On the Use of Asynchrony in Achieving Extensibility and High Performance in an object Storage System

David E. Langworthy

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-22
July 1995

On the Use of Asynchrony in Achieving Extensibility and High Performance in an Object Storage System

by

David E. Langworthy

B. S., Southern Methodist University, May 1988

Sc. M., Brown University, May 1992

May 1995

This dissertation by David E. Langworthy
is accepted in its present form by the Department of
Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Stanley B. Zdonik

Recommended to the Graduate Council

Date _____ _____

Robert Netzer

Date _____ _____

Thomas W. Doeppner

Approved by the Graduate Council

Date _____ _____

This work is dedicated to my grandparents:
Asher Clinton and Georgia Hodges Langworthy
David Huston and Thelma Hemphill Evans

Abstract

An object store provides persistence for an object-oriented database or a special-purpose application such as CAD or hypermedia. In this thesis we examine the architecture of an object store in light of the following developments: the cost of network latency is increasing rapidly in relation to other I/O costs, users demand increased availability without the added cost of redundancy, and new application domains require specialized secondary search structures for high performance.

We determined that these problems could not be solved within any existing object store architecture: solving all of them required fundamental changes in the design of the object store. We present a novel architecture which takes advantage of asynchrony to alleviate latency problems, increase availability, and enable extensibility. We analyze the performance of our prototype to demonstrate the feasibility of such an approach.

Our asynchronous cache-coherency protocol in combination with client disk caching greatly reduces the cost of network latency. However, without the correct invalidation protocol, an unacceptable number of aborts results. We experimentally evaluate several invalidation protocols and introduce a new protocol that provides both low network overhead and fewer aborts.

We introduce a recovery algorithm that improves on the popular and efficient write-ahead logging. No-undo, no-redo write-ahead logging requires no updates to the database partition during recovery, reducing recovery to a single analysis scan of the log. The redos are applied on demand as the objects are read or asynchronously

after recovery.

Our novel architecture allows new secondary storage structures to extend the client interface. The new structure is encapsulated in a storage class. The storage class abstraction integrates the new structure with transaction management to provide semantically consistent atomic transactions.

Vita

David Langworthy was born in Kansas City, Missouri in 1964 and graduated from the Loretto Academy in 1982. In 1988 he received a Bachelor of Sciences degree *cum laude* from Southern Methodist University in Dallas, Texas. He received his Master's degree from Brown University in 1992 and his doctorate in 1995.

Acknowledgments

There is not one colleague, friend or relation that I have had over the years that has not in some way contributed to this work. My brother George and my parents Mary Ann and George, Sr. have been incredibly supportive and encouraging. I should also thank my virtual grandparents Jessie and Bill Benton.

I've had the pleasure to work with several great educators. David Wells got me thinking about graduate school and supported me in the begining. David Wallace, Shawn McMahn, and Ruth Illmer gave me the basic learning tools.

My officemate Ted Leung deserves thanks for many interesting discussions and the use of his extensive reference collection. Carl Childs, David Lively, Roberto Bayardo and S. Sairam have been good friends and at one time or another roommates of mine.

My thesis committee, Tom Doeppner and Rob Netzer, provided many useful comments on this thesis and Trina Avery and Judy Ziajka turned it into English.

I've saved my advisor, Stan Zdonik, for last. Anyone who has worked closely with Stan will know why he deserves special appreciation. Stan has guided my research, told me jokes, given me excellent advice, and even helped me keep my car running. Without him this work would not exist.

