Extensible High Performance Support for Persistence

David E. Langworthy

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-45
September 1993

ObServer2:

Extensible High Performance Support for Persistence

Ph.D. Thesis Proposal

David E. Langworthy

July 25, 1993

Abstract

In the late 80's new applications such as CAD, hyper-media, and programming environments stressed to the limit existing database technology [2]. To meet the demands of these new domains OODBs were developed. There was, and still is a great diversity in their architecture and functionality. Furthermore, this functionality continues to evolve [14, 22]. However, early on one common feature emerged and that is they all used some distinct service that provided stable storage for objects or pages, referred to as object servers and page servers respectively [13].

This paper describes problems facing the continued development of object stores and addresses them in the design of ObServer2. The design takes a new approach to the construction of an object server that allows it to meet changing needs. Also, it presents original contributions in the design of concurrency control and recovery algorithms.

New applications such as CAD, hyper-media, and programming environments stress existing database technology to the limit [2]. Object Oriented Databases (OODBs) were developed to meet the demands of these new domains. All OODBs use some distinct service that provides stable storage for objects or pages, referred to as object servers and page servers respectively [13, 4]. OODBs support new application domains with extensible type systems that allow new abstractions to be added easily. In contrast, the object storage layer is fixed or very difficult to modify. This causes performance problems whenever an OODB attempts to model a domain for which specialized secondary storage structures exist. It would be desirable to have an extensible object store that could accommodate new interfaces. Incorporating a new storage structure into an existing database implementation is a formidable task. In addition to implementing the storage structure, concurrency control, recovery and other modules would need to be changed.

The goal of this project is to build extensible, high-performance, persistent storage for objects in a distributed environment. Distribution introduces the problem of communication overhead and partial failure. Distributed commit protocols [11] guarantee correct operation in the event of partial failure, but under certain conditions all distributed commit protocols block until the failed server recovers. ObServer2 addresses the problem of failure with an original recovery algorithm that does not require any undos or redos while requiring reduced output during commit. Therefore, the server recovers quickly after a failure.

High performance in a distributed system requires strict control of I/O costs. The technology trends for both disk and network I/O are similar. Bandwidth is increasing in proportion to other system resources such as memory size and CPU speed. However, communication latency is already a bottleneck [10], and since it is not improving at the same rate as other system resources, it will continue to become more of a problem. ObServer2 is constructed from loosely coupled entities that cooperate asynchronously through well defined interfaces. Avoiding synchronous operations helps reduce communication overhead.

1 Contributions

The work described in this paper was motivated by several problems in existing database technology and distributed systems.

1. In a client-server distributed system, there is the problem of keeping cached copies of data up-to-date with respect to the primary copy at the server. This problem is particularly acute in database systems which provide strict correctness guarantees [10]. The communication cost of keeping these caches up-to-date becomes a performance bottleneck.

2. A traditional implementation of the most popular and efficient recovery mechanism in database systems, Write Ahead Logging, requires either a redo phase or an undo phase or both during recovery. This phase is expensive because updates must be applied to base versions. There has been a great deal of work to reduce the time required for this phase, but none has totally eliminated it.

3. Concurrency control relies on the physical layout of data in secondary storage. Interactions with concurrency control make it difficult optimize storage.

4. Multiplexing client requests is a complex component of a database system. A traditional implementation multiplexes all client requests and internal activities within a small number of operating system processes.

5. Testing new database concepts is difficult because database architectures are tightly coupled and difficult to modify. Simulations explore restricted parts of a system or a particular algorithm, but cannot explore the complexity of a large system to discover unexpected interactions.

The contributions of this work correct or alleviate these problems and provide insight into the nature of constructing a distributed database.

1. **Multiple Cache Coherency Policies**

   ObServer2 is the first database that does not depend on cache coherency for transactional correctness. This provides ultimate freedom in choosing a cache coherency policy. It is likely that for common workloads it is desirable to occasionally let the cache drift out of synchronization because communication costs will be greatly reduced. Reducing the cost of maintaining client caches makes it possible to cache large amounts of data on client for long periods of time on the clients local disk. Allowing the caches to temporarily drift out of synchronization allows tolerance of communication failure and may eventually allow support for disconnected, mobile operation.

2. **Efficient Recovery Technique**

   ObServer2's recovery algorithm improves on the popular and efficient Write ahead logging. ObServer's No-Undo, No-Redo Write Ahead Logging does not require any updates during recovery. This reduces recovery to a single scan of the log. The redos are applied on demand as the objects are read or asynchronously after recovery by the same background process that normally applies updates.

3. **Local Concurrency Control**

   The design of ObServer2 applies the theory of local concurrency control to separate concurrency control and storage management concerns [20]. Objects in ObServer2 are fully abstract with respect to the transaction manager. Objects can be implemented in different ways or offer different operations. This allows support for all existing database access methods as well as any that may be invented in the future. No other database system has applied this theory in this way.

4. **Novel Use of Threads to Achieve Effective Asynchrony**

   An ObServer2 server is structured as a collection of asynchronous threads. Coping with asynchrony in this way simplifies construction of the object store and allows for several of the optimizations described previously. For example each client will have several dedicated threads. One thread manages direct client requests and another does background work to anticipate the needs of that particular client. The updates of all committed transactions are applied to the stable store by another set of background threads.

5. **Experimental Testbed**

   The prototype implementation will act as an experimental testbed. This required the design of a very flexible database architecture which could be easily modified and extended. All of the previous contributions relate to this flexible architecture. A great deal of work went into designing strong abstraction boundaries and reducing restrictions on the implementation of storage structures and communications.

The experiments described here will shed light on the implementation of all database systems and offer insight into the construction of high performance database systems.

- Naming

  Our experiments with the original ObServer showed that object naming and location is an important performance parameter [5]. In the ObServer2 prototype, the storage class mechanism allows experimentation with alternative naming strategies.

- Caching

  Some work at Brown on intelligent, predictive caching algorithms has shown their theoretic importance and simulations have shown that they will have a practical application [16, 19]. There are a number of issues to be resolved in actually incorporating an intelligent, predictive cache in a database. The storage class mechanism allows experimentation with intelligent caching policies.

- Concurrency Control policies

  ObServer2 can exploit the semantics of a particular operation and the arguments to that operation to attain higher potential concurrency. Experiments will determine the correct level of abstraction for concurrency control.

2 Architecture

Early object servers and page servers offered support for multi-client, single-server operation. To allow more users, provide access to more data, and take advantage of advances in hardware technology servers in the future will be fully distributed, multi-server systems. Distribution increases the aggregate hardware resources available to the database, but it introduces a number of problems that make it difficult to capitalize on those resources.

In a distributed system communication overhead can become a bottleneck. Increasing network bandwidth alleviates the problem to a certain extent; however network latency is not improving proportionally. The cost of a small message per unit of information is significantly above that of a larger message. Another important cost measure is the round trip latency for synchronous communication which is the basis of many concurrency control and cache coherency algorithms. A synchronous communication consists of a remote call and return. Such communication is particularly expensive because the sending process must block until the communication completes. While the process is blocked it cannot accomplish any work.

Failure presents another problem in a distributed system where many components can fail independently. Upon failure of a component the rest of the system needs to continue processing. If the failed component is on the path of a synchronous communication, the sender will be blocked until the component recovers, i.e., the failure propagates to the sender. A solution to both latency and partial failure is to use asynchronous communication. Small asynchronous communications can be grouped into larger, relatively less expensive messages. Furthermore, no process waits for an asynchronous communication. Thus, a failure along the communication path does not propagate delays to other parts of the system. ObServer2 uses algorithms that significantly reduce synchronous communication.

Figure 1 shows three end user applications: a VLSI tool, a CAD tool, and a Hyper-Media application. Each of these is implemented using a client OODB, which in turn is implemented on top of ObServer2. Clients communicate

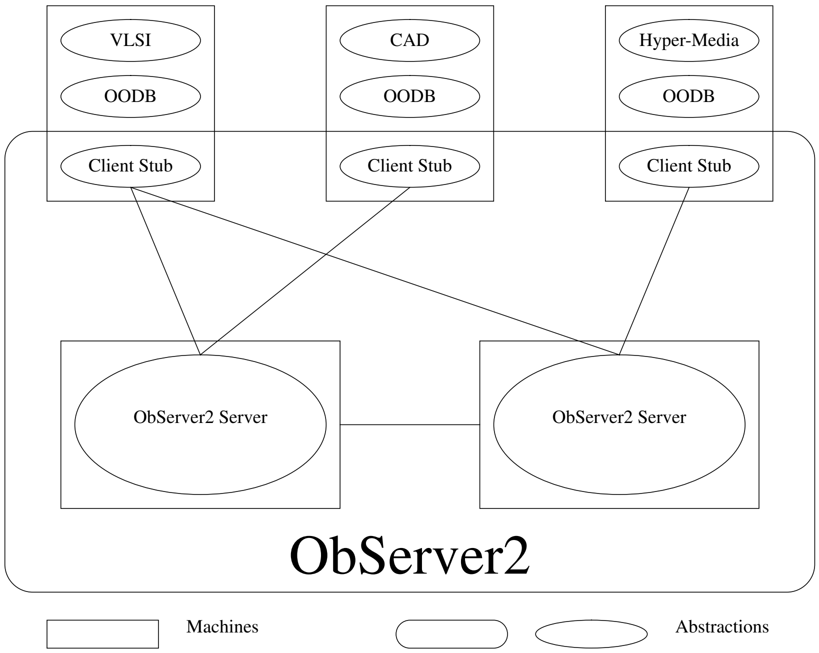


Figure 1:

with ObServer2 via a client stub. ObServer2's client stub runs in the address space of the client OODB. The protection boundary between the database and the application must be implemented by the OODB between it and the application. The OODB calls operations on the stub of a storage class. These operations can result in computation at the client or communication with the server. Not all operations will result in communication with the server. The client will communicate with the server once to fetch the object and once each time a transaction that read the object commits no matter how many times the object is read or how many transactions read the object.

