //  for unlocking.
```

```
void  _A_unlock_all();
        //  Unlocks all the uids locked by the calling
        //  process.

#endif


/************************************************/
/*         Parallel Timer Operations            */
/************************************************/



void    _A_Synch_servers();
         /*
         **  In distributed ADAMS, force the servers
         **  to catch up with
         **  each other. In the sequential version do
         **  nothing.
         */

void  _A_Time_servers();

void  _A_Time_display_servers();
```

Appendix B - Server Instruction Code List

```
 // SET Codes
_A_PCD_SET_NEW 51
_A_PCD_SET_EMPTY 52
_A_PCD_SET_DELETE 53
_A_PCD_SET_INSERT 54
_A_PCD_SET_REMOVE 55
_A_PCD_SET_CARDINALITY 56
_A_PCD_SET_MEMBER 57
_A_PCD_SET_UNION 58
_A_PCD_SET_INTERSECT 59
_A_PCD_SET_COMPLEMENT 60
_A_PCD_SET_COPY 61
_A_PCD_SET_FIRST_ELEM 62
_A_PCD_SET_NEXT_ELEM 63
_A_PCD_SET_IS_PERSIST 64
_A_PCD_SET_NBR_BLOCKREADS 65
_A_PCD_SET_NBR_BLOCKWRITES 66
_A_PCD_SET_NBR_BLOCKS 67
_A_PCD_SET_CHANGE_PERSISTENCE 68
_A_PCD_SET_SAVE 69
_A_PCD_SET_DISP_HDR 70
_A_PCD_SET_GETHEADER 71
_A_PCD_SET_DELETE_HEADER 72
_A_PCD_SET_CHANGE_HEADER_PERSISTENCE 73

// MAP Codes
 _A_PCD_MAP_NEW 101
 _A_PCD_MAP_IMAGE 102
 _A_PCD_MAP_REMOVE 103
 _A_PCD_MAP_DISP_HDR 104
 _A_PCD_MAP_DELETE 105
 _A_PCD_MAP_GET_SET 106
 _A_PCD_MAP_STORE 107
 _A_PCD_MAP_GET_VAL 108
 _A_PCD_MAP_NBR_BLOCKREADS 109
 _A_PCD_MAP_NBR_BLOCKWRITES 110
 _A_PCD_MAP_NBR_BLOCKS 111
 _A_PCD_MAP_LOAD_INVERSES 112
 // for new inverse, old map
 _A_PCD_MAP_TRANSFER_INVERSES 113
 // for new inverse, old map
 _A_PCD_MAP_RECEIVE_INVERSES 114
 _A_PCD_MAP_LOAD_SET_INVERSE_RESULT 115
 // for existing inverse
 _A_PCD_MAP_TRANSFER_INVERSE_RESULT 116
 // for existing inverse
 _A_PCD_MAP_RECEIVE_INVERSE_RESULT 117
 // for existing inverse
 _A_PCD_MAP_STORE_INVERSE 118
 _A_PCD_MAP_STORE_FORWARD 119
 _A_PCD_MAP_REMOVE_INVERSE 120
 _A_PCD_MAP_GETHEADER 121
 _A_PCD_MAP_DELETE_HEADER 122
 _A_PCD_MAP_CHANGE_HEADER_PERSISTENCE 123
 _A_PCD_MAP_CHANGE_HEADER_HAS_INVERSE 124
 _A_PCD_MAP_GET_SET_FROM_SET 125
 _A_PCD_MAP_LOAD_ELEMENT_INVERSE_RESULT 126

// Attribute Codes
 _A_PCD_ATTR_MINOP 150
```

```
 _A_PCD_ATTR_NEW 151
 _A_PCD_ATTR_DISP_HDR 152
 _A_PCD_ATTR_ASSIGN_BYTES 153
 _A_PCD_ATTR_VALUE 154
 _A_PCD_ATTR_REMOVE 155
 _A_PCD_ATTR_COPY 156  // not used
 _A_PCD_ATTR_DELETE 157
 _A_PCD_ATTR_INVERSE 158
 _A_PCD_ATTR_RANGE_INVERSE 159
 _A_PCD_ATTR_NBR_BLOCKREADS 160
 _A_PCD_ATTR_NBR_BLOCKWRITES 161
 _A_PCD_ATTR_NBR_BLOCKS 162
 _A_PCD_ATTR_GETHEADER 163
 _A_PCD_ATTR_DELETE_HEADER 164
 _A_PCD_ATTR_CHANGE_HEADER_PERSISTENCE 165
 _A_PCD_ATTR_SET_FILTER 166
 _A_PCD_ATTR_CHANGE_HEADER_HAS_INVERSE 167