Contents

1 Introduction **1**

1.1 Thesis Statement 2

1.2 Contributions 4

1.3 Thesis Overview 5

2 Background **7**

2.1 ObServer 7

2.2 The Persistent Object Management System 8

2.3 The Exodus Storage Manager 9

2.4 Concurrency Control 10

2.5 Cache Coherency 11

2.6 Recovery 13

3 High-Level Architecture **16**

3.1 Computation Model 16

3.2 Modularity 17

3.3 Asynchrony 19

3.4 The Object 20

4 The Storage Class **21**

4.1 Local Concurrency Control 23

4.1.1 Storage Class Client 25

4.1.2 Storage Class Server 26
4.2 Example Interactions 27
4.2.1 The Storage Class and Transaction Manager 27
4.2.2 Storage Class Internals 29
4.3 The Fixed-Sized-Segment Storage Class 30
4.4 Designing a Storage Class 34
4.5 The Notification Storage Class 38

5 Global Transaction Management 43
5.1 Concurrency Control 44
5.2 Cache Coherency 45
5.3 Persistence 46
5.4 Recovery 47
5.5 Data Structures 49
5.5.1 The Log 49
5.5.2 The Version Table 50
5.5.3 The Transaction Table 50
5.5.4 Operation Records 52
5.5.5 Prepare and Commit Records 52
5.6 Application Process 53
5.7 Examples 54
5.7.1 Update 54
5.7.2 Background Processing Example 56
5.8 Algorithms 57
5.8.1 Lazy Updates 57
5.8.2 Prepare and Commit 59

6 Memory Object Management 62
6.1 MOM Architecture 63

6.2 The Memory Object 64
6.3 Memory-Management Policies 64
6.4 Application Interface 68
6.5 Example B-Tree Memory Management 71

7 Cache Coherency Experiments 72
7.1 Assumptions About Contention 73
7.2 System Configuration 74
7.3 Client Disk Caching: Experiment #1 74
7.4 Results of Experiment #1 76
7.4.1 No Invalidations and No Contention 76
7.4.2 No Invalidations and Minimal Contention 76
7.4.3 Invalidate on Commit and Minimal Contention 78
7.4.4 Invalidate-on-Commit and Low Contention 78
7.5 Invalidate-on-Abort 79
7.5.1 Invalidate on Abort and Minimal Contention 80
7.5.2 Invalidate on Abort and Low Contention 80
7.5.3 Conclusion of Experiment #1 81
7.6 Higher-Cost Transactions: Experiment #2 82
7.6.1 Minimal Contention 83
7.6.2 Low Contention 84
7.7 Experimental Conclusions 85

8 Related Research 86
8.1 Object and Page Servers 86
8.2 Object-Oriented Databases 90
8.3 Extended Relational Databases 91
8.4 File Systems 92

9 Summary 94

Bibliography **96**

List of Figures

2.1 The messages required for synchronous cache coherency. 11
2.2 The messages required for asynchronous cache coherency. 12
2.3 Recovery passes in traditional WAL. 13

3.1 An overview of the BOSS distributed environment. 18
3.2 Layers and vertical partitions. 19

4.1 The fixed storage class module interface. 26
4.2 The transaction manager and a storage class. 30
4.3 Inside a storage class. 31
4.4 The fixed storage class module interface. 33

5.1 Operation records in the version table and transaction table. 47
5.2 Components of the current state of an object. 49
5.3 Asynchronous versus synchronous recovery. 50
5.4 Mapping the log into virtual memory. 51
5.5 Chaining intention records. 53
5.6 Operations in the log. 54
5.7 A complete simple transaction in the log. 55
5.8 A sample update 57
5.9 Background processes brining system to a coherent state. 58

6.1 A layered view of a storage class. 66

6.2 MOM object mapping. 68
6.3 A two handed clock for object replacement. 68
6.4 The states and transitions of a memory object 71
6.5 The storage class interface to the MOM module. 72

7.1 Three benchmark workloads. 78
7.2 No invalidations with no contention. 80
7.3 No invalidations with minimal contention. 80
7.4 Invalidate on commit with minimal contention. 81
7.5 Invalidate on commit with low contention. 82
7.6 Invalidate on abort with minimal contention. 83
7.7 Invalidate on abort with low contention. 84
7.8 Invalidate-on-commit. 86
7.9 Invalidate-on-abort. 86
7.10 Invalidate-on-commit. 87
7.11 Invalidate-on-abort. 87

Chapter 1

Introduction

The confluence of technological advancements and new application requirements demands a reanalysis of the architecture and implementation of object storage systems. An object store provides persistence and distribution as a substrate of an object-oriented database (OODB). Developing the first generation of object stores was a challenge because OODBs extend the traditional notion of a database with an integrated, computationally complete data definition and programming language [43, 7, 33]. The demands these extensions place on the object store required entirely new architectures. Research into the architecture of object stores yielded the important results covered in Chapter 2, but this work needs to be reevaluated in light of three factors.