In our computation model the client explicitly contacts each of the servers it needs information from and the servers communicate with each other only to synchronize for distributed commit. This model was chosen because ObServer2 does not aim to support general distributed computation. Its primary function is to provide persistent storage for objects.

ObServer2's internal interfaces are fixed and well defined, while the architecture allows the interface ObServer2 presents to client OODBs to be extended. ObServer2 allows new storage structures to be added to the system as a **storage class**. A storage class is a vertical slice through the layers in the memory hierarchy. Each piece of this slice has a well defined interface. The external interface it presents to the client OODB depends on the semantics of the storage class. Figure 2 shows a very abstract view of ObServer2 configured with two storage classes: a segment and a B-Tree. The rounded rectangles are Abstract Data Types (ADT's) that span the client and server. The rounded rectangle contains the operations which the ADT provides to the client OODB. Segments are comprised of any number of variable sized byte streams which support **fetch**, **read**, **write**, and **create**. B-Trees contain associations between values and objects and provide **insert**, **delete**, **find**, and **create** operations.

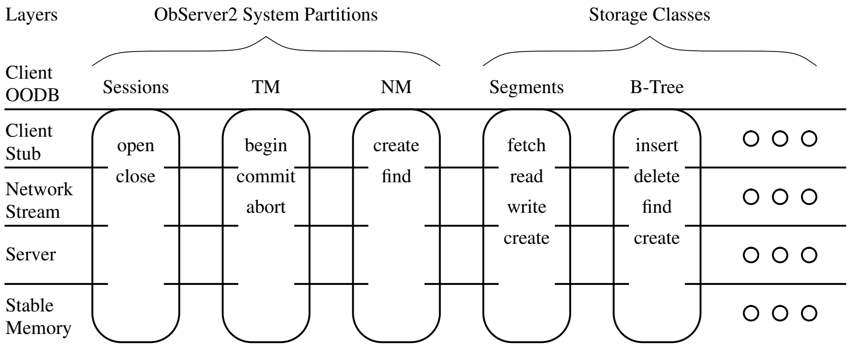


Figure 2: Layers & Vertical Partitions

Figure 2 also shows how ObServer2 vertically divides basic system services. There is a Session Manager (SM) which is responsible for maintaining connections to active clients and other servers. The Transaction Manger (TM) synchronizes the operations on all objects in all the different classes. The Name Manger (NM) allocates names that the classes use to identify objects. The basic system services are the same in all configurations of ObServer2, only the storage classes change.

3 Storage Classes

One of the major advantages of a storage class is that it gives the implementor control over how and where the state of an object is implemented. The base implementation of ObServer2 includes one storage class that implements simple objects that consist of a byte stream and an Object Identifier (OID). These simple objects provide adequate support for some applications. A new storage class could be implemented if an application needed to take advantage of a specialized secondary storage structure. As an example we can consider the B-Tree mentioned in Section 2. One way to implement the tree would be to send the pages of the tree up to the client on demand and perform the **insert**, **delete**, and **lookup** operations at the client. Alternatively, the operations could be sent to the server and performed there. The result would then be sent back to the client. Each option would perform best under some workload. ObServer2 allows both implementations to coexist.

A storage class contains ObServer2 objects. ObServer2 objects can take many forms. All objects are addressed by a unique Object Identifier (OID). This OID encodes the class of an object to prevent the class of an object from being altered. ObServer2 uses an optimistic concurrency control algorithm which allows multiple versions of the same object to coexist. An OID and a Version Number together identify a **copy** of an object. All copies of an object with the same Version Number and the same OID have the same state.

Each storage class defines its own type specific concurrency control. The advantages of type specific concurrency control are discussed at length in [21]. In short, higher potential concurrency can be achieved by taking advantage of the semantics of the various operations that can be performed on an object. Concurrency control and recovery are performed on a per object basis within a class. The class determines if each object was accessed correctly and the Transaction Manager determines if the entire transaction can commit. Recovery is also based on an object's semantics. The Transaction Manager logs the operations performed on an object, not the physical storage that was modified. Logging operations makes it possible to recover objects individually as discussed in Section 4.

There are, however, several important limitations to the storage class which keep ObServer2 from serving as a general purpose OODB. Objects store references to other objects in the form of an OID, but the operations on storage objects cannot follow these links. Storage classes are further restricted by the the fact that there is no type or inheritance hierarchy among storage classes.

Servers communicate only to synchronize, so one server cannot ask another server to perform an operation or for the value of an object. Therefore, the arguments to an operation on an object at the server must be either values or references to objects. For these reasons, the implementation of a storage class is concerned only with the representation of independent storage objects. The relationships between objects are exploited at the client by the OODB.

ObServer2 offers the storage class implementor alternatives to deliver the best performance to the clients of the storage class. There is a tradeoff between whether to perform an operation at the client on a copy of an object or to perform the operation at the server, and send the result to the client. Sending the object to the client has the advantage that the object can be cached. Performing the operation at the server requires that the object stay in the server buffers for a longer period of time and requires greater server CPU resources.

A storage classes' extensibility is achieved by the same sort of dispatch table used in a remote procedure call service. The difference being that the dispatch is integrated with transaction management. The dispatch table allows new storage classes to register operations it wishes to provide. When the database starts up, a storage class will register its operations with the transaction manager. In addition to the operations a class provides the class registers a conflict operation. The conflict operation takes two arguments, an OID and an operation:

```
bool Conflict(OID o, Object() *Op1);
```

The Conflict operation determines if Op could invalidate the committed state of o based on the semantics of the operation and the arguments to the operation [21]. The operation cannot access the state of the object or the state of objects to which the object under consideration refers, but the arguments to the operation are available. Conflict returns true if Op has been invalidated by some prefious operation in o's history.

4 Recovery & Persistence

Storage classes play a central role in this architecture. A key design decision that makes the storage class notion feasible is our approach to recovery. This approach is based on operation logging and asynchronous update of copies. Thus, a storage class does not have to constantly return cached object versions whenever a transaction commits.

Observer2 relies on the service of a persistence layer (i.e. the stable store) to achieve stability for data and transactions. This layer consists of two fundamental components. One is a memory-mapped log and the other is one or more partitions that contain the base versions of the data. Durability of modifications made by transactions is achieved by entering an intentions record in the log. It is the coordination of the log and the heap of base versions that provides full persistence.

ObServer2 introduces a recovery algorithm called No-Redo Write Ahead Logging (NR-WAL) as a way to reduce the overhead of recovery. Intentions lists are used to eliminate the undo phase of recovery [1]. There is no redo phase because NR-WAL does not require the current state of an object to be stored stably, so the stable state does not need to be recovered immediately after a crash. The most recent state before the crash can be reconstructed later on demand.

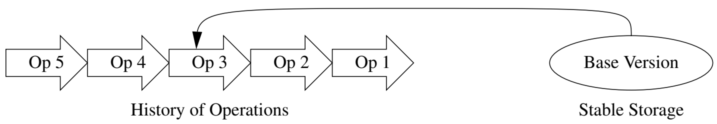


Figure 3: Components of the Current State of an Object.

The stable state of an object is composed of a **base version** and a sequence of operations called a **history**. Operations could be as simple as read and write or more sophisticated semantic operations such as insert, delete, and lookup. Operations are either reads or updates. A read operation does not change the state of the object.

Figure 3 shows the history of a series of update operations on an object. The update operations are stored in a list with the most recent operation at the head of the list. The base version contains a pointer to the last operation that was applied to the base version. During normal operation the base version can fall behind the current state. The object is brought up-to-date, by applying the appropriate operations from the history. In the figure, the base version is two operations out-of-date. Applying **Op 4** then **Op 5** brings the object up-to-date. The core of NR-WAL algorithm is based on a set of data structures that determine when to apply an operation to an object.

A further explanation of NR-WAL requires a discussion of our concurrency control algorithm. ObServer2 uses a timestamp based, optimistic concurrency control algorithm [12]. A transaction collects its operations in an **intentions list** [1]. During the transaction the operations are incrementally written to a sequential log in stable storage. Figure 4 shows the **Transaction Table** and two **Intention Records** in the log.

A record in the log is addressed by its **Log Sequence Number** (LSN). As records are added to the log, their LSNs strictly increase. Since the LSN of an operation is a strictly increasing value, it can be used as a timestamp for the

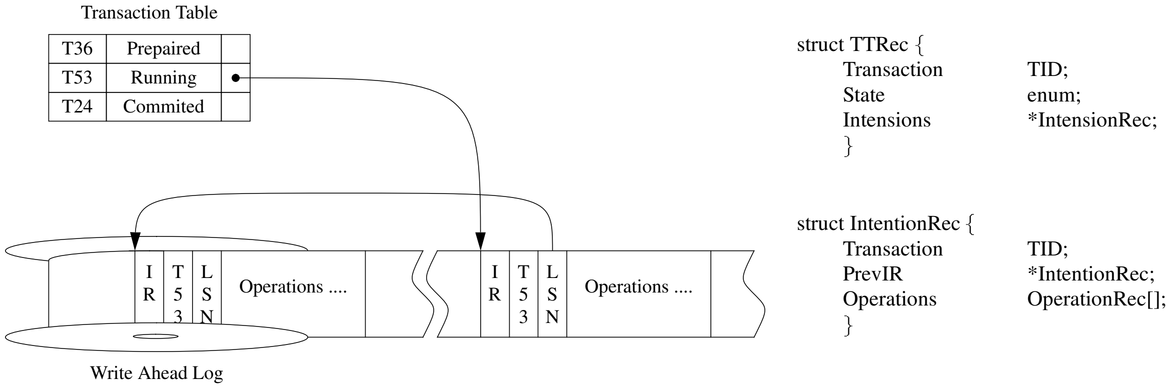


Figure 4: Chaining Intention Records

purpose of concurrency control. In this implementation, the LSN of the last applied operation denotes both the time of the last operation and its location in the log. To check for correctness, every operation in the intentions list contains the version of the object it should be applied to. The version "points" directly to the previous operation on the object. These links implement the history of operations shown in Figure 3.

An important recovery data structure, the **Version Table**, contains the most recent LSN for every cached object. It is the Version Table that indicates that operations 4 and 5 need to be applied in Figure 3. At commit, the transaction manager checks the operations in the intentions list against the Version Table. There is no conflict if the transaction read the most recent version of every object. If there is a conflict the transaction must abort. Every time an object is read by a client, the base version is checked against the entry in the Version Table. The object is up-to-date if they are the same. Otherwise, the Version Table "points" to the operations that need to be applied.

If the transaction spans multiple servers, a 2-phase commit algorithm is used [11]. The first and second phases of 2-phase commit modify the tables to indicate their current state. It is worth noting that updates do not lock the entire Version Table or Transaction Table at once. During the prepare phase the intentions list is forced to the log along with a prepare record. A small record is forced to the log to commit or abort the transaction. Committing an object requires no updates to the base versions of objects. These will be brought up-to-date either as needed or eventually by a background process.

Applying the appropriate operation to a copy of an object brings that copy up-to-date, but it does not actually modify the base version. It only modifies the copy in volatile memory. The base version is finally modified when the buffer manager at the server needs more space. The buffer manager keeps track of which objects have been modified and sends them to stable storage before it discards a volatile copy.

Both the history of operations and the base version of an object are stored stably and are not affected by a system failure. The Version Table and the Transaction Table must be updated frequently. Recovery is instantaneous if there is a small amount of non-volatile memory to hold these tables. The intention, prepare, and commit records in the log contain the information necessary to recover the tables after a crash. The tables are checkpointed periodically to reduce recovery time and to compact the log. Several optimizations keep the checkpoints small. There does not need to exist an entry for an object whose base version is up-to-date. Likewise, a transaction that is known to have committed or aborted at all sites does not require an entry. When a checkpoint is complete, its location is written to a well known location in stable storage.

The two volatile structures that ObServer2 depends upon for correctness are the Version Table and the Transaction Table. A failure will corrupt volatile memory and these structures will need to be recovered. The TM periodically checkpoints these structures in the log so that they can be recovered. The log implementation guarantees that there will always be at least one checkpoint in the log.

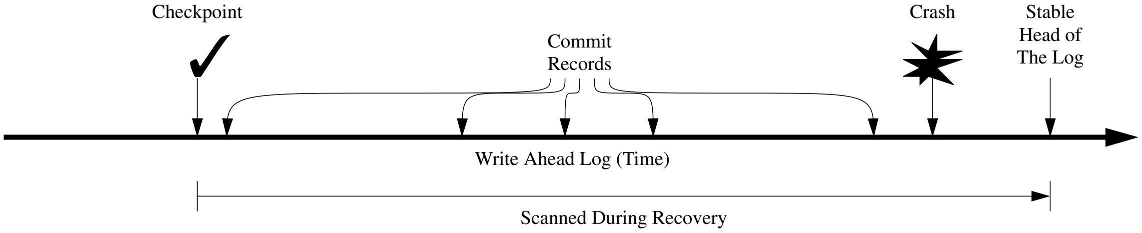


Figure 5: A Checkpoint & A Crash.

Figure 5 illustrates the portion of the log that is scanned during recovery. The crash indicates the position in the log that was being written as the failure occurred. The stable head of the log is kept ahead of the actual head of the log at a well known location in stable storage. The recovery pass begins at the last complete checkpoint and replays all the updates to the Version Table and Transaction Table. Normal operation begins upon completion of the scan. A volatile copy of an object that was lost is recreated on demand from its base version and history.

5 Examples

Two examples illustrate the interactions between the storage class and the transaction manager and the workings of the storage class itself. The first example follows the course of one transaction, **T**, through some of its significant events. The second follows one object, **O**, through a storage class. The example consists of a diagram with labeled arcs and a textual description. The labels in the diagram correspond to the labels in the text and the events in the text are given in the order that they occur.

1. **Begin:** Begin is the first message in a transaction. In response, a client stub creates a Transaction Identifier, **T**. Once **T** is created, the client OODB calls operations directly on the storage classes.
2. **Operations:** The operations that a client OODB calls on the client stub can be different from those that are actually sent to the server. Also, the number of operations sent may differ from the number actually invoked on the client stub. For example, the OODB can invoke many write operations and only one update will be sent to the server.

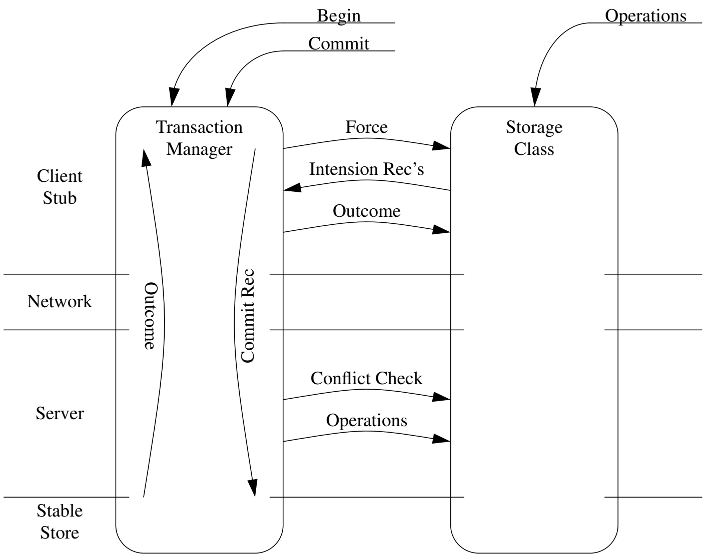


Figure 6: The Transaction Manager and a Storage Class

3. **Intention Rec's:** The operations of **T** are collected and sent to the server in an intention record. **T** has one or more intention records streamed to the server. Streaming reduces I/O at commit.

4. **Commit:** To save the modifications made during the transaction the client calls Commit on the transaction manager. The client transaction manager then causes a Force and sends a Commit Rec to the server transaction manager.

5. **Force:** In response to the force message, the storage class puts any of **T**'s remaining operations into an intention record. The Force blocks until this is complete.

6. **Commit Rec:** The final intention record is combined with a commit request for **T**. These are sent to the server and stored stably.

7. **Conflict Check:** At the server, the TM calls the Conflict operation for each object in the intentions records for **T**. Conflict returns true if **T**'s operation conflicts with a previously committed operation.

8. **Outcome:** Provided there are no conflicts, **T** is committed at the server. The client TM is notified of the outcome after a commit record has been forced to the log. In the event **T** must abort, the client stub must flush any objects **T** modified.

9. **Operations:** After **T** commits, its operations may be applied at the server. These operations are of a restricted form described in Section 3. Section 4 describes what happens if an object **T** modified is read before **T**'s operation is applied to it.

The next example follows the path of a simple object, **O**, as it moves through the different layers of a storage class. **O** is a simple object that supports only read and write operations.

1. **Read:** The first operation on **O** is a Read from the client OODB. A Read operation can only be called while the client OODB has a transaction running. Objects can be cached between transactions but for the purpose of this example, **O** is not cached at the client stub and the Read results in an object fault.

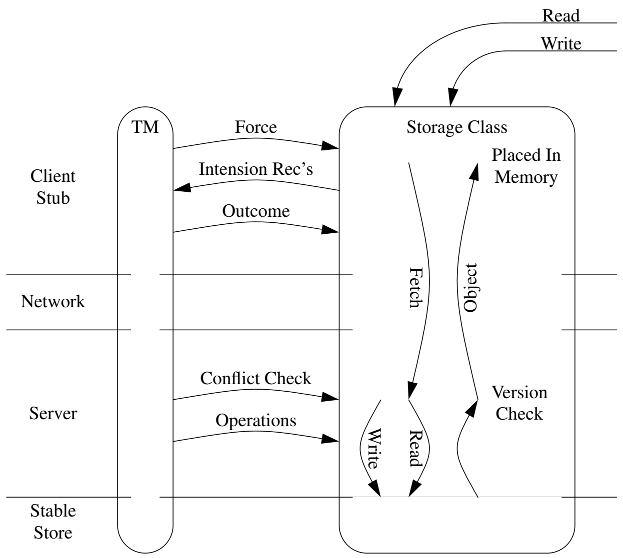


Figure 7: Inside a Storage Class

2. **Fetch:** To handle the fault on **O** a fetch is issued to the server.

3. **Read:** The server checks its cache for **O** and finds that it is not cached and issues a read to the stable store. The storage class uses its location table to determine where **O** is stored in the stable store and issues a read for the appropriate pages.

4. **Version Check:** After **O** is in the server memory the storage class checks that the version of **O** that was read from disk is the most recent version. If it is not, it is brought up-to-date as described in Section 4.

5. **Object:** This example is only concerned with **O** as it is sent back to the client stub. More than one object may be returned as a result of a fetch operation because of clustering and prefetching. It is up to the storage class to determine how many additional objects are sent to the client.

6. **Placed In Memory:** When **O** arrives at the client stub it is cached in the buffer pool for future operations.

7. **Write:** At a later point in the transaction the client OODB writes a new value into **O**. This change is now reflected in the clients buffer.

8. **Commit:** At some point later, the client transaction sends a commit message to the transaction manager.

9. **Force:** The transaction manager sends force to the storage class. The new value of **O** is placed in an intention record and sent to the server as described in the previous example.

10. **Conflict Check:** Before the transaction commits the TM in the server calls Conflict on **O** to determine if the update was invalidated. If so the transaction is aborted.

11. **Write:** Some time after the transaction commits, the new value of **O** is placed in server memory. The new value can either be sent down from the client or be reconstructed from the log. This does not have to occur immediately and once the new value is in server memory it does not need to be forced to stable storage. **O**'s new value is sent to stable storage when the buffer manager decides that it is no longer needed in server memory.

6 Memory Management

To facilitate the implementation of classes and to keep classes from competing with each other for buffer space ObServer2 provides a specialized memory management system. The system is specifically for management of objects within a particular layer in the memory hierarchy. For this reason this subsystem is called the Memory Object Manager (MOM).

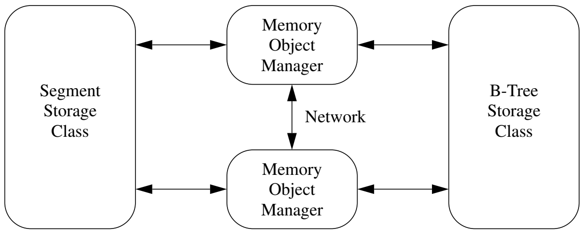


Figure 8: Two storage classes and two memory object managers (MOMs).

Each client and each server will have a MOM. MOM provides services to all of the classes within that layer of the system. Figure 8 shows an example system configuration. There are two storage classes: the Segments Storage Class and the B-Tree Storage Class. One MOM is on the client and the other is on the server. They are connected by the network.

A Memory Object (MO) is simply an arbitrary sized piece of memory identified by a Memory Object Identifier (MOID). A MOID is a handle to the MO that hides its actual location in virtual memory. The MOID is valid only within the MOM that created it. When a MOID is created there is no memory associated with it. Some time later when a storage class wants to use the MO it associates the MOID to some memory of a certain size with the Bind operation.

There are two other operations on a MOID: Pin and Unpin. The Pin operation returns the actual location of the object in memory. Since a MOID hides the location of an object in memory, a storage class can only operate on the memory when the object is pinned. While objects are unpinned MOM is free to move or discard them to make the best possible use of memory. Figure 9 a storage class calling Pin on a MOID.

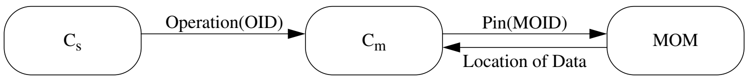


Figure 9: Cs, Cm and MOM.

The storage class is divided into two parts Cs and Cm. The "s" in Cs stands for semantics. Cs translates the abstract interface that the storage class provides to clients into the classes' internal operations. There is not a Cs portion of a class on the server. The "m" in Cm stands for memory. The Cm keeps track of memory for the class and has several other responsibilities. The implementation of Cm is class specific. In the most general case, Cm keeps a map from operations called on the class to their result stored in memory. For example in the segments class, Cm keeps a map from OIDs to MOIDs that acts much like a page table.

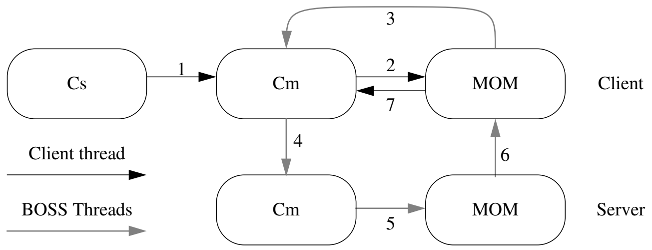


Figure 10: An example of a fetch.

Figure 10 shows the interactions between a segments classes' Cs and Cm and MOM. The example follows a fetch of one object as did Figure 7. This example only shows the fetch process and does not consider interactions outside of the class which were covered previously.

1. In response to a client read request on an object, Cs accesses the object in Cm.
2. Cm determines the MOID for the object in MOM, creating one if there is no entry in its map. Once it has the MOID it calls pin on the MOID.
3. In this case MOM does not contain the necessary memory object. The thread which attempted to pin the MOID blocks. In another thread, MOM calls an object fault handler in the Cm to retrieve the object.
4. The fault handler makes a call to the classes' Cm on the server to get a new copy.
5. The server Cm places a copy of the object in the server MOM.
6. The server MOM moves the memory object into the appropriate client MOM.
7. The thread that attempted to pin the MOID is restarted and the call returns.

In this example of an object fault the client thread blocks while ObServer2 synchronously retrieves the object from the server. ObServer2 also provides support for prefetching. Prefetching moves an object into the client's MOM before the object is explicitly requested. The advantage to prefetching is that object motion can be done asynchronously in the background. The client's synchronous fetch operation will not block because the object is already in the cache.

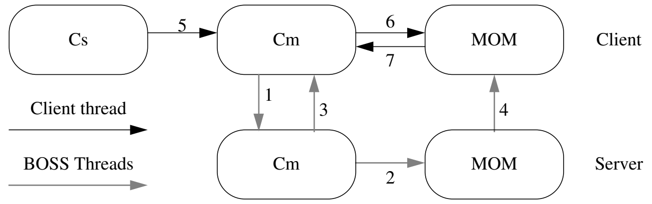


Figure 11: An example of a prefetch.

There are two different sorts of prefetching. Either the client Cm can request objects in advance of their use or the server can send objects before the client Cm has requested them. ObServer2 supports both models. The prior model is useful when an intelligent agent monitors the client's activity. An intelligent agent resides in the client Cm.

When it notices a pattern in the access behavior, it will request enough objects so that the client can complete the activity without blocking. The server sends objects in advance of a request when the objects are clustered together on secondary storage. A clustering algorithm will reorganize secondary storage so that objects that are used frequently together will be stored together. This a global optimization because all clients are effected by the clustering.

Figure 11 shows an example of the server sending clustered objects which have not yet been requested but probably will be soon. It only shows the movement of the prefecthed objects and does not show the fetch which initiated this action because that is the same as the previous example.

1. Before the prefetch begins the client Cm sends some MOIDs to the server. These MOIDs act as place holders in the client's memory so the server will have somewhere to put the prefetched objects.
2. When the server determines that it has objects that a client could use it instructs MOM to move the objects up to the client's MOM
3. The objects are moved up to the client's MOM just as they would be in a regular fetch.
4. The server informs the client which objects it sent up and the client Cm records this.
5. Sometime later the client requests the object from the Cm.
6. The Cm will notice that the object is already cached.
7. Finally the request will return without blocking.