// Subscript Codes

  _A_PCD_SUBSCRIPT_GETHEADER 201
  _A_PCD_SUBSCRIPT_DELETE 202
  _A_PCD_SUBSCRIPT_DISP_HDR 203
  _A_PCD_SUBSCRIPT_REMOVE 204
  _A_PCD_SUBSCRIPT_STORE 205
  _A_PCD_SUBSCRIPT_GETVAL 206
  _A_PCD_SUBSCRIPT_CHANGE_PERSISTENCE 207
  _A_PCD_SUBSCRIPT_NBR_BLOCKREADS 208
  _A_PCD_SUBSCRIPT_NBR_BLOCKWRITES 209
  _A_PCD_SUBSCRIPT_NBR_BLOCKS 210
  _A_PCD_SUBSCRIPT_DELETE_HEADER 211
  _A_PCD_SUBSCRIPT_NEW 212
  _A_PCD_SUBSCRIPT_EMPTY 213
  _A_PCD_SUBSCRIPT_CHANGE_HEADER_PERSISTENCE 214

// General Operation Codes

  _A_PCD_SYSTEM_INIT 251
  _A_PCD_SYSTEM_INIT_DB 252
  _A_PCD_SYSTEM_SHUTDOWN 253
  _A_PCD_SYSTEM_SHUTDOWN_DB 254
  _A_PCD_SYSTEM_REFRESH_DB 255  // nws
  _A_PCD_SYSTEM_PLEASE_ACK 256
  _A_PCD_SYSTEM_ACK 257
  _A_PCD_SYSTEM_TIME_SEND 258
  _A_PCD_SYSTEM_TIME_PRINT 259
  _A_PCD_SYSTEM_ENABLE_STREAMS 260
  _A_PCD_SYSTEM_DISABLE_STREAMS 261
  _A_PCD_SYSTEM_INTRODUCE_SERVER 262
  _A_PCD_SYSTEM_SERVER_SYNCH_DOWN 263
  _A_PCD_SYSTEM_SERVER_SYNCH_UP 264
  _A_PCD_SYSTEM_INIT_DB_V2 265
  _A_PCD_SYSTEM_NEW_MON_RATE 266
  _A_PCD_SYSTEM_FLUSH_LOG_REQUEST 267
```

Appendix C - Sample ADAMS Program - Database Schema

```
                        /* query_schema.src */

#include <stdio.h>

main()
        /*
            **   Define a schema for benchmarking retrieval
          operations
        */
        {
<<      open_ADAMS  job_id      >>


<<      location        instantiates_a STRING_ATTR
                        scope is TASK                   >>
<<      type            instantiates_a INTEGER_ATTR
                        scope is TASK                   >>
<<      manufacturer    instantiates_a STRING_ATTR
                        scope is TASK                   >>
<<      model           instantiates_a STRING_ATTR
                        scope is TASK                   >>
<<      settings        instantiates_a STRING_ATTR
                        scope is TASK                   >>
<<      install_date    instantiates_a INTEGER_ATTR
                        scope is TASK                   >>

<<      INSTRUMENT isa CLASS
                having attrs = { location, type,
                                 manufacturer, model,
                                 settings, install_date },

                scope is TASK                           >>


<<      INSTR_MAP isa MAP with image INSTRUMENT >>



<<      time            instantiates_a REAL_ATTR
                        scope is TASK                   >>
<<      temperature     instantiates_a REAL_ATTR
                        scope is TASK                   >>
<<      pressure        instantiates_a REAL_ATTR
                        scope is TASK                   >>
<<      observer        instantiates_a STRING_ATTR
                        scope is TASK                   >>
<<      description     instantiates_a STRING_ATTR
                        scope is TASK                   >>
<<      instrument_ref instantiates_a INSTR_MAP                 >>



<<      MEASUREMENT isa CLASS
                having attrs = { time, temperature,
                                 pressure, observer,
                                 description},
                having maps = { instrument_ref},
                scope is TASK                           >>