First, improvements in network bandwidth cannot yield corresponding improvements in overall system performance without protocols tuned to new cost factors. I/O latency is already a problem for distributed systems and the problem is becoming more acute. Bandwidth is increasing in proportion to other system resources such as memory size and CPU speed. However, latency is not improving at the same rate, causing an imbalance that increases the cost of latency relative to other system resources.

Second, users now require not only that their data persist after a failure, but also

that it be available shortly after or even during the failure. A group of designers working against a deadline will not tolerate long downtimes for their CAD system. The problem of availability is even more acute in a computer-integrated manufacturing system where the database must respond to hard real-time constraints. High availability can be achieved through redundancy [27, 37], but the cost quickly becomes prohibitive. Highly available systems will not become widely used unless the resource requirements are reduced.

Third, applications such as hypermedia, geographic information systems, and the human genome project require new storage structures that cannot be supported efficiently by existing object stores. For example, geographic information systems require efficient multidimensional dynamic search structures for secondary storage, an area of active research [18, 16]. OODBs support new abstractions on top of the database, but the performance advantages of new secondary search structures must be realized not in a layer above the database, but in new secondary search structures incorporated into the OODB.

1.1 Thesis Statement

During the research leading to this thesis we investigated solutions to the above problems. We determined that addressing of them required not simply augmenting an existing object store, but rather making fundamental changes in the design of the object store. This investigation addresses the following hypothesis:

Asynchrony solves many problems facing object storage systems.

- It alleviates the network I/O bottleneck.
- It increases availability.
- It makes possible extensibility.

Asynchronous systems differ fundamentally from traditional database implementations built on a synchronous computation model. Asynchronous systems do not depend on the timing of operations and thereby introduce the potential for opportunism. Further, there is less dependence among modules, which facilitates the integration of new storage structures.

High performance in a distributed system requires strict control of I/O costs. Cache coherency in a traditional client/server database architecture is based upon many small synchronous operations [14]. In a centralized system, communication is not an issue, but in a distributed system, these synchronous operations result in numerous small blocking messages. These messages are particularly expensive because they must be sent immediately and the cost of initiating a message cannot be amortized over a reasonable amount of data. Transforming them into asynchronous messages allows them to be delayed and bundled into larger, less expensive messages.

A system failure in a traditional database system requires that all operations pending at the time of the failure be executed synchronously before the database can begin servicing a new request even if the request is for an object with no pending operations. An asynchronous system tolerates this harmless inconsistency and can begin servicing requests before all pending operations are executed. This flexibility is an advantage during normal operation as well, because during peak periods operations on less important portions of the database can be delayed without sacrificing correctness.

Constraints on the timing and ordering of operations in a synchronous system module restrict the range of possible implementations. An asynchronous system specifies only what needs to be done, without specifying how it is to be done in the form of orderings or timings. Asynchrony frees the designer to implement a new module in whatever way best suits its abstraction. For example, once asynchrony is introduced into cache coherency, not only can small control messages be bundled, as mentioned above, but also new asynchronous cache-coherency protocols can be developed to

better exploit the opportunities asynchrony permits. The protocol we developed is introduced in the next section.

1.2 Contributions

We investigated the benefits of asynchrony in the design and implementation of the Brown Object Storage System (BOSS). BOSS demonstrates the feasibility of loosely coupled asynchronous architecture. Our investigations resulted in contributions in the areas of cache coherency, recovery, concurrency control, and object store implementation.