Memory Object Managers give the storage classes a great deal of support without restricting their functionality. In addition to support for object motion, MOMs provide specialized buffer management policies so that once objects are brought into memory the best ones can be kept there. Where "best" can be determined by the class designer.

7 Concurrency Control & Recovery in Detail

Section 4 discusses the separation of concurrency control and recovery from the storage class and gives a basic outline of how it is accomplished in ObServer2. New data structures and algorithms allow ObServer2 to satisfy its goal of high performance

The key to performance in ObServer2 is asynchrony in both communication and processing. Data structures implemented on top of the log allow updates to be delayed. Out-of-date copies in the client cache are removed by sending bundles of invalidation messages asynchronously. An asynchronous, background process assures that updates are eventually applied and that transaction management runtime structures do not become excessively large.

7.1 The Database Partition & The Log

ObServer2's novel implementation of the database partition and the log allows it to delay writes to the stable store and recovery quickly after failure. Mapping the log directly into virtual memory and a unique implementation of the Log Sequence Number are the keys to this ability. The LSN and the memory provided by the log are used to build up the intention records that are used in both concurrency control and recovery. The database partition is stable storage that provides classes with extents of stable pages. The class builds the object abstraction on top of these pages.

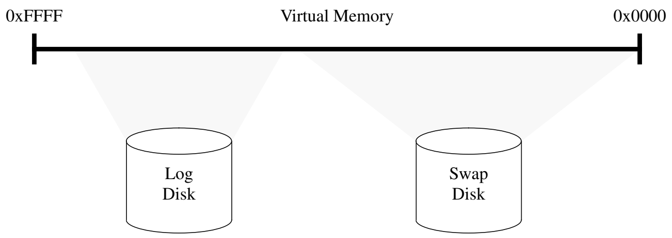


Figure 12: Mapping the log into virtual memory.

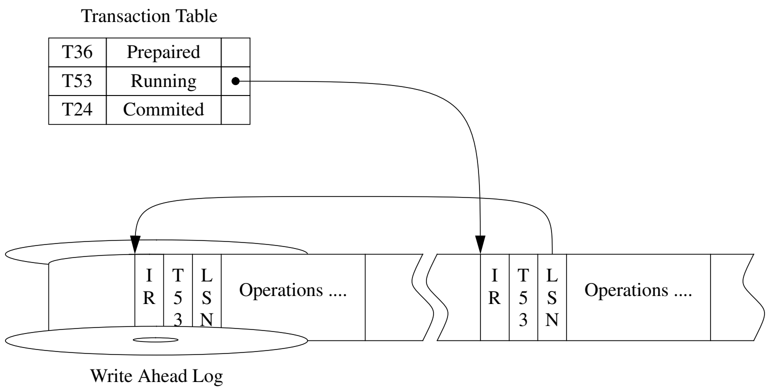
ObServer2 implements a circular log based on a raw disk partition mapped directly into virtual memory. Mapping the log keeps pages in the log from being swapped out by virtual memory (double buffering) without explicitly pining an unpinning each page (Figure 12). Since UNIX does not provide good support for this activity, this mapping provides both implementation and performance advantages. It also allows direct control of page layout.

The implementation of Log Sequence Numbers takes advantage of direct mapping to enable fast location of log records. An LSN is a tuple, where the low order bits encode a memory address in the log. Interpreted as an integer this number strictly increases as the log is written until the log wraps around. Our concurrency control algorithm [12] requires a time stamp that increases strictly over all time, not in a piece meal fashion. To correct this, the memory address is augmented with a count of how many times the log has wrapped around. This results in a LSN that consists of a 32 bit pointer and a 32 bit integer for 64 bits total.

Structures that will not fit in one record are built using linked lists. The log stores two collections of lists: the intentions lists and the update lists. An intentions list is rooted by a transaction's entry in the transaction table and the update list is rooted by an object's entry in the version table.

The version table serves several purposes. An entry in the version table consists of the LSN of the last operation performed on an object and a flag to support 2-phase commit [11]. Because this operation could have occurred in a previous wrap of the log, the entire LSN is stored. The version table must contain an entry for every object whose stable copy is not up-to-date. During the prepare phase of 2-phase commit the version table indicates whether an operation was invalidated by some transaction which prepared earlier. To accelerate prepare, the version table caches the LSNs of other objects in addition to those out-of-date with respect to the stable store. These entries can be discarded at anytime because they can be recovered by reading the object from the stable store.

The transaction table stores an entry for every active transaction. An entry consists of the LSN of the last record in the log and a flag to indicate the state of the transaction: running, prepared, committed, or aborted. Transactions store three sorts of records in the log: **Intention**, **Prepare**, and **Commit**. An Intention Record begins with a tag that is followed by a transaction identifier and a pointer to the previous intention record for this transaction (See Figure 8). The remainder of the record contains operations to be performed on objects.




```
struct TTRec {
    Transaction     TID;
    State           enum;
    Intensions      *IntensionRec;
    }

struct IntentionRec {
    Transaction     TID;
    PrevIR          *IntentionRec;
    Operations      OperationRec[];
    }
```

Figure 13: Chaining Intention Records

Operation records are stored inside of intention records. An operation can be either a read or an update. A read operation does not change the state of the object. The operation record stores the OID and the LSN of the copy that was read. An update operation changes the state of the object. So, in addition to the OID and the LSN of the object there need to be some way to transform the old state in to the new state. This can take the form of an operation, a new value, or a delta. ObServer2 does not restrict this choice. The choice of how to derive the new value from the old value and what to log is made on a class by class basis.

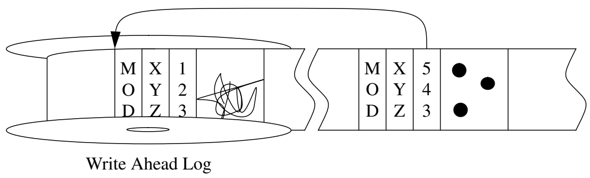


Figure 14: Operations in the Log.

7.2 Prepare & Commit

In ObServer2 transactions are serialized in they order they prepare. This is the instant which the state of the database must be consistent with that that the transaction observed. Prepare and commit do not bring the entire database into a consistent state. That is done sometime later by a background process.

Figures 6 and 7 show the basic flow of information and control during a transaction. The 2-phase commit algorithm described in [11] describes how to synchronize the servers. This section describes what happens within the phases.

Before a client transaction manager sends a commit record to the server it has sent all the intention records. By the time the server transaction manager begins to prepare a transaction all of the operations the transaction performed will be at the server.

Prepare

if Conflict

In the prepare phase the server transaction manager uses the Conflict operation (Section 4) to determine if any operation from the transaction's intentions list was invalidated by a transaction which prepared earlier.

then Abort Transaction

The transaction aborts and its intention records are ignored. ObServer2 uses a presumed abort variety of 2-phase commit that does not require an abort be recorded stably. The records are not discarded because that might mean rewriting a portion of the log. The space will be reclaimed by a log cleaning process (Section 7.4).

The client is notified of the abort and must flush all objects the transaction modified. In addition, the client is notified of copies that are no longer up-to-date due to an update by another transaction, so it can flush those too and avoid future aborts.

else Update tables and force a prepare record.

The transaction manager forces a prepare record to the log, updates the transaction table, and updates the version table. All three of these actions occur as an atomic operation with respect to other prepares. The current implementation allows many transactions to commit concurrently within a server. However, only one transaction prepares at a time, so atomicity is achieved by default. The transaction table needs to reflect the current state of the transaction and hold the LSN of the prepare record. The version table needs to contain the LSN of the most recent prepared operations for each object. If the transaction is distributed over multiple servers, the version table will also note that the modification has only prepared and not yet committed. The prepare record contains the LSN of the transaction's last intention record, the transaction identifier, and a timestamp used by the 2-phase commit process (Figure 15).

endif

End Prepare

Commit

Committing a transaction requires modification of the tables, but does not require atomicity. At commit the version table is modified to reflect that the operations are now committed. The transaction table notes that the transaction has committed and stores the LSN of the commit record. The commit record contains only the transaction identifier and the LSN of the prepare record.

After the transaction commits, the client is notified. The notification contains the LSNs of the modified objects. At the time of commit the client has dirty copies of objects. When the LSN of the latest modification is stored in one of these dirty objects it becomes a clean copy of a new version of the same object.

