Storage Class Extensibility in the Brown Object Storage System

David E. Langwoprthy and Stanley B. Zdonik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-22
May 1994

Storage Class Extensibility in the Brown Object Storage System

David E. Langworthy
Stanley B. Zdonik

Department of Computer Science
Brown University

New applications such as CAD, hyper-media, and programming environments stress existing database technology to the limit [1]. 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 [9, 4, 12, 6, 7]. OODBs support new application domains with extensible type systems that allow new abstractions to be added easily. This causes problems whenever an OODB attempts to model a domain for which specialized secondary storage structures exist. Incorporating a new storage structure into an existing database implementation is a formidable task. Concurrency control, recovery and other modules would need to be changed in order to accommodate the new storage structure. It would be desirable to have an extensible object store that could accommodate new interfaces without causing major disruption to the existing system.

The Brown Object Storage System (BOSS) achieves extensibility by providing well defined interfaces to the designers of specialized storage structures. Through these interfaces the designer augments the structure with correct concurrent operation, fault tolerance and client-server distribution. BOSS aims to minimally constrain storage structure designers. To this end, BOSS is constructed from loosely coupled entities that cooperate asynchronously through well defined interfaces that communicate by message passing.

1 Architecture

Figure 1 illustrates BOSS as it fits into a distributed environment. It 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 BOSS. Clients communicate with BOSS via a client stub. The client stub takes advantage of

This work was partially supported by the Advanced Research Projects Agency order number 8220 under the Office of Naval Research contract number N00014-91-J-4085.

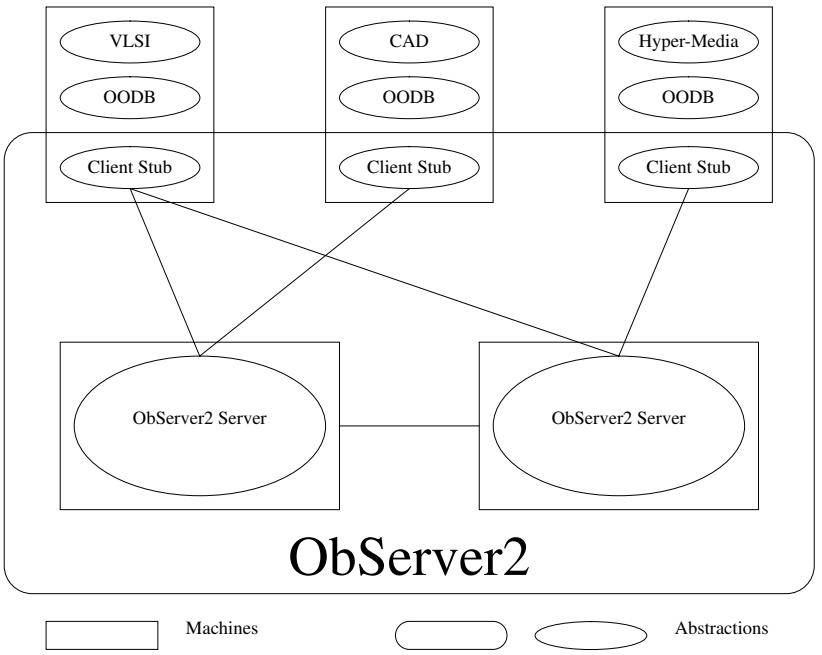

Figure 1:

local resources and communicates over the network to take advantage of BOSS servers.

BOSS'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. Application-level operations can result in computation at the client or communication with the server. Not all operations will result in communication with the server. For example, if an object is cached at the client, no communication is needed at the time of either a read or a write. These operations are buffered and can be sent to the server at commit.

Servers are safe, relatively passive repositories for persistent information. Clients pull what ever information they need to accomplish their tasks from the servers. BOSS's computation model does not hide distribution. The client explicitly contacts each of the required servers. A server never forwards a method or object request to another server. The servers communicate with each other only to synchronize for distributed commit.

1.1 Vertical Partitions

Figure 2 takes a closer look showing an abstract view of BOSS system internals. The rounded rectangles are Abstract Data Types (ADTs) that span the client and server. The rounded rectangle contains the operations which the ADT provides to the client OODB.

BOSS 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 internal interface. The external

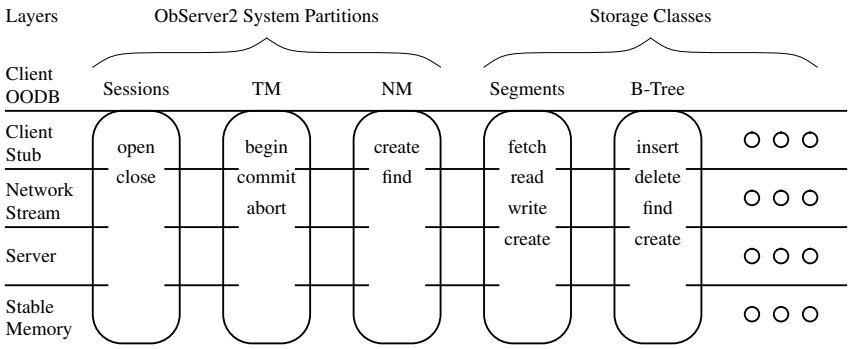

Figure 2: Layers & Vertical Partitions

interface it presents to the client OODB depends on the semantics of the storage class. Figure 2 shows BOSS configured with two storage classes: Segment and B-Tree. 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 also shows how BOSS 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 BOSS, only the storage classes change. The remainder of this paper describes exactly how the storage class fits into the vertical partitioning.

1.2 The Storage Class

The base implementation of BOSS 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 many applications. A new storage class could be implemented if an application needed to take advantage of a specialized secondary storage structure.

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. As an example we can consider the B-Tree mentioned in Section 1. One way to implement a B-Tree would be to send its pages up to the client on demand and perform the **insert**, **delete**, and **lookup** operations at the client. Alternatively, these operations could be sent to the server and performed there. The result would then be sent back to the client. Each option performs best under different workloads. BOSS allows both implementations to coexist.

Although the storage class has few constraints, all BOSS objects are addressed by a unique OID which encodes the class of an object and are stamped with a version number. 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. BOSS uses an

optimistic concurrency control algorithm which allows multiple versions of the same object to coexist.

Each storage class defines its own type specific concurrency control. The advantages of type specific concurrency control are discussed at length in [13]. In short, higher potential concurrency can be achieved by taking advantage of the semantics of an object's operations. 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.

2 Recovery & Persistence

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.

BOSS 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.

A new recovery algorithm, No-Redo Write Ahead Logging (NR-WAL), reduces the overhead of recovery. Intentions lists are used to eliminate the undo phase of recovery [2]. 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.

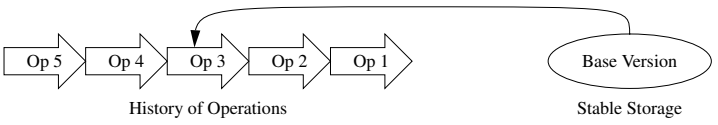

Figure 3: Components of the Current State of an Object.

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 modifications or observations. An observation does not change the state of the object.

Figure 3 shows the history of a series of modifications on an object. The modifications 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.

2.1 Implementation

A further explanation of NR-WAL requires a discussion of our concurrency control algorithm. ObServer2 uses a timestamp based, optimistic concurrency control algorithm [8]. A transaction stores its operations in operation records. An intension record is a collection of operation records and some additional information [2]. During the transaction the intension list is incrementally written to a sequential log in stable storage. Figure 4 shows the transaction table and two intention records in the log.

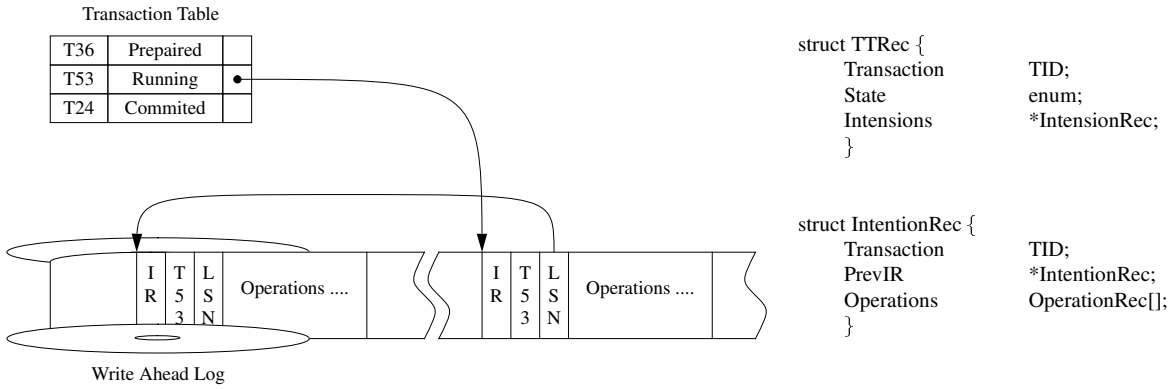

Figure 4: Chaining Intention Records

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 purpose of concurrency control. 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.

The operation record stores the OID and the LSN of the copy that was read. An modification operation changes the state of the object. So, in addition to the OID and the LSN of the object its operation record needs some indication of how to transform the old state in to the new state. It can take the form of an operation, a new value, or a delta. The choice of how to derive the new value from the old value and what to log is made on a class by class, operation by operation basis.

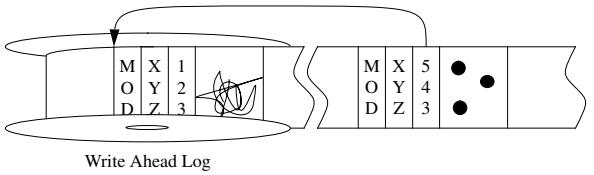

Figure 5: Operations in the Log.

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.

3 The Fixed Segment Storage Class

The storage class in the base configuration of BOSS is the Fixed Segment storage class (Fixed). It provides objects clustered into segments. Neither the objects or the segments can change size after creation, thus the name "Fixed." The functionality and implementation of these segments are similar to those provided by the original ObServer system [9].

A segment is a collection of variable length byte streams. The byte stream what is typically meant by an object in an object store although in BOSS the meaning of object is more general. To be explicit the segments are called fixed segments and the byte streams are called fixed objects. The fixed segment and each fixed object have a unique identity. Identifiers are structured so that the class of an object can be determined by the identifier. The implementation of the Fixed storage class encodes the identity of the segment in the identity of each object it contains.