1. **Multiple Cache-Coherency Policies**

   BOSS does not depend on cache coherency for transactional correctness. This provides ultimate freedom in choosing a cache coherency policy. It is likely that for common workloads it is desirable occasionally to let the cache drift out of synchronization because communication costs will be greatly reduced. Reducing the cost of maintaining client caches makes it possible to cache large amounts of data for long periods of time on the client's local disks. In addition, allowing the caches to drift temporarily means that communication failure can be tolerated and may eventually allow support for disconnected, mobile operation.

2. **Efficient Recovery Technique**

   BOSS's recovery algorithm improves on the popular and efficient *write-ahead-logging* [17]. BOSS's *no-undo, no-redo write-ahead-logging* does not require any updates to the database partition during recovery, reducing recovery to a single analysis scan of the log. The redos are applied on demand as the objects are read or asynchronously after recovery.

3. **Local Concurrency Control**

   BOSS applies the theory of local concurrency control to separate concurrency control and storage management concerns [46]. Objects in BOSS are fully abstract with respect to the transaction manager, a model that supports all existing database access methods as well as any that may be invented in the future. No other database system has applied this theory in this way.

The primary software artifact of this work is the BOSS implementation. The implementation operates in a network of Sun Sparc10 workstations running SunOS

4.1. We have used the implementation as an experimental testbed to explore the advantages of client disk caching and also different cache-coherency protocols. These investigations led to the design of a new cache coherency protocol called "invalidate on abort".

1.3 Thesis Overview

This first chapter describes the challenges and contributions of this thesis. The next chapter contains background information for readers unfamiliar with the field. It introduces two broad areas. First, the chapter reviews the history of work in object storage which began when it became clear that flat files did not offer enough structure and relational databases incurred too much overhead for new application domains. Second, the chapter describes asynchrony as it is used in the BOSS design and implementation.

The third chapter begins the technical portion of the thesis. It contains a high-level overview of the BOSS architecture and introduces the concept of a storage class. Chapter 4 describes how a storage class interacts with the rest of the system, how a storage class is constructed, and how to add a new storage class. Chapter 5, Global Transaction Management, describes the BOSS transaction manager design which exploits user defined atomic datatypes. The final architecture chapter, Chapter 6 on Memory Object Management, describes how BOSS supports the addition of a new storage class by providing high-level support for memory management.

Chapter 7 contains the results and analysis of performance experiments. After the BOSS prototype was complete, we performed experiments on its asynchronous cache coherency protocols. These experiments led to a new cache coherency protocol that aggressively exploits asynchrony.

Chapter 8, comparing BOSS to other systems, shows that BOSS differs from other efforts in two main areas: it is extensible and it has an asynchronous architecture.

New storage structures allow support for new application domains and exploit semantics for higher performance. Objects can fall out of synchronization without impacting correctness, a nondeterminism that offers many opportunities for optimization unavailable in a synchronous system.

We have found the BOSS architecture quite robust. Several developments have come along after the design and implementation of the prototype that fit well into the BOSS framework. The final chapter summarizes the contributions of this research.

Chapter 2

Background

In the interest of completeness, we present the history of object storage technology. In addition, we describe algorithms for two BOSS subsystems, cache coherency and recovery.

2.1 ObServer

ObServer was the first object server our group developed [9, 13, 21]. The project had two main objectives: to support efficient traversal of an object space that supports identity [4] and to provide the primitives to support cooperative transaction synchronization for design environments [34, 24].

An object in ObServer consists of a unique object identifier (OID) coupled with a variable-size block of bytes. The OID labels the object's data and provides clients with a handle to the object. Because ObServer was intended to support different types of systems and object formats, ObServer has no knowledge of the structure of an object. Only client applications can interpret the semantics of an object.

Efficient traversal of a highly connected object space is a serious problem in a relational database [44]. ObServer minimizes the cost of navigating a complex graph by clustering related objