

# Storage Class Extensibility in the Brown Object Storage System

David E. Langworthy  
Stanley B. Zdonik

Department of Computer Science  
Brown University  
(del,sbz)@cs.brown.edu

## Abstract

The Brown Object Storage System (BOSS) provides extensible support for persistence in a distributed multi-client, multi-server environment. BOSS is built on a loosely coupled, asynchronous computation model. One of the artifacts of asynchrony is increased extensibility. This paper presents the storage class, the means by which the BOSS client interface is extended.

The BOSS prototype is completed and operated in a network of Sparc10s running SunOS 4.1. Currently the untuned prototype operates at 6 short transactions (20 reads, 4 writes) per second.

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. OODBs support new application domains with extensible type systems that allow new abstractions to be added easily. All OODBs use some distinct service that provides stable storage for objects or pages, referred to as object servers and page servers respectively [11, 5, 19, 13, 1]. Whenever an OODB attempts to model a domain for which specialized secondary storage structures exist, the object store causes a problem. Concurrency control, recovery and other modules need to be changed in order to accommodate the new storage structure. An extensible object store would accommodate new interfaces without causing major disruptions 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.

---

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

Figure 1: An overview of the BOSS system architecture.

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.

Figure 2: Layers & Vertical Partitions

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 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 [21]. 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 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 require new storage structures, such as multi-dimensional search trees or structures for approximate string matching. Traditionally, such applications have to make a choice between using an OODB or taking advantage of their own 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. Adding new storage structures 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 the requirements for adding a new storage class and presents an example of adding a non-traditional database service as a storage class.

### 2.1 Criteria 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 or in the object store itself, as a storage class?" Nothing beats experience, but the designer can use this checklist as a guide.

- A. Will significant performance advantages be gained by implementing the ADT as a storage class?

The primary reason for considering a storage class is to significantly improve performance. The BOSS storage class facility was designed to allow applications to use an OODB without losing the performance advantages

of their specialized storage structures. BOSS gives the designer control of memory management so the performance of a storage class can rival that of a fully custom implementation built directly on the operating system. An added advantage of 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 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. OIDs 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 might use direct mapping, whereas 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 that cannot be constructed using existing storage classes?

The base configuration of BOSS includes the fixed segments storage class, which provides generic objects that are no more than unique OIDs associated with byte streams. There is little that cannot be constructed with this class of objects, but occasionally there is a need for different functionality. Active objects require the setting of triggers and therefore require more functionality than any passive storage class can 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 a specific functionality is needed in a particular application domain, such as gene-sequence matching in the Human Genome Project. The BOSS approach is to include only 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 object. The implementation of the employee as a storage class is more questionable because this abstraction is usually included as an application-level object, not built in to 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 it fares. The condition variable is on the other end of the commonality spectrum from the B-Tree. A CV is common in multi-threaded operating systems but does not exist in any database implementation. However, a CV is a very desirable tool in modern database implementations. 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. Table 1 shows the results of applying the to each ADT.

Table 1: The Criteria Evaluated for Three Potential Storage Classes

	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 downward, and its operations provide opportunities for more concurrent interleavings.

The Employee as a storage class does fare as well. This abstraction can be handled effectively in an existing record oriented database. The one operation that could be optimized is iterating through all employees, but a B-Tree could be used for this operation.

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. Also, 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 in the next section.

### 3 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 component is a database partition that contain the base versions of the data. Durability of modifications made by transactions is achieved by entering an intentions record in the log. 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 [3]. There is no redo phase because NR-WAL does not require and up-to-date copy of an object to exist, so it does not need to be recreated immediately after a crash.

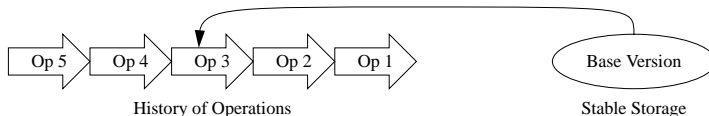


Figure 3: Components of the Current State of an Object.