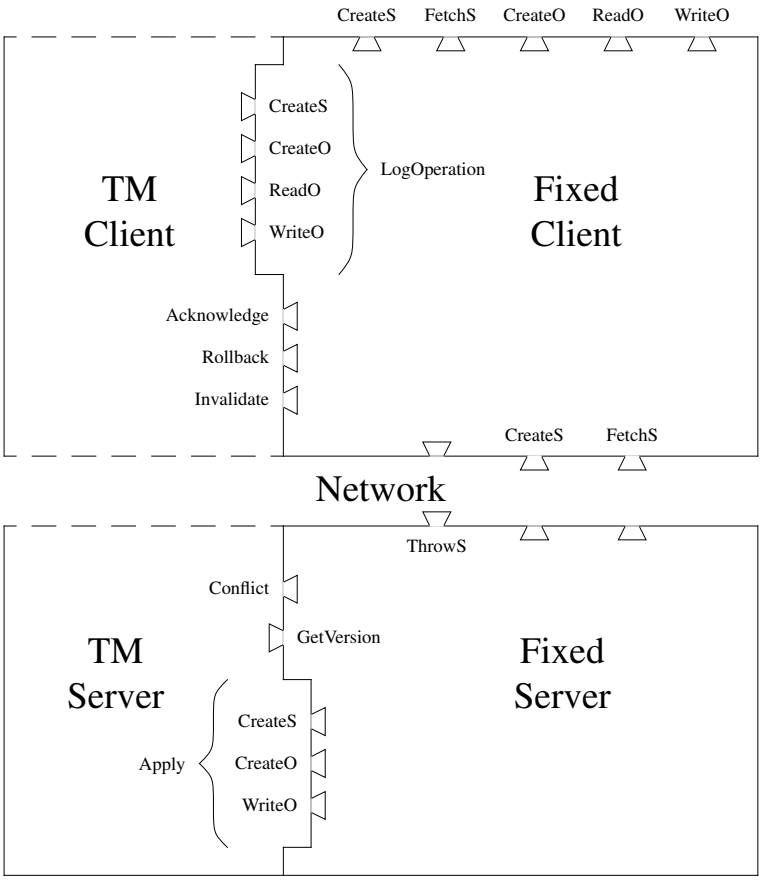

Figure 6: The Fixed Storage class module interface.

Figure 6 zooms in on the interaction between the storage class and the transaction manager and the distribution of

the storage class across the client and server. There are four main components. The API that Fixed provides to BOSS clients runs along the top of the diagram. This interface is completely determined by the storage class. BOSS places no restrictions on the client interface a storage class provides. The interface between the client portion and server portion of Fixed runs along the middle of the diagram. This interface is also unrestricted. Notice that it is a peer-to-peer interface not strictly client-server.

A standard interface between the storage class and the transaction manager on both the client and server. Storage class clients do not directly call transactional operations on storage class servers. Rather these operations are moderated by the transaction manager to achieve transactional consistency. The Fixed client passes operations into the transaction manager via the generic LogOperation entry point. These operations along with operations on other storage classes are combined into intension records by the client portion of the transaction manager. If these operations commit, the transaction manager applies these operations to the base versions. It does so via the generic Apply entry point.

3.1 Client Interface

We begin with a description of the "Fixed" client interface[1]. Fixed provides five operations to its clients.

`FixedStatus CreateS(ObjID& oid, unsigned size);` This creates a new segment and returns its OID.

Since CreateS is creating a new segment, it does not conflict with operations on any existing objects. The only possible conflict occurs when two segments are assigned the same OID. In this case one transaction must be aborted. To prevent this from occurring the client stub makes a synchronous rpc to the server portion of the storage class, so the it can assign a unique OID. The client then logs this operation and the resulting OID for persistence and transactional correctness.

`FixedStatus FetchS(ObjID& oid);` This moves a segment into the client machine.

FetchS performs no modifications and does not give the client client access to any objects. It merely indicates that the segment will be used in the near future and it should be resident. Because it has no impact on transactional correctness and makes no changes to the persistent state, this operation does not need to be logged. It simply makes an asynchronous call to the server requesting the segment.

`FixedStatus CreateO(ObjID& oid, unsigned size);` This creates a new object and returns its OID.

Like CreateS, CreateO does not conflict with operations on existing objects. The only possible conflict occurs if two objects are assigned the same OID. The client specifies the segment that the object should reside in and this is encoded in the OID which serves to differentiate objects created in different segments. Because of this, concurrent creation of conflicting OID s is unlikely. This operation generates the OID locally at the client

[1] The interfaces here are taken directly from the BOSS implementation.

without contacting the server. The operation and the resulting OID are logged for persistence and transactional correctness.

`FixedStatus ReadO(ObjID& oid, FixedObject *&obj);` This notifies the system that the object is to be read and returns a handle to the object.

The handle returned by this call is implemented as a virtual memory pointer which requires that the object be resident. If the object is not resident, its segment will be synchronously fetched from the server. Once this is complete, the segment is pinned in virtual memory (See section 5). The segment is pinned so the handle will be valid until the end of the current transaction. ReadO can cause a transaction to abort if it returned old data, so before the call returns the operation is logged for transactional correctness.

`FixedStatus WriteO(ObjID& oid);` This notifies the system that the object has been modified.

Clients have virtual memory pointers to objects which they have read. This allows the clients to make modifications directly on the cached copies. However, these modifications will not become permanent unless the WriteO operation is called. This operation takes the modifications from the object and logs them for persistence and transactional correctness.

3.2 TM Client Interface

The interface between the storage class client and the transaction manager client is peer-to-peer. The storage class client informs the transaction manager of the operations that it performs during a transaction. In return, the transaction manager informs the storage class client of global events which could effect the correctness of future operations.

3.2.1 Transactional Operations

The transactional operations performed at the storage class client in the client interface are not passed directly to the server portion of the storage class. Rather they are mediated by the transaction manager to ensure no out-of-date objects are observed and only committed operations take effect. A client gathers an operation's relevant information into a record and gives the record to the transaction manager at the client via the generic `LogOperation` entry point. `LogOperation` constructs an intensions list that will eventually be sent to the server.