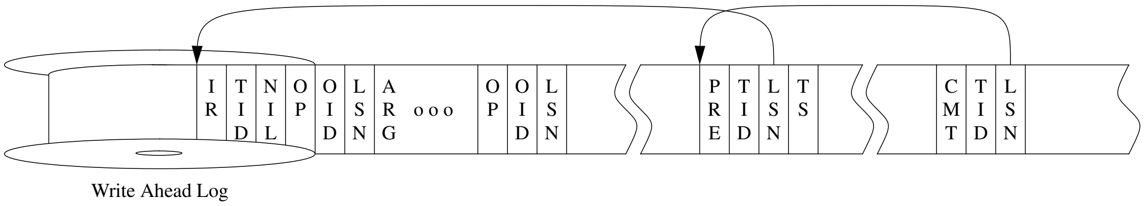
End Commit

Abort

If a transaction aborts in the second phase of 2-phase commit, the version table needs to be reverted to its previous state and the transaction table needs to be updated. An abort record is forced to the log.

The client is notified that it must flush all modified objects

End Abort




```
struct PrepareRec {
      Transaction        TID;
      LastIR             *IntentionRec;
      Timestamp          TS;
      }

struct CommitRec {
      Transaction        TID;
      Prepare            *PrepareRec;
      }
```

Figure 15: A Complete Simple Transaction in the Log

When the server receives an Intention Record, it assigns it an LSN and places it in buffers that are eventually forced to the log. Placing an Intention Record in the log does not cause the log to be forced immediately because it is an asynchronous operation. Only Prepare and Commit Records are synchronous and force the log. The buffers that hold the intention records are not pinned, so they are written to disk earlier in case there is a need for more buffer space.

Figure 16 shows an example system with a single class and two clients. It follows the evolution of an object X through the stages of a transaction.

Stage 1: The state of X is consistent through out the system. All copies of X are of version 123.

Stage 2: A client modifies a copy of X. It now contains a dirty copy of version 123.

Stage 3: When the transaction commits, a modification record is placed in the log at position 254. The version table is updated to indicate that the most recent version of the object is 254.

Stage 4: After commit, X's new LSN is propagated back to the client. The client now has a clean copy of version 254 rather than a dirty copy of version 123.

At the end of the transaction several copies of X that are not the current version still exist. ObServer2 allows this situation to occur and handles it in other parts of the system. The following sections describe how the other copies are brought up-to-date.

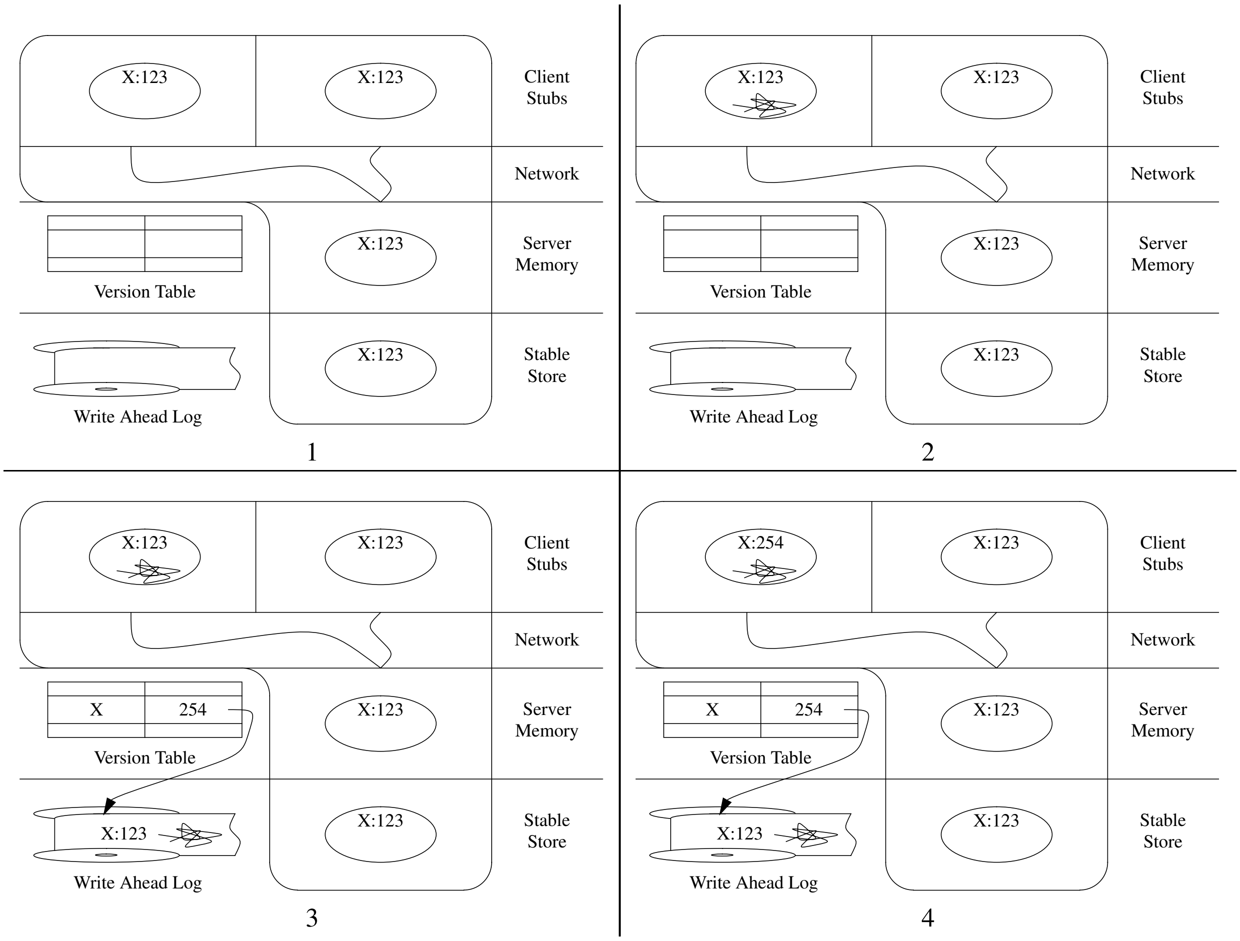


Figure 16:

7.3 Cache Coherency

Cache Coherency is very important for high performance, but in ObServer2 it is not required for correctness. This might seem to be either a contradiction or completely irrelevant. However, it is an important design feature. Each ObServer2 storage class is free to implement whatever cache coherency policy is best for its semantics and individual performance characteristics without regard to correctness or the policies implemented in other storage classes.

Copies of an object can fall out of date with respect to committed state. During normal operation a client fetches some objects into its cache and operates on those objects for some time. To avoid an abort, it is important to observe the most recent state of an object. This may require bringing the object up-to-date before a read completes. A client uses the fetch operation to request a copy of an object from the server. The pseudo code for the fetch operation at the server follows.

Fetch(OID)

Locate Object in Buffer Pool

Each class implements the best object location system for the sort of objects it supports. This step locates the object in the buffer pool. If it is not in the pool when the step begins it must be read from disk.

Apply(Version_Table(OID).LSN, Copy)

Apply is called on every object before it is sent to the client. The Apply operation checks that the copy of the object has same LSN as that passed in. Otherwise it brings the copy up-to-date.

Move Up Copy to Client Buffer Pool

The up-to-date copy is moved up to the client

End Fetch

Apply(Lsn, Copy)

if Lsn $\neq$ Copy.Lsn

If the copy has the same LSN as the argument then the corresponding operation in the log has already been applied to the object.

then Apply(Lsn.Lsn, Copy)

Otherwise, there is at least one operation that needs to be applied to the object. Apply recurses to determine if there are other operations that need to be applied. Each modification in the log contains the LSN of the previous operation. The arguments to the recursive call are the copy and the LSN of the previous operation. This is found by dereferencing the LSN to the operation record in the log and taking the LSN stored there.

The Apply operation replays history against the copy. The operations in the log are sent to the copy in the serial order determined by the transaction manager. When Apply returns, the copy is ready for the modification corresponding to Lsn. After the modification is sent to the copy, Copy.Lsn will equal Lsn.

endif

End Apply

Once an object is cached at a client, it is read or written by a transaction without additional communication. Correctness is guaranteed by a timestamp that is carried along with each object. ObServer2 informs client caches of updates asynchronously. In ObServer2, cache coherency is not required for correctness because each object is tagged with a version number. In many systems cache coherency is required for correctness and the required synchronous communication becomes a bottleneck [10]. This allows ObServer2 to construct larger messages containing more updates and make more efficient use of the network.

ObServer2 informs the clients of updates by periodically broadcasting recent updates to the Version Table. Sending recent changes to the Version Table requires little work at the server, especially if there is a broadcast or multicast communications protocol available. The clients can quickly process the table to determine if any objects need to be flushed or refetched.

7.4 Background Processing

The architecture of ObServer2 pushes as much processing into the background as possible. Surprisingly the two background processes are quite simple. One process applies updates in the log to the database partition and the other sends update notification to the clients.

There are actually several process which perform this update notification process. There is one update notification process per storage class, since each class may wish to handle this task in a different way. The basic function of these different process is the same. One possible implementation of the update process would select the relevant updates from the version table and send them to the appropriate clients resulting in a cache invalidation scheme. Another implementation could select relevant updates from the log and forward those resulting in a update propagation scheme.