<<      SET_TYPE isa SET of MEASUREMENT elements,
```

```
                scope is TASK                              >>

<<      set1 instantiates_a SET_TYPE          >>
<<      set2 instantiates_a SET_TYPE          >>
<<      set3 instantiates_a SET_TYPE          >>

<<      close_ADAMS  job_id     >>
        printf ("set_of schema definition complete\n");

        }
```

Appendix D - Sample ADAMS Program - Database Setup

```
                /* query_setup.src */

  #include <stdio.h>
  #include <math.h>
  #include <sys/time.h>
  #include “wall_clock.h”

  #define NUMBER_MEASUREMENTS 25000
  //#define NUMBER_MEASUREMENTS 1000000

  #define MEASUREMENTS_PER_INSTRUMENT 250
  //#define MEASUREMENTS_PER_INSTRUMENT 5000

  #define NUMBER_MANUFACTURERS 3

  #define NUMBER_OBSERVERS 5

  #define NUMBER_LOCATIONS 100

  main()
          /*
          **  Given a persistent set, ‘set1’ populate it
          **  with elements so that
          **  and
          */
          {
          int     nbr_set1, nbr_set2, nbr_common, min_nbr, card,
num_this_instr;
          int     loc_index;
          int     i, temp_type;
          float   z;
          char    current_char[20];
          char    *loc_ptr;
          char    *man_ptr;
          char    *obs_ptr;
          char    *desc_ptr;

          static char    *manufacturers[NUMBER_MANUFACTURERS] =
                                  {“U-Measure-it Inc.”,
                                   “Measuring Devices R-Us Ltd.”,
                                   “Mr. Wizard’s”};

          static char    *observers[NUMBER_OBSERVERS] =
                                  {“Anonymous”,
                                   “Dr. Frederick Biggles”,
                                   “Jacques Cousteau”,
                                   “Timmy”,
                                   “is it on yet?”};

          static char    *locations[NUMBER_LOCATIONS] =
                                  {“Oak Ridge, TN”,
                                   “Charlottesville, VA”,
                                   “Carlsbad, NM”,
                                   “Medora, ND”,
                                   “West Glacier, MT”,
                                   “Baker, NV”,
                                   “Interior, SD”,
                                   “Beaumont, TX”,
                                   “Salt Flat, TX”,
                                  “Springdale, UT”,
                                   “Bryce Canyon, UT”,
                                   “Moose, WY”,
```

“Sedro Woolley, WA”,
“Bar Harbor, ME”,
“Key West, FL”,
“Death Valley, CA”,
“Three Rivers, CA”,
“Mammoth Cave, KY”,
“Twentynine Palms, CA”,
“Saratoga, NY”,
“Amherst, MA”,
“Medfield, MA”,
“Ventura, CA”,
“Middlebury, VT”,
“Grand Canyon, AZ”,
“Makawao, HI”,
“Gustavus, AK”,
“Fairbanks, AK”,
“Estes Park, CO”,
“Sulphur, OK”,
“Hot Springs, AR”,
“Pea Ridge, AR”,
“Republic, MO”,
“Lincoln City, IN”,
“Saint Croix Falls, WI”,
“Empire, MI”,
“Munising, MI”,
“Chillicothe, OH”,
“La Junta, CO”,
“Torrey, UT”,
“Tucson, AZ”,
“Scranton, PA”,
“Elverson, PA”,
“Washington, DC”,
“Rome, NY”,
“Titusville, FL”,
“Sullivans Island, SC”,
“Tupelo, MS”,
“Manteo, NC”,
“Richmond, VA”,
“Site 2, Oak Ridge, TN”,
“Site 2, Charlottesville, VA”,
“Site 2, Carlsbad, NM”,
“Site 2, Medora, ND”,
“Site 2, West Glacier, MT”,
“Site 2, Baker, NV”,
“Site 2, Interior, SD”,
“Site 2, Beaumont, TX”,
“Site 2, Salt Flat, TX”,
“Site 2, Springdale, UT”,
“Site 2, Bryce Canyon, UT”,
“Site 2, Moose, WY”,
“Site 2, Sedro Woolley, WA”,
“Site 2, Bar Harbor, ME”,
“Site 2, Key West, FL”,
“Site 2, Death Valley, CA”,
“Site 2, Three Rivers, CA”,
“Site 2, Mammoth Cave, KY”,
“Site 2, Twentynine Palms, CA”,
“Site 2, Saratoga, NY”,
“Site 2, Amherst, MA”,
“Site 2, Medfield, MA”,
“Site 2, Ventura, CA”,
“Site 2, Middlebury, VT”,
“Site 2, Grand Canyon, AZ”,
“Site 2, Makawao, HI”,

```
                                        “Site 2, Gustavus, AK”,
                                        “Site 2, Fairbanks, AK”,
                                        “Site 2, Estes Park, CO”,
                                        “Site 2, Sulphur, OK”,
                                        “Site 2, Hot Springs, AR”,
                                        “Site 2, Pea Ridge, AR”,
                                        “Site 2, Republic, MO”,
                                        “Site 2, Lincoln City, IN”,
                                        “Site 2, Saint Croix Falls, WI”,
                                        “Site 2, Empire, MI”,
                                        “Site 2, Munising, MI”,
                                        “Site 2, Chillicothe, OH”,
                                        “Site 2, La Junta, CO”,
                                        “Site 2, Torrey, UT”,
                                        “Site 2, Tucson, AZ”,
                                        “Site 2, Scranton, PA”,
                                        “Site 2, Elverson, PA”,
                                        “Site 2, Washington, DC”,
                                        “Site 2, Rome, NY”,
                                        “Site 2, Titusville, FL”,
                                        “Site 2, Sullivans Island, SC”,
                                        “Site 2, Tupelo, MS”,
                                        “Site 2, Manteo, NC”,
                                        “Site 2, Richmond, VA” };


<<      ADAMS_var x, result, current_measurement, current_instrument >>


<<      open_ADAMS  job_id      >>


        num_this_instr = 0;
        nbr_set1 = 0;
        loc_index = 0;

                                    /* First make sure they are empty */
<<      set1 <- NULLSET         >>

<<      result instantiates_a SET_TYPE >>


        while (nbr_set1 <   NUMBER_MEASUREMENTS )
                {

                nbr_set1++;

                if (((nbr_set1)%100) == 0)
                {
                     printf(“(%d)”,nbr_set1);
                     fflush(stdout);
                }



                if (((nbr_set1)%20000) == 0)
                {

<<      | card int | <- set1.cardinality             >>
                  printf(“>%d<“,card);

                }
```

```
                    if ((num_this_instr%MEASUREMENTS_PER_INSTRUMENT) == 0)
                    {
                        printf(“.”);
                     if (((nbr_set1/MEASUREMENTS_PER_INSTRUMENT)%40) == 0)
                         printf(“\n”);
                        num_this_instr = 0;
   <<                current_instrument instantiates_a  INSTRUMENT      >>
                        loc_ptr = locations[loc_index%NUMBER_LOCATIONS];
                        loc_index++;
   <<                  current_instrument.location <- | loc_ptr char* | >>
                        temp_type = loc_index/47;  //
   <<                   current_instrument.type <- | temp_type int | >>
                        man_ptr =
manufacturers[loc_index%NUMBER_MANUFACTURERS];
   <<               current_instrument.manufacturer <- | man_ptr char* | >>
                        obs_ptr = “A67FFF”;  // just a model number
   <<                   current_instrument.model <- | obs_ptr char* | >>
                        obs_ptr = “S:5 A:78 ON/OFF:OFF TRI:ORANGE Z:4”;
   <<                  current_instrument.settings <- | obs_ptr char* | >>
   <<               current_instrument.install_date <- | loc_index int | >>
                    }
                    num_this_instr++;

   <<               x instantiates_a MEASUREMENT    >>

   <<               x.instrument_ref <- current_instrument >>
   // We have a time range per instrument, 1 to MEASUREMENTS_PER_INSTRUMENT
   <<               x.time <- | num_this_instr int |    >>

                                          /* values in [0.0, 1.0] */
                                      /* uniformly distributed       */
                    z = drand48();
                    z = z * 100;   // want 0.00 to 100.00
   <<               x.temperature <- | z float |    >>
                    z = drand48();
                    z = z * 100;  // want 0.00 to 100.00
   <<               x.pressure <- | z float |       >>
```