The OperationRecord shown in Figure 7 contains a tag defined in Figure 9 to indicate which sort of operation is being performed. Next it stores the LSN of the copy that the operation observed. The OID of the object the operation effects is obviously required. The final entry that all OperationRecords contain is the size of the entire record. Individual storage classes add to the structure of operation records to carry what ever information they need to check the operation for correctness and propagate its effects.

```
class OperationRecord {
public:
    OperationType  type;
    Lsn object_lsn;
    ObjID  oid;
    unsigned long  record_size; // length of entire record

    // There will be more data here following these fields. The record_size
    // field contains the length of the header plus this extra data.
    };

class FixedOpRecord : public StorageOpRecord{
public:
    unsigned  osize;    // size of the object
    unsigned  offset;   // offset is the begining of the data field
    char *data()        {return (char *) &offset;}
    };
```

Figure 7: The Operation Record

All Fixed's operations record the size of the object. CreateO records the offset at which the object is allocated in the segment header. For a WriteO operation, this same space is the begining of the new value of the object.

3.2.2 Callbacks

The TM Client informs Fixed of the state of objects through three calls: `Invalidate`, `Acknowledge`, and `Rollback`.

```
class ClientStorageClass : public StorageClass {
public:
    virtual int  Invalidate(VerList&);
    virtual void Acknowledge(VerList&);
    virtual void Rollback(VerList&);
    };
```

Figure 8: The interface all storage classes clients present to the TM.

`Invalidate` is called by the server to notify the clients of objects which have been updated. It passes all updates performed at the server since a certain point in time and selects those objects from the notification that are cached locally and either discards them or requests the most recent version. `Acknowledge` is called after a transaction commits and contains the new LSNs of all the objects the transaction modified. It places those new LSNs in the objects at the client thereby completing the transaction, so the dirty copies of old versions become clean copies of the new versions. When a transaction aborts, all of the modified objects in the clients cache must be rolled back or discarded. `Invalidate` takes care of the out-of-date copies that caused the abort.

3.3 TM Server Interface

```
class OperationType {
  public:
    enum   StorageOpType {
        STORAGE_OP_NULL,
        FIX_OP_CREATE_S,
        FIX_OP_CREATE_O,
        FIX_OP_READ_O,
        FIX_OP_WRITE_O,
        STORAGE_NUM_OPS
    };

    int   isModifier()const;

    int   Apply(StorageClass *, const ObjID&, const Lsn&,  StorageOpRecord *);

  private:

    int   AppFixedCreateS(FixedServer *, const ObjID&, const Lsn&,
                          FixedOpRecord *);
    int   AppFixedCreateO(FixedServer *, const ObjID&, const Lsn&,
                          FixedOpRecord *);
    int   AppFixedWriteO(FixedServer *, const ObjID&, const Lsn&,
                         FixedOpRecord *);
};
```

Figure 9: Example OperationType class

All classes register their transactional operations in the class OperationType. Each operation is assigned a numonic operation code (`FIX_OP_WRITE_O` etc.). Transactional operations support `isModifier` and all operations that are modifiers support the `Apply`. Transaction manager determines if a committed operation is a modifier and if so calls `Apply` which dispatches to the appropriate application operation (`AppFixedWriteO` etc.) based on the operation code. The application operation is given raw data and casts it to the appropriate type so that a method may be called on the correct storage class.

Since the transaction manager does not even know which operations clients perform it cannot possibly know their concurrent semantics. This is handled genericly by `Conflict`. Each storage class implements its own conflict predicate which returns true if there is a conflict and false otherwise. To provide a consistent interface to the rest of the system the common functionality of all the server portions of storage classes is bundled together in the `ServerStorageClass`. Pseudo-code for the class is shown in Figure 10. The name manager and the server storage class cooperate to determine which conflict predicate to use. The transaction manager calls `ServerStorageClass::Conflict` with the LSN of an operation record. Based on the OID in the operation record the name manager determines which class the operation belongs to. `ServerStorageClass` then calls a method on the appropriate storage class.

`Conflict` determines if any previously committed operation conflicts with the operation currently under consid-

```
class ServerStorageClass : public StorageClass {
 public:
   static bool Conflict(const Lsn&);

 private:
   virtual bool LocalConflict(const Lsn&) = 0;
};
```

Figure 10: Pseudo-code for the `ServerStorageClass`.

eration. Each storage class determines exactly the semantics that it wants its operations to have and encodes those in the conflict operation. The concurrent semantics of the Fixed Segment storage class operations are described at their introduction. The pseudo-code in Figure 11 shows the implementation of the semantics.

```
bool
FixedServer::LocalConflict(const Lsn& lsn) {
    // Cast the LSN to an OperationRecord
    OperationRecord *or = (OperationRecord *) lsn;

    if (or->object_lsn.isNull()) {
        // The operation is a creator
        if (_tm->GetVersion(or->oid) == NULL)
        // No one else has already created it.
            return FALSE;
        else
            // The OID has already been used
            return TRUE;
        }
    else {
        // Get the previous operation from the version table
        Lsn *prev = _tm->GetVersion(or->oid);

        if (*prev == or->object_lsn) {
            // Read the most up-to-date version
            return FALSE;
            }
        else {
            // Read an out of date object, Abort
            return TRUE;
            }
        }
    }
```

Figure 11: Pseudo-code for the Fixed Segment Conflict operation.

Storage classes are free to implement the `Conflict` operation in any way, so long as it provides consistent transaction semantics for its operations. Sorting through the cryptic C++ code, we have a simple binary decision tree with 4 leaves. The first condition tests whether the operation is a creator or not. If the object is a creator, for example `CreateS` or `CreateO`, the only possible conflict is the previous existence of another object with the same identifier. This condition differentiates the first and second leaves. The first leaf indicates no conflict. It is reached if there is

no other object with the same identifier. The second leaf indicates conflict. It is reached if an object with the same identifier already exists. The second and third conditions are reached only if the operation is not a creator. The third leaf indicates no conflict. It is reached if the version of the object which was read is the current version. The forth leaf indicates conflict. It is reached if the version read is not the current version. This occurs when another transaction performs an update after the read but before the commit.

3.4 Internal Client Server Interface

The client stub is built on a network services layer that connects it with the server portion of the storage class. It implements its services using the API presented in Figure 12. The two calls from the client to the server are `CreateS`

```
class ClientRPCDisp : public MTCP_RPCDisp {
public:
    // Generate a unique id and reserve space at the server.
    FixedStatus RPCFixCreateS(ObjID&, unsigned);

    // Synchronously move a segment to the client
    FixedStatus RPCFixFetchS(const ObjID &, Mo *&);
};

class ServerRPCDisp : public MTCP_RPCDisp {
public:
   // Asynchronously move a segment to the client
   FixedStatus RPCFixThrowS(const Fixed & , const SesID & );
    };
```

Figure 12: The API for Fixed Segment network services.

and `FetchS`. `CreateS` takes an OID as an argument. The second argument is a size parameter. The segments in `FetchS` takes an OID and returns a pointer to a memory object (See 5) containing the desired segment or some error status indicating why it could not complete the operation. The one call from the server to the client, `ThrowS` passes the client a segment asynchronously.

Notable by their absence are `ReadO`, `WriteO`, and `CreateO`. None of these operations require any resources from the server during a transaction because they all operate on segments cached in memory or on local disk. If the segment is resident, the client stub logs the operations and the transaction manager takes care of the rest as described in Section 3.3. If the target segment is not cached, this call will generate a `FetchS` and wait for it to complete before they proceed.

`CreateS` does not actually have to contact the server. A client stub could be implemented that generated the id locally. The problem with this approach is the uniqueness of the id. Semantically there is no conflict with creating two segments at the same time. However, if the implementation assigns them both the same id there is a conflict at the physical level.

4 Designing a Storage Class

The intended client of BOSS is an OODB or some advanced application with special storage requirements. The basic functionality that BOSS provides is enough to implement any such application. However, new application domains such as Geographic Information Systems or the Human Genome Project occasionally mean new storage structures such as multi-dimensional search trees or structures for approximate string matching. Traditionally one of these applications has a choice between using an OODB or taking advantage for its specialized storage structures. Usually the performance of a standard OODB is too poor and the application is built from scratch.

BOSS allows the addition of new storage structures. This will not be a common activity, but the option is available to allow developers of new applications to integrate their specialized storage structures into BOSS. This section describes what is required to add a new storage class and presents an example of adding a non-traditional database service as a storage class.

4.1 Criterion for a Good Storage Class

The BOSS storage class designer requires special skills and an understanding of BOSS system internals. The first question the designer must answer is "Should the Abstract Data Type (ADT) under consideration be implemented as an abstraction in the application running above the object store in the object store itself as a storage class." Nothing beats experience, but the designer can use this checklist as a guide.

A. Are there significant performance advantages to be gained by implementing the ADT as a storage class?

   The primary reason for considering a storage class is to attain a significant performance improvement. The BOSS storage class facility was designed to allow applications to use an OODB without loosing the performance advantages of their specialized storage structures. BOSS gives control of memory management over to the designer so that the performance of a storage class can rival that of a fully custom implementation built directly on the operating system. An added advantage to implementing an ADT as a storage class rather than on the OS is consistent transaction semantics across not only the one new ADT but all other ADT s that have been incorporated into the object store.

   This criterion is difficult to test without knowing the specific qualities that a storage class can exploit. The next several points expound on this theme.

B. Can the ADT exploit the storage media?

   A storage class has direct access to both the server's disk for persistent storage and the client's disk for local disk caching. Significant performance improvements are attainable by exploiting the disk geometry. Sequential storage of B-Tree pages is one example and sequential storage of large segments containing objects is another.

Another facility that BOSS provides in this area is the mapping of object identifiers to objects. OID s are 32 bits long, but BOSS uses only the first 10 of these bits. The remaining 22 are left to the storage class to implement whatever sort of OID to storage mapping best suits its needs. A storage class that provides pages could use a direct mapping while a storage class that provides mobile objects would require more sophisticated OID to object mapping.

C. Is the access behavior predictable or exploitable?

   If data access can be predicted it can be exploited A storage class can have more knowledge about how its own data is accessed. It knows both the semantics of the operations it provides and the history of the objects it manages. Using this information a storage class can make predictions about future access behavior. It can exploit this knowledge by moving an object to the appropriate place in the memory hierarchy. An object that will be accessed in the near future can be moved closer to the client and an object which will not be used for some time can be moved further from the client to free limited resources.

D. Does the ADT offer new functionality which cannot be constructed using existing storage classes.

   The base configuration of BOSS includes the Fixed Segments storage class which provides the generic objects that are no more than unique OID s associated with byte streams. There is little that cannot be constructed with this class of objects, but occasionally there is a need for very different functionality. Active objects are require the setting of triggers and would therefore, require more than any passive storage class could deliver.

E. Does the ADT offer enough general utility to warrant the effort of constructing a new storage class.

   This last criterion is just good software engineering practice. A storage class should serve as wide an audience as possible and if it will not be generally used, don't build it. To improve the utility of a storage class, reduce the functionality to the absolute minimum. Often there is a need for a specific functionality in a particular application domain such as gene sequence matching in the human genome project. The BOSS approach is to only include the core functionality in a storage class and build the rest of the functionality as an application.

To demonstrate the use of this checklist we consider three different abstractions, the B-Tree, the Employee, and the Condition Variable (CV) and their implementations as storage classes. Almost every database implements a B-Tree for associative access, so a priori it is a likely candidate. Another common element in business databases is the Employee. It's implementation as a storage class is more questionable because it is usually included as an application level object not built into the database. However, an objective of BOSS is to help capture and exploit the semantics of objects, so we will consider the Employee as a storage class to see how well if fairs. The Condition Variable is on the other end of the commonality spectrum from the B-Tree. The CV is common in multi-threaded operating systems, but does not exist in any database implementation. The CV turns out to be a very desirable tool in a modern database implementation. Triggers and Cooperative Transactions, neither of which are implemented within BOSS, could be implemented as a layer above the database using the CV as a fundamental building block. The checklist is applied to each ADT and its implications discussed.

	B-Tree	Employee	CV
A.	√	X	√
B.	√	X	X
C.	√	X	X
D.	√	X	√
E.	√	√	√

Not surprisingly the B-Tree turns out to be a highly desirable candidate for implementation as a storage class. Its pages can be laid out sequentially, its nodes are always accessed from the root downwards, and its operation provide opportunities for more concurrent interleavings.

The Employee does not fair so well it can be handled well in an existing record oriented database. The one operation that could be optimized is iterating through all employees, but this could be handled well by a B-Tree.

The CV is an interesting case. It does not have specialized storage structures or predictable access behavior. It is, however, an active object and cannot be constructed using any passive storage class. And, there is a broad class of applications that can take advantage of a CV. For these last two reasons the CV would make an excellent storage class and it will be used as an example later in the paper.

4.2 Refining the Interface

Once the designer has an ADT that is a good candidate for implementation as a BOSS storage class, the interface of the ADT must be refined and integrated into the BOSS framework. The process begins by determining the ideal semantics of the ADT and gradually refines the interface to maintain the necessary invariants for correct concurrent, fault-tolerant operation. To illustrate the refinement process we use the Condition Variable. A standard CV provided by an operating system has the following operators:

Create Create a new condition variable

Wait Suspend the calling thread.

Signal Signals the condition and wakes one blocked thread.

Broadcast Signals the condition and wakes all blocked threads.

There are several semantic refinements that the CV needs as stated. The transaction contains a tread of control, but there is no formal concept of thread in BOSS. Signal and Broadcast are somewhat redundant. The semantics of Signal can actually be implemented using Broadcast and another simple read/write object in a transactional context.

In the above description a well defined notion of time is taken for granted. This does not exist in BOSS and must be constructed using the facilitates that BOSS provides. Unlike threads, transactions have a well defined ordering.

The semantics of CV must prevent a transaction from waking up before it is signaled. For example, transaction S signals the variable then does some work before it commits. Transaction W wakes up and commits immediately. W is likely to be serialized before S, a clear violation of the semantics of CV.

Another problem that a transaction can be woken up by a transaction that aborts , creating the appearance that the transaction was woken up by nothing at all. For example, transaction S signals the variable. Transaction W wakes up. Transaction S aborts. A transaction which aborts must appear to have never run, so W must have been woken up by nothing at all.

The concurrent semantics of CV are that a transaction can only be woken up by a previously committed transaction. This constraint can be implemented using timestamps. Wait keeps track of the timestamp of the condition when it was called. Notify increments the timestamp when it commits. A Wait cannot commit successfully if the timestamp is the same as when the condition was first observed. The first column in the following table shows the time stamp of the condition variable. All operations are tagged with the timestamp of the version that was observed.

Condition Variable Semantics

Condition	Trans' S		Trans' W	
@57	begin	ok	begin	ok
	wait@57	(delay)		
			notify@57	ok
	(resume)	ok		
	commit	fail		
@73			commit	ok
	begin	ok	begin	ok
	wait@73	(delay)		
			notify@73	ok
	(resume)	ok		
			commit	fail
	commit	fail		
	begin	ok	begin	ok
	wait@73	(delay)		
			notify@73	ok
	(resume)	ok		
@98			commit	ok
	commit	ok		

This example consists of three pairs of transactions. The first two have a semantic error that causes an abort. The last one does not so both transactions commit successfully. The first pair have the problem that Transaction W tries to commit before Transaction S. This is forbidden because the timestamp on the base object is equal to that observed. The second pair have the problem that Transaction W apparently wakes up for no reason, actually do to Transaction S's notify and abort. Again Transaction W cannot commit because the timestamp on the base object is equal to that observed. In the final pair Transaction S notifies and commits before Transaction W thereby incrementing the timestamp and allowing Transaction W to commit.

These examples do not explicitly show the log, but its contents can be inferred. At location 73 in the log there is an operation record containing "notify@57". At location 98 there is another operation record containing "notify@73". The actual records contain more information, but it is clear that modifiers for the CV storage class also form a linked list.

Determining the concurrent semantics of the candidate storage class is the first step of integrating a new storage class into BOSS. The next step is determining how the semantics can be implemented using the facilities that BOSS provides. Now the storage class designer is ready to begin integrating the storage class into BOSS.

4.3 Storage Class Integration

This section presents the integration of the CV storage class (Cond) into the existing BOSS system. The base BOSS implementation contains one storage class that provides variable sized segments that contain variable sized objects. Since the size of the segments and objects cannot change after creation, it is called the Fixed Segment storage class (Fixed). Integration of Cond is presented along side of Fixed.

4.3.1 CV Client Interface

Cond is different from Fixed in many ways, but integrating Cond requires a similar description of the operations. One way is that the clients never requires access to the representation of a Cond object. Another difference is in the size of a cond object. Buffering these objects is not much of an issue, so there is no explicit fetch operation for the Cond storage class.

`CondStatus Create(ObjID& oid);` Create a new CV and return its OID.

Like the create operators for Fixed, Create conflicts with no existing object. The only possible conflict is returning an OID which has already been allocated. To avoid this possibility Create makes a synchronous call to the server.

`CondStatus Notify(ObjID& oid);` Wake up all transactions waiting on this CV.

As the transaction examples illustrate, waking up another transaction before the notifier commits is likely to cause aborts, so the implementation of Notify actually does nothing except log the operation. After the transaction commits, the cache coherency policy informs other clients that the object is out-of-date.

`CondStatus Wait(ObjID& oid);` Block until the condition variable is notified.

Wait simply logs the operation and blocks until the cache coherency policy notifies the client that the object is out-of-date. At that time the Wait call returns and the transaction can continue processing.

4.3.2 CV TM Server Interface

The three operations that Cond registers with `OperationType` are `COND_OP_CREATE`, `COND_OP_WAIT` and `COND_OP_NOTIFY`. Figure 9 illustrates the requirements for Fixed and the additions necessary for Cond are trivial. A more significant difference exists in the Conflict operator because it has very different semantics.

```
bool
CondServer::LocalConflict(const Lsn& lsn) {
    // Cast the LSN to an Operation Record
    OperationRecord *or = (OperationRecord *) lsn;

    // Get the previous operation from the version table
    Lsn *prev = _tm->GetVersion(or->oid);

    if (or->type == COND_OP_CREATE) {
        if (prev == NULL)
            // No one else has already created it.
            return FALSE;
        else
            // The OID has already been used
            return TRUE;
        }
    else if (or->type == COND_OP_NOTIFY) {
        if (*prev == or->object_lsn)
            // Read the most up-to-date version
            return FALSE;
        else
            // Read an out of date object, Abort
            return TRUE;
        }
    else if (or->type == COND_OP_WAIT) {
        if (*prev == or->object_lsn)
            // No one has called Notify, Abort
            return TRUE;
        else
            // Notify has been called
            return FALSE;
        }
    else
        // There is an error.
        return TRUE;
    }
```

Figure 13: C++ code for the CV Conflict operation.

Again, we have a cryptic C++ decision tree in Figure 13. The first two cases are very similar to those in Fixed. The creator succeeds so long as the OID has not already been allocated. Notify succeeds so long as it observed the most recent state of the object. This raises the question of why Notify should ever fail.

The examples in section 4.2 do not show an invocation of Notify ever failing, because the only semantic constraint on the CV is that a transaction cannot be woken up before another transaction notifies it. There is no semantic constraint

on when notify can be called. There is however a physical conflict. A physical conflict in the absence of a semantic conflict is called a false conflict.

Simple read/write objects like pages or the variable length objects in Fixed naturally link modifiers together in a linear order because the write operation both observes and modifies the state of the object. A modifier which does not observe the state of the underlying object is said to be a blind operator [13]. Notify is such an operator. Blind operators never cause semantic conflicts, but can cause physical conflicts.

BOSS recovery has a physical requirement that modifiers be linked together in a linear order. In Weihl's terminology, modifiers cannot commute. For Cond this means that Notify can cause an abort. This constraint is not as bad as it might at first seem. First, any physical conflict that occurs in BOSS would occur in a system which just offered physical concurrency control; thus, BOSS is strictly better than a physical system with respect to false conflicts. Secondly, observers can commute over modifiers and modifications tend to be less frequent than observations which implies modifier/modifier conflicts are far less frequent than modifier/observer conflicts.

The final operator, Wait, does cause semantic conflict. The semantics of Wait are that it cannot commit before the transaction that notified it which means that there must be a modification between the time the transaction waits and the time the transaction commits. Thus Wait causes a conflict if the object is in the same state as when it was first observed, in direct contrast to almost every other possible operator.

4.3.3 CV TM Client and Internal Interface

The Cond server is trivial. All it needs to do is keep track of the most recent version of every CV and give the clients access to this information. The operations that it provides to the client are correspondingly simple.

```
class ClientRPCDisp : public MTCP_RPCDisp {
public:
    // Stuff for Fixed deleted . . .

    // Generate a unique id
    CondStatus RPCCondCreate(ObjID&, unsigned);

    // Synchronously give the client the most recent version number
    CondStatus RPCCondFetch(const ObjID &, Mo *&);
};
```

Figure 14: RPC Stub for the Cond internal interface.

Create generates a unique identifier as explained in section 3.1. Fetch is needed because the client needs to know what the most recent version of the CV when wait is called. Note that no state is returned. Since, no state is required for Cond there is no `CondOpRecord`. Cond uses the default `OperationRecord` because it contains all the information about what operation was invoked on which object when and that is all that Cond requires.

5 Memory Object Management

The storage class allows new ADT s to be added safely. Memory Object Management (MOM) makes the addition of a storage class easier. MOM hides much of the complexity of dealing with the operating system and mediates between conflicting demands of storage classes. We considered many architectures for this service and experimented with several. The one we have settled on places one MOM on each BOSS host, client or server. This provides the storage class implementer with a consistent interface, reuses the MOM source code, and gives BOSS intelligent client disk caching for almost nothing.

The unit of memory that MOM manages is the Memory Object (MO). A MO is a variable length, continuous piece of memory. All MOs have a piece of non-volatile memory allocated to them. When the storage class needs to use the MO, it pins the MO into virtual memory. This does not guarantee that the page will be loaded into RAM, but it does allocate a virtual memory handle that the storage class can use to access the contents of the MO.

5.1 Storage Class Interface

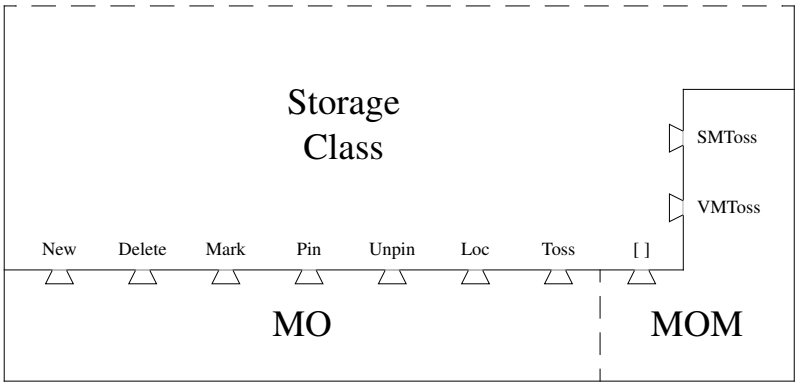

Figure 15: The Storage Class interface to the Mom module.