Several issues have yet to be resolved with the design as it is presented thus far. The circular log could fill up, causing transactions to be aborted. The Version Table could become so large that it infringes on the buffer pool and needs to be paged. The number of updates that need to be applied to an object before it is sent to a client could grow large and cause delays. These are all related problems. The Version Table only needs to keep an entry for an object whose stable state is not up-to-date with its committed state; therefore, no work needs to be done to the object on a read if there is not an entry in the Version Table. Operations must be kept in the log until they are applied to the stable store. An object whose stable state is out-of-date is potentially consuming log space that could be reclaimed.

The Application Process corrects these problems by applying operations in the log to stable storage. It chooses which operations to apply based on an Application Pointer. The Application Pointer is an LSN that satisfies the invariant that any operation with a lower LSN is reflected in stable storage. The Application Process scans forward from the Application Pointer looking for the oldest operation on an object with an entry in the Version Table. If there is an entry in the Version Table, it is possible that this operation has not been applied to stable storage. It then checks the LSN of the object and brings it up-to-date if necessary. When the object has made it back to stable storage, the Application Pointer moves forward.

Applying the operation reduces the amount of work that needs to be done on a read and frees that slot in the Version Table correcting two of the above problems. The third problem is also corrected because the Application Pointer is an upper bound on the tail of the log, so by moving the Application Pointer forward, space in the log is effectively freed up.

Figure 17 continues the example of an object in the server. The previous part of the example left off when the client that modified X had been informed of X's new LSN, 254. This example follows the new copy of X to the other clients into the stable store and out of the log.

Stage 5: The server broadcasts recent modifications to the version table to clients. When the second client notices a change to X, it has a choice of bringing X up-to-date or discarding it. In this case the client chose to discard the

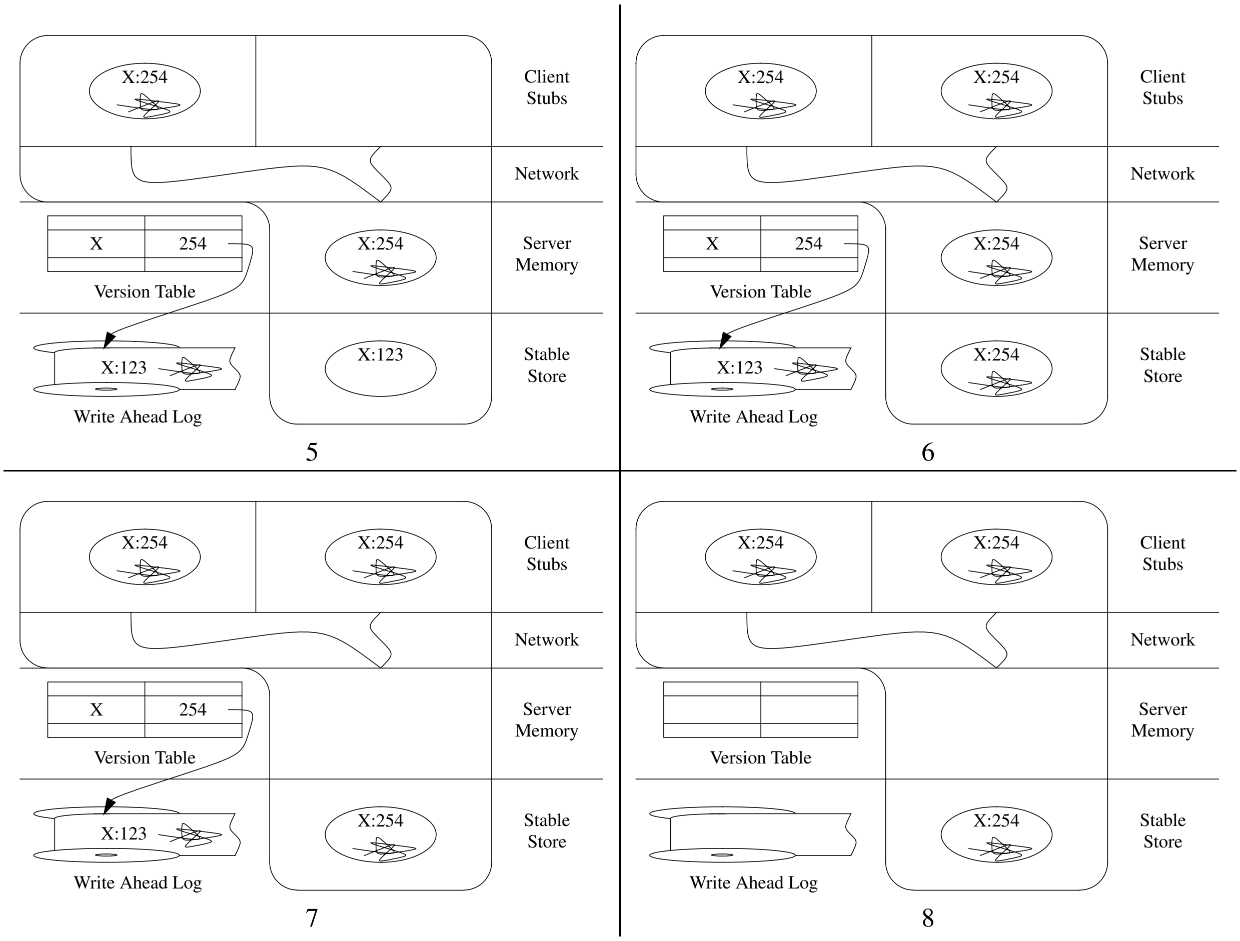


Figure 17:

out of date copy and request a new copy from the server.

Stage 6: The second client calls fetch on the server. The server updates its copy of X. The buffer manager notes that the copy has been updated and moves that copy into the cache of the second client. Now both clients have up-to-date copies and the server cache has an up-to-date copy, but the database partition still has an out-of-date copy. This is not a problem in ObServer2.

Stage 7: The stable copy is brought up-to-date when the buffer manager at the server decides that the space could be put to a better use. The buffer manager knows that this copy is more recent than the copy in stable storage, so rather than discarding it, it writes that copy back to the store.

Stage 8: Now that the stable copy is up-to-date the version table is not required to keep an entry for X and the operation record in the log can be discarded.

8 Related Research

This section compares ObServer2 to other object and page servers, extensible databases, and distributed file systems. There are a number of other systems that serve objects or pages. These include Mneme [15], ESM [4], and the original ObServer [13]. Postgres [18] and Starburst [9] are two extensible relational database systems. Object servers draw from file system technology for persistent storage and cache coherency. The section closes with a comparison between ObServer2 and the Andrew File System [8], the Harp replicated file system [7], and a Log Structured File System [17].

8.1 Object & Page Servers

The basic service ObServer2 provides is transactional support for persistent objects in a distributed client-server environment. There are two different sorts of systems which provide this service, page servers and object servers. Both of these systems differ from traditional database applications in that the data is moved into a client's work space, operated on for some time, and then returned. A traditional database sends an operation to the database, the operation is performed locally, and then the result is returned.

Page servers act much like a file system. Pages are provided to clients and updates are performed directly on those pages. An advantage of such a system is that a data reference is a direct address and locating the appropriate page is very simple. Since the page is a fixed size unit, it is very simple to move it from the server to the client. Page servers require that applications implement object abstractions on top of the raw pages which leads to some problems. Since a reference to an object is a direct address it is difficult to grow or move an object. Also, concurrency control is based on the physical layout of the data, not on the semantics of the operation that read or modified the data.

Object servers correct many of these problems by mapping the objects directly into storage objects. Concurrency control can be done at the object level, so updates to distinct objects do not cause false aborts. Further, an object store takes advantage of semantics of operations, so that concurrent operations which do not conflict semantically do not cause unnecessary aborts. In an object store the identity of an object is fully independent of its value, size, or location, so that an object can grow or move easily. It is easier to move an object in an Object store because object identity is independent of object location. This comes at the cost of added complexity in dereferencing an object.

ObServer [13], the precursor to ObServer2, introduced several features: semantic clustering of related objects, a novel lock set for cooperative work, and a notification system which also supports cooperative work. ObServer2 improves on ObServer in the areas of distribution, extensibility, and performance. ObServer offered only multi-client, single-server distribution. There was some work done to tie together multiple servers, but it lacked correctness guarantees. The storage class facility allows ObServer2 to evolve to handle new application domains. Much of the functionality that ObServer offered is encapsulated in one storage class within ObServer 2. The ability to add new storage structures allows ObServer2 to be specialized for high performance on specific applications.

The Exodus Storage Manager (ESM) [4], a page server developed at Wisconsin, provides support for values and indices. It generalizes the Aries Transaction Recovery System [3] to operate in the client-server model. Aries uses the concept of a Log Sequence Number to reduce the time that a lock must be held and facilitate recovery in a pessimistic system. The Aries implementation of a Log Sequence Number is different from that presented here, but the concept is basically the same. The LSN is a strictly increasing address into the log. The ObServer2 implementation allows faster access to a record in the log by using memory mapping.