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.

#### 3.1 Implementation

A further explanation of NR-WAL requires a discussion of our concurrency control algorithm. BOSS uses a timestamp based, optimistic concurrency control algorithm [10]. A transaction stores its operations in operation records. These records are collected in an intention list [3]. During the transaction the intention list is incrementally sent to the server then 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

```

struct TTRec {
    Transaction
    State
    Intentions
}
    TID;
    enum;
    *IntentionRec;

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

```

Figure 4: Chaining Intention Records

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.

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.

## 4 The Fixed-Segment Storage Class

The storage class in the base configuration of BOSS is the fixed segment storage class (also known as Fixed). It provides objects clustered into segments. Neither the objects nor 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 [11].

Figure 5: The Fixed Storage class module interface.

A segment is a collection of variable length byte streams. A byte stream is 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 5 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. The interface has four main components: Fixed to client, Fixed to TM server, Fixed client to Fixed server.

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 that a storage class provides.

The interface between the client and server portions 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 separates 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, the transaction manager moderates these operations to achieve transactional consistency. The Fixed client passes operations to the transaction manager via the generic `LogOperation` entry point. These operations, along with operations on other storage classes are combined into intention lists by the client portion of the transaction manager. If these operations commit, the transaction manager applies them to the base versions via the generic `Apply` entry point.

## 4.1 Client Interface

The Fixed client interface<sup>1</sup> provides five operations to its clients.

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

Because `CreateS` creates 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 problem, the client stub makes a synchronous rpc to the server portion of the storage class, so 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 call moves a segment into the client machine.

`FetchS` makes no modifications and does not give the client access to any objects. It merely indicates that the segment will be used in the near future and that 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 call 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 where the object should reside and this location is encoded in the OID to differentiate objects created in different segments. This encoding makes concurrent creation of conflicting OIDs unlikely. The operation generates the OID locally at the client without contacting the server. The operation and the resulting OID are logged for persistence and transactional correctness.

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

---

<sup>1</sup>The interfaces here are taken directly from the BOSS implementation.

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 operation is complete, the segment is pinned in virtual memory (see Chapter 6). 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 returns old data, so before the call returns, the operation is logged for transactional correctness.

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

Clients have virtual memory pointers to objects they have read. These pointers allow 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.

## 4.2 TM Client Interface

The storage class client and the transaction manager client have a peer-to-peer relationship. The storage class client informs the transaction manager of the operations it performs during a transaction. In return, the transaction manager informs the storage class client of global events that could affect the correctness of future operations.

### 4.2.1 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 that no out-of-date objects are observed and that 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 intentions list that eventually is sent to the server.

**OperationRecord**, in Figure 6, contains a tag defined in Figure 8 that indicates what sort of operation is being performed. Next, it stores the LSN of the copy that the operation observed. Then the OID of the object the operation affects appears. The final entry that all operation records contain is the size of the entire record. Individual storage classes add entries to the operation record's structure to contain whatever information they need to check the operation for correctness and propagate its effects.

The **FixedOpRecord** is passed as data after an **OperationRecord**. 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 location is the beginning of the new value of the object.

```

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 beginning of the data field
    char *data()      {return (char *) &offset;}
};

```

Figure 6: The definition of the operation record

#### 4.2.2 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 7: The interface all storage classes clients present to the TM

The server calls **Invalidate** to notify clients of objects that have been updated. **Invalidate** passes all updates performed at the server since a certain point in time and then selects from the notification those objects that are cached locally and either discards them or requests the most recent version. **Acknowledge** is called after a transaction commits. It 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, **Rollback** discards all of the modified objects in the clients cache. **Invalidate** takes care of the out-of-date copies that caused the abort.