The interface that MOM presents is the same at both the client and the server. Actually, MOM presents only one operation to the storage class which will be discussed later. Most interactions are with Memory Objects themselves. The MO interface is fairly straight forward. There are seven calls:

`new (int size) (ObjID oid)` New creates a new memory object of the requested size and pins it into virtual memory. New returns a handle to that memory. All MOs are associated with an OID which is passed to the constructor.

`delete *Mo` Delete frees the non-volatile memory and any virtual memory associated with the MO.

`int mark()` Marks the object read for the purposes of the default replacement algorithm. Mark returns 1 on success.

`void *pin()` Pin allocates a portion of virtual memory to the MO and returns a pointer to that memory if successful.

`int unpin()` Unpin does not deallocate memory, it only gives MOM the option to do so. MOM attempts to keep MOs that will be used in the near future in virtual memory. Unpin returns 1 on success.

`void *loc()` If a MO has virtual memory allocated to it, Loc will return its address without pinning the object or allocating virtual memory. Storage classes use this call to determine whether or not an MO is already cached.

`int toss()` Toss indicates that the storage class will not be using the MO for some time and that it should be discarded.

Just as important as the calls in the MO interface are the calls that are not. There are no calls to mark a MO dirty or to write its contents out to disk. MOM handles these tasks, but notifies the storage class when it takes an action via the two callbacks:

`int VMToss(Mo *mo)` MOM notifies the storage class that it intends to discard an object via this call. The storage class does have the option of refusing the request. The storage class should only do so if the MO will be used in the immediate future. Consistently refusing to discard MOs will quickly cause the entire system to deadlock.

`int SMToss(Mo *mo)` SMToss serves the save function that VMToss but in the domain of non-volatile memory. This call is only made at the client where non-volatile memory is used as a cache. The server non-volatile memory is used only for persistent MOsso if it fills up new simply fails.

The last operator we have to discuss is the association operator, [ ]. There was a great deal of debate about whether to include this operator. Its function is simple, given an OID return a MO, but it has major implications on the sort of storage classes that can be constructed. It does not restrict the interface of the storage class, but it does have some performance implications.

A performance critical operation in an OODB is the mapping of logical OID s to physical data locations and there are many ways this mapping can occur [11, 5, 3]. Previous MOM designs did not include this mapping. The rationale was that storage classes should implement what ever logical to physical mapping that best suits its semantics. For instance this would allow a storage class that implemented pages to compute the physical location from the logical identifier and save a table lookup. However, this is not of much use when storage class cannot choose MO's physical location. A storage class which required this sort of mapping could be built with out using MOM, but this would require a greater implementation effort.

The design decision for this iteration was that since MOM determined the physical location of MOs it should also perform the physical to logical mapping. This decision did not significantly complicate the implementation of MOM and greatly simplified the implementation of the existing storage classes. A future design may allow specialization of the association operator which would allow storage classes to choose its implementation. Static binding in C++ makes this sort of specialization difficult to accomplish.

5.2 Example B-Tree Memory Management Policy

Storage classes are expected to be good neighbors and not aggressively acquire as many resources as possible. This will cause poor overall system performance. This assumption of benevolence is justified by the fact that all storage classes run in the same address space and if there were malevolence, there are much worse things the malevolent class could accomplish.

B-Tree page replacement is an example of how a storage class can use MOM to achieve good overall resource utilization. MOM implements a global LRU policy. LRU performs well for general purpose computing but fails for special applications. To correct this in BOSS the B-Tree augments the global policy to exploit the semantics of its application. B-Trees are always accessed from the root downwards, so there is no point to keeping a child of a node that is being discarded. This augmentation occurs within the VMToss and SMToss calls. Whenever a node is discarded, the storage class recursively descends the tree using loc and tosses all resident child nodes.

6 Conclusion

The BOSS storage class mechanism allows new ADT s to be added into an object store with few restrictions. The storage class can exploit the semantics of an ADTto offer higher levels of concurrency and better throughput. The implementation of a storage class is supported by the Memory Object Manager which provides a high level interface to operating system resources. The policies that MOM provides can also be customized to take advantage of storage class semantics. The BOSS system operates in a network of Sparc 10's running SunOS and the performance of the unoptimized system is within an order of magnitude of commercial systems[10].

References

[1] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*, chapter 1. Morgan Kaufmann, 1992.

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

[3] A. Brown and J. Rosenberg. Persistent object stores: An implementation technique. In *The Fourth International Workshop on Persistent Object Systems*, 1990.

[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] B. Koch *et al.* Cache coherency and storage management in a persistent object system. In *The Fourth International Workshop on Persistent Object Systems*, 1990.

[7] M. Atkinson *et al.* The persistent object management system. Technical Report PRRR-1, The Universities of Glasgow and St. Andrews, 1983.

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

[9] M. F. Hornick 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.

[10] D. Langworthy and S. Zdonik. Extensibility and asynchrony in the brown object storage system. In V. Kumar, editor, *Performance of Concurrenty Controll Mechanisms in Centralized Database Management Systems*. Prentice Hall, 1994.

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

[12] E. Shekita and M. Zwilling. Cricket: A mapped, persistent object store. In *The Fourth International Workshop on Persistent Object Systems*, 1990.

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