Wisconsin simulations show that caching locks across transactions significantly improves performance. The benefit of lock caching is that clients can keep data across transaction boundaries and access that data without having to suffer a round trip communication delay to the server. The ObServer2 algorithm allows clients to keep objects for an indefinite amount of time without accessing the server. ESM requires clients to remain in contact with a server in order to keep data cached. As clients perform more optimizing transformations on data in the cache and architectures become heterogeneous, caching for periods longer than a single session with a server becomes more of an advantage.

More recent simulations confirm that caching for long periods of time on the client workstation can improve performance [**?**]. The ObServer2 storage class mechanism serves as a test bed which allows new memory management policies to be experimented in the context of an operating database.

The Mneme Persistent Object Store [15] explores the integration of an object store with a persistent programming language. It provides a persistent heap of objects that are available in a distributed system. The object model for Mneme is fixed, but it does offer some support for specialized buffering policies. A general policy is called a strategy and an instantiation of a strategy is called a pool. For example, a strategy might be LRU paging and a specific pool might be an LRU cache with 2000 elements. Every object belongs to a pool and this pool can change.

The strategy intends to provide specialized support for different classes of objects as does the ObServer2 storage class. The storage class is more powerful than the strategy in many ways. The storage class provides not only different buffering policies, but different storage structures and object location methods. Perhaps the most important difference is that an ObServer2 storage class actually extends the ObServer2 interface. Completely new access methods can be added to the object store. These new access methods can have different abstract interfaces and ObServer2 takes advantage of the concurrency semantics of this interface.

Mneme is intended to be a layer between an object store and an OODB. It does not provide recovery. Mneme semantics could possibly be provided by on storage class in an ObServer2 configuration. This would provide the benefits of a Mneme system and in addition recovery and the availability of other storage classes for specialized purposes.

Camelot [6] is another system that could be classified as a page server, but with a slightly different computation model. Camelot provides the abstraction of persistent virtual memory to data servers that run on the same node as the persistent data is stored. A Camelot server node must perform all disk I/O and computation associated with a transaction. The ObServer2 model allows clients to operate on remote machines. A specialized server can handle all disk operations. Client workstations with large main memories and fast CPUs cache their data and perform computation locally instead of using an RPC call to a remote server for every computation.

Camelot offers a several choices for logging. A data server can choose either local or remote logging. Each transaction can choose either new value, old value or new value only logging. Recovery is facilitated by LSNs somewhat similar to ObServer2. Camelot uses a structure called “the grid” that is similar to the update and intentions list in ObServer2. However there is a major difference. The Camelot “grid” an index structure stored in volatile memory and must be reconstructed after a failure. The update and intentions lists in ObServer2 are stored persistently in the log and do not need to be recovered after a failure.

The Avalon [6] language uses Camelot to provide a persistent C++. Because Camelot provides only persistent virtual memory, the semantics of the objects implemented in Avalon are lost at the persistence layer. ObServer2 captures semantics within the persistence layer to allow for special memory management policies or type specific concurrency control.

8.2 Extended Relational

Starburst [9] extends relational databases in five areas, external data storage, storage management, access methods, abstract types, and complex objects. Many of the issues that Starburst address are at the data model level and should be compared to an OODB, not to an object store. Abstract types, complex objects, and most of the access methods fall into this category. The one category of access method that is comparable within ObServer2 are what Starburst calls “exogenous” access methods. This name refers to how Starburst adds new functionality. It allows specialized components to be added “on the side” of a conventional database system. These components will be “called as they are needed directly from within components of the primary system.” “An ‘exogenous’ access method is one for which the data structure used to store access information is not managed by the primary database system.”

This differs from the ObServer2 architecture which is designed to add new functionality in a new storage class. In ObServer2 the storage class can define its own storage layout and access methods as Starburst allows. In addition, an ObServer2 storage class can define its concurrency control semantics at the level of the operations it supports rather than at the level of physical storage. We exploit the concurrency semantics of these operations. A new ObServer2 storage class actually extends the ObServer2 interface with a new ADT. The storage class implementor has exact control over whether parts of the abstraction are built up at the client or the server.

The goals of the Postgres Storage Manager are instantaneous recovery, historical access, and utilization of new technology [18]. Postgres provides a linear history and allows more than one version of an object to be viewed in a transaction. Since Postgres keeps the entire history of all its data, the recovery system differs from conventional systems. It provides instantaneous recovery without using conventional write ahead logging (WAL) or disk shadowing.

Their algorithm requires large amounts of Non-volatile main memory to perform as well as WAL for normal processing. ObServer2's NR-WAL performs as well as WAL during normal processing and offers instantaneous recovery without the requirement of large amounts of non-volatile memory.

8.3 File Systems

A page server is very close to a file system with added support for transactions. Two interesting developments in file systems that influence ObServer2, the Sprite Log Structured File System (LFS) and the Andrew File System (AFS).

LFS is interesting because it shows the feasibility of reading from a log given a reasonable cache. ObServer2 uses this facility to delay writes to the stable store. LFS [17] keeps all data in a log, significantly reducing the cost of a write. The disadvantage of a log is that it cannot be accessed quickly. A large RAM cache contains all the access structures to find data in the log and the most recently used data. A major overhead cost in LFS is keeping the file location information stable because it is constantly changing. In an object store the problem would be even worse because of the finer granularity of naming. Also, garbage collection is much more complex than segment compaction.

AFS is interesting for its distribution properties. AFS 3.x [8] is a distributed file system that serves a similar purpose to Sun's NFS. However, the scale of the distributed system that AFS can operate within is much larger than Sun's NFS. NFS was meant to operate within a single organization, while AFS can handle global distribution. One interesting feature of AFS that ObServer2 shares is non-volatile client caching. Data can be cached on a client's local disk for extended periods of time. Cache coherency becomes a problem when caching data for long periods of time. AFS introduced **call back locking** which caches locks as well as data at clients. Franklin [10] shows that this is a desirable protocol for database systems as well. Failures cause problems with consistency when locks are cached. AFS solves this problem by using locks that expire after a well known period of time. ObServer2 does not require that the cache be kept perfectly coherent. The version number associated with each object will indicate whether the object is up-to-date, so ObServer2 can tolerate failures with out any special means.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurreny Control and Recovery in Database Systems*. Addison Westley, 1987.

[2] Toby Bloom and Stanley B. Zdonik. Issues in the Design of Object-Oriented Database Programming Languages. In *OOPSLA*, pages 441–451. ACM, October 1987.

[3] et al. C. Mohan. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1), 1992.

[4] Michael J. Carey et al. Object and file management in the EXODUS extensible database system. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 91–100, Kyoto, Japan, August 1986.

[5] G. Delott. Performance improvements in the observer object server. Masters thesis, Department of Computer Science, Brown University, 1989.

[6] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon*. Morgan Kaufmann, 1991.

[7] B. Liskov et al. Replication in the harp file system. In *13th ACM SOSP*, 1991.

[8] J. H. Howard et al. Scale and performance in a distributed file system. *ACM TOCS*, 6, 1988.

[9] P. Schawarz et al. Extensibility in the starburst datbase system. In *Intl. Workshop on Object-Oriented Database Systems*, 1986.

[10] M. Franklin and M. Carey. Client-server caching revisited. In *Int'l Workshop on Distributed Object Management*, Edmonton, Canada, 1992.

[11] J. Grey. Notes on database operating systems. In R. Bayer, editor, *Lecture Notes in Computer Science*. Springer Verlag, 1978.

[12] M. Herlihy. Apologising versus asking permission: Optimistic concurrency control for abstract datatypes. *ACM Trans. on Database Systems*, 15(1):96–124, March 1990.

[13] M. F. Hornik and S. B. Zdonik. A shared, segmented memory system for and object-oriented database. *ACM Transactions on Office Information Systems*, 5(1):70–85, January 1987.

[14] B. Liskov. A highly available object repository for use in a hetrogenous distributed system. In *Persistent Object Systems Workshop*, September 1990.

[15] E. Moss. The mneme persistent object store. Technical Report TR 89-107, COINS, University of Massachusetts at Amherst, October 1989.

[16] M. Palmer and S. Zdonik. Fido: A cache that learns how to fetch. In *VLDB*, 1991.

[17] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *13th SOSP*, 1991.

[18] M. Stonebraker. The design of the postgres storage system. In *Proc. 13th VLDB*, 1987.

[19] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. In *ACM FOCS*, 1991.

[20] W. Weihl. *Concurrency Control For Abstract Data Types*. PhD thesis, MIT, 19??

[21] W. Weihl and B. Liskov. Implementation of resilient atomic data types. In *ACM Transactions on Programing Languages and Systems*, April 1985.

[22] Stanley Zdonik and Gail Mitchell. ENCORE: An Object-Oriented Approach to Database Modelling and Querying. *IEEE Data Engineering Bulletin*, 14(2), June 1991.