### 4.3 TM Server Interface

All classes register their transactional operations in the class **OperationType** (see Figure 8). Each operation is assigned a numonic operation code (**FIX\_OP\_WRITE\_0** and so on). Transactional operations support **isModifier** and all operations that are modifiers support **Apply**. The TM determines whether a committed operation is a modifier and, if so calls **Apply** which dispatches the appropriate storage class operation (**AppFixedWrite0** etc.) based on the operation code.

The storage class operation is given raw data which it casts to the appropriate type so that a method may be called.

```
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 8: Definition of `OperationType`

The TM does not have any knowledge of the operations clients perform. Therefore, it cannot determine or moderate their concurrent semantics. This is handled generically by `Conflict`. Each storage class implements its own conflict predicate, which returns true if a conflict occurs 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` shown in Figure 9. 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. Using the OID in the operation record, the name manager determines the class to which the operation belongs. `ServerStorageClass` then calls a method on the appropriate storage class.

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

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

Figure 9: Definition of `ServerStorageClass`

`Conflict` determines if any previously committed operation conflicts with the operation currently under consideration. 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 10 shows the implementation of the semantics.

```

bool
FixedServer::LocalConflict(const Lsn& lsn) {
    // Cast the LSM 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 10: Pseudo-code for the fixed segment conflict operation

Storage classes must implement the **Conflict** operation to provide consistent transaction semantics for its operations. Sorting through the C++ code in Figure 10 reveals a simple binary decision tree with four leaves. The first condition tests whether the operation is a creator or not. If the operation is a creator, **CreateS** or **CreateO**, the only possible conflict is the previous existence of another object with the same identifier. If the operation is not a creator it is either **ReadO** or **WriteO**. In either case the operation succeeds if the most recent version of the object was observed and fails otherwise. Failure occurs when another transaction performs a write after the current transaction's read but before the commit.

#### 4.4 Internal Client Server Interface

The client stub is built on a network services layer that connects it to the server portion of the storage class. It implements its services using the API presented in Figure 11. The two calls from the client to the server are **CreateS** and **FetchS**. **CreateS** takes an OID and a size parameter. **FetchS** takes an OID as an argument and returns a pointer to a memory object (see Chapter 6) containing the desired segment unless an error occurs then it returns 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.

```
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 11: The API for fixed segment network services

Notable by their absence are `ReadO`, `WriteO`, and `CreateO`. None of these operations requires 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 4.3. If the target segment is not cached, this call generates a `FetchS` and waits for it to complete before proceeding.

`CreateS` does not actually have to contact the server. A client stub could be implemented that generated the OID locally. The problem with this approach is the uniqueness of the OID—semantically, creating two segments at the same time does not cause a conflict. However, if the implementation assigns them both the same OID a conflict occurs at the physical level.

## 5 Adding a Condition Variable Storage Class

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 refining 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** This operator creates a new condition variable

**Wait** This operator suspends the calling thread.

**Signal** This operator signals the condition and wakes one blocked thread.

**Broadcast** This operator signals the condition and wakes all blocked threads.

The CV needs several semantic refinements. The transaction contains a thread of control, but there is no formal concept of thread in BOSS. Signal

Table 2: Condition Variable Semantics

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

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

CV semantics must prevent a transaction from waking up before it is signaled. For example, transaction S signals the variable and 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 CV semantics.

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

The concurrent semantics of the CV are that a transaction only be awakened only 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. Wait cannot commit successfully if the timestamp is the same as when the condition was first observed. The first column in table shows the time stamp of the condition variable. All operations are tagged with the timestamp of the version that was observed.

This example consists of three pairs of transactions. The first two have a semantic error that causes an abort. The last one does not have an error, so both transactions commit successfully. The first pair have the problem that transaction W tries to commit before transaction S. This sequence is forbidden because the timestamp on the base object is equal to that of the version

observed. The second pair have the problem that transaction W apparently wakes up for no reason, though it is actually awakened by transaction S's notify and abort. Again transaction W cannot commit because the timestamp on the base object is equal to that of the version 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. Location 73 in the log stores an operation record containing "notify@57." Location 98 stores an 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 in 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.

## 5.1 Storage Class Integration

This section presents the integration of the CV storage class (Cond) into the existing BOSS system.

### 5.1.1 CV Client Interface

Cond is different from Fixed in many ways, but integrating Cond requires a similar description of the operations. One difference is that 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** does not conflict with any existing object. The only possible conflict is the return of an OID that was already 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.

### 5.1.2 CV TM Server Interface

The three operations that Cond registers with `OperationType` are `COND_OP_CREATE`, `COND_OP_WAIT` and `COND_OP_NOTIFY`. Figure 8 illustrates the requirements for Fixed; 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 LSM 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 12: C++ code for the CV Conflict operation

Again, in Figure 12 we have a decision tree. The first two cases are analogous to those in Fixed. The creator succeeds so long as the OID has not already been allocated and `Notify` succeeds so long as it observed the most recent state of the object, which raises the question of why `Notify` should ever fail.

The examples in Section 5 do not show an invocation of `Notify` ever failing, because the only semantic constraint on the CV is that a transaction cannot be awakened 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 such as pages or the variable length objects in Fixed naturally link modifiers in a linear order because the write operation

both observes and modifies the state of the object. A modifier that does not observe the state of the underlying object is said to be a blind operator [21]. **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 linearly linked. 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 that offered only physical concurrency control; thus, BOSS is strictly better than a physical system with respect to false conflicts. Second, 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 a modification must occur 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.

### 5.1.3 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 clients access to this information. The operations that it provides to the client are correspondingly simple.

**Create** generates a unique identifier, as explained in Section 4.1. **Fetch** is needed because the client needs to know 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 operation record because this record contains all the information on what operation was invoked on which object and when the invocation occurred, and that is all that Cond requires.