```
                 obs_ptr = observers[num_this_instr%NUMBER_OBSERVERS];
<<               x.observer <- | obs_ptr char* | >>
                desc_ptr = "A short description of some 42 characters";
<<               x.description <- | desc_ptr char* | >>
<<               insert x into set1      >>
                 }
       fprintf (stdout, "Total elements instantiated = %d\n", nbr_set1);
         fflush (stdout);

                                        /* force creation of inverse    */
                                        /* attribute O-trees            */
         printf("beginning first inverse...\n");
<<       result <- { x in set1 | x.time < '2' }  >>
<<       | card int | <- result.cardinality              >>
         printf("Cardinality #1 is %d\n",card);
         fflush(stdout);
<<       result <- { x in set1 | x.time = '1' }  >>
<<       | card int | <- result.cardinality              >>
         printf("Cardinality #1A is %d\n",card);
         fflush(stdout);
<<       result <- { x in set1 | x.temperature < '0.1' } >>
<<       | card int | <- result.cardinality              >>
         printf("Cardinality #2 is %d\n",card);
         fflush(stdout);
<<       result <- { x in set1 | x.pressure < '0.1' }    >>
<<       | card int | <- result.cardinality              >>
         printf("Cardinality #3 is %d\n",card);
         fflush(stdout);
         loc_ptr = locations[1];
<< result <- { x in set1 | x.instrument_ref.location = | loc_ptr char*
| } >>
<<       | card int | <- result.cardinality              >>
         printf("Cardinality #4 is %d\n",card);
         fflush(stdout);


<<       close_ADAMS  job_id     >>
         }
```

Appendix E - Sample ADAMS Program - Database Query

```
                    /* the_query.src */

  #include <stdio.h>
  #include <math.h>
  #include <sys/time.h>
  #include “wall_clock.h”

  #define NUMBER_MEASUREMENTS 250000
  //#define NUMBER_MEASUREMENTS 10000

  #define MEASUREMENTS_PER_INSTRUMENT 2500
  //#define MEASUREMENTS_PER_INSTRUMENT 5000

  #define NUMBER_MANUFACTURERS 3

  #define NUMBER_OBSERVERS 5

  #define NUMBER_LOCATIONS 50


  main()

          {

          int     i, temp_type;
          int     start_time, stop_time;
          float   z, low_temp, high_temp, low_pressure, high_pressure;
          char    current_char[20];
          char    *loc_ptr;
          static char    *location = “Charlottesville, VA”;

          int card;

          wall_clock *the_clock;
          wall_clock *clock2;

          timeval total_loop_time, total_insert_time, total_synch_time,
temp_time;


  <<     ADAMS_var x, result, current_measurement, current_instrument >>


  <<      open_ADAMS  job_id      >>


          the_clock = new wall_clock();
          clock2 = new wall_clock();


          total_insert_time.tv_sec = 0;
          total_insert_time.tv_usec = 0;

          total_loop_time.tv_sec = 0;
          total_loop_time.tv_usec = 0;

          total_synch_time.tv_sec = 0;
          total_synch_time.tv_usec = 0;



  <<      result instantiates_a SET_TYPE >>
```

```
          printf("beginning query...\n");


   //        int     start_time, stop_time;
   //      float   z, low_temp, high_temp, low_pressure, high_pressure;

          // for 25,000 db, time runs from 1 to 250 evenly distributed


          start_time = 200;
          stop_time = 225;   // 10 percent

         // temp and pressure are floats running from 0 to 100, created
by drand

          low_temp = 20.00;  //
          high_temp = 40.00; // 20 percent

          low_pressure = 50.00;  //
          high_pressure = 100.00; // 50 percent

                                  // == 25,000 * .1 * .2 * .5 = 250

          loc_ptr = location;  // each location is 1/100 = 1 percent, so
          _A_Enable_Streams();
          the_clock->start();

   << result <- { x in set1 |
                | start_time int | <= x.time <= | stop_time int | and
                | low_temp float | <= x.temperature <= | high_temp float |
                        and
               | low_pressure float | <= x.pressure <= | high_pressure float |
                        or
                    x.instrument_ref.location = | loc_ptr char* | } >>

   <<      | card int | <- result.cardinality              >>

          the_clock->stop();

          printf("Cardinality #1 is %d\n",card);
          fflush(stdout);


          printf("Setup - time to query - ");

          the_clock->show_time();
          fflush(stdout);


   <<     close_ADAMS  job_id     >>
          }
```