```
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 &, Lsn &);
};
```

Figure 13: RPC Stub for the Cond internal interface.

## 6 Memory Object Management

The storage class allows new ADTs to be added safely. Memory Object Management (MOM) makes the addition of a storage class easier. MOM hides much

Figure 14: The Storage Class interface to the Mom module.

The interface that MOM presents is the same at both the client and the server. Most interactions are with Memory Objects themselves. Actually, MOM presents only one operation to the storage class which will be discussed later. 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, pins it into virtual memory, and returns a handle to that memory. The MO is associated with the OID 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.

**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 OIDs to physical data locations and there are many ways this mapping can occur [16, 6, 4]. 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, MOM does not allow a class to choose a MO's physical location, so this is not of much use. 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 MOM should also perform the physical to logical mapping since MOM determined the physical location of MOs. 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.

## 6.2 Example B-Tree Memory Management Policy

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.

## 7 Related Work

The topic of object servers has been well explored in the literature. Here we draw comparisons with this work. Further, as an extensible system, BOSS must be compared to efforts to create extensible databases in the relational community. Finally, BOSS draws from novel distributed file system technology so the similarities and differences are noted.

### 7.1 Object & Page Servers

ObServer [11], the precursor to BOSS, introduced several features: semantic clustering of related objects, a novel lock set for cooperative work, and a notification system which also supports cooperative work. BOSS 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. Much of the functionality that ObServer offered is encapsulated in the segment storage class within ObServer 2. The ability to add new storage structures allows BOSS to be specialized for high performance on specific applications.

The Exodus Storage Manager (ESM) [5], a page server developed at Wisconsin, provides support for values and indices. It generalizes the Aries Transaction Recovery System [15] 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 BOSS implementation allows faster access to a record in the log by using memory mapping.

The Mneme Persistent Object Store [16] 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.

Camelot [8] 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 BOSS model allows clients to operate on remote machines and lets a specialized server can handle all disk

operations, so 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.

The Avalon [8] 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. BOSS captures semantics within the persistence layer to allow for special memory management policies or type specific concurrency control.

Arjuna [7] offers functionality similar to Avalon, but the internal architecture is completely different. There is no separate persistence layer. Persistence is implemented within Arjuna using replication and atomicity is provided based on read/write locking.

## 8 Extended Relational

Starburst [18] extends relational databases in five areas, external data storage, storage management, access methods, abstract types, and complex objects. Many of the issues that Starburst addresses 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 BOSS are what Starburst calls “exogenous” access methods. Starburst adds new functionality with specialized components “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 BOSS architecture which is designed to add new functionality in a new storage class. In BOSS the storage class can define its own storage layout and access methods as Starburst allows. In addition, a BOSS 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 BOSS storage class actually extends the BOSS 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 [20]. 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. BOSS’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.

## 9 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 BOSS, 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. BOSS uses this facility to accelerate recovery and simplify transaction management data structures. LFS [17] keeps all data in a log, significantly reducing the cost of a write. The disadvantage of a log is that it does not support random access. The file location information must be kept stable resulting in a major overhead cost 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 in a distributed persistent heap is much more complex than segment compaction [14].

AFS is interesting for its distribution properties. AFS 3.x [12] 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 BOSS 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 [9] 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. BOSS 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 BOSS can tolerate failures with out any special means.

## References

- [1] M. Atkinson et al. The persistent object management system. Technical Report PRRR-1, The Universities of Glasgow and St. Andrews, 1983.
- [2] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O<sub>2</sub>*, chapter 1. Morgan Kaufmann, 1992.
- [3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Westley, 1987.
- [4] A. Brown and J. Rosenberg. Persistent object stores: An implementation technique. In *The Fourth International Workshop on Persistent Object Systems*, 1990.
- [5] M. 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.

- [6] G. Delott. Performance improvements in the observer object server. Masters thesis, Department of Computer Science, Brown University, 1989.
- [7] G. Dixon, G. Parrington, S. Shrivastava, and S. Wheeler. The treatment of persistent objects in arjuna. In *Proceedings of the Third European Conference on Object-Oriented Programming*, July 1989.
- [8] J. Eppinger, L. Mummert, and A. Spector, editors. *Camelot and Avalon*. Morgan Kaufmann, 1991.
- [9] M. Franklin and M. Carey. Client-server caching revisited. In *Int'l Workshop on Distributed Object Management*, Edmonton, Canada, 1992.
- [10] M. Herlihy. Apologising versus asking permission: Optimistic concurrency control for abstract datatypes. *ACM Trans. on Database Systems*, 15(1):96–124, March 1990.
- [11] M. Hornick and S. Zdonik. A shared, segmented memory system for and object-oriented database. *ACM Transactions on Office Information Systems*, 5(1):70–85, January 1987.
- [12] J. H. Howard et al. Scale and performance in a distributed file system. *ACM TOCS*, 6, 1988.
- [13] B. Koch et al. Cache coherency and storage management in a persistent object system. In *The Fourth International Workshop on Persistent Object Systems*, 1990.
- [14] E. Kolodner, B. Liskov, and W. Weihl. Atomic garbage collection: Managing a stable heap. In *ACM SIGMOD Proceedings*, 1989.
- [15] C. Mohan et al. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1), 1992.
- [16] J. E. B. Moss. The mneme persistent object store. Technical Report TR 89-107, COINS, University of Massachusetts at Amherst, October 1989.
- [17] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *13th SOSP*, 1991.
- [18] P. Schawarz et al. Extensibility in the starburst database system. In *Intl. Workshop on Object-Oriented Database Systems*, 1986.
- [19] E. Shekita and M. Zwillig. Cricket: A mapped, persistent object store. In *The Fourth International Workshop on Persistent Object Systems*, 1990.
- [20] M. Stonebraker. The design of the postgres storage system. In *Proc. 13th VLDB*, 1987.
- [21] W. Weihl and B. Liskov. Implementation of resilient atomic data types. In *ACM Transactions on Programming Languages and Systems*, April 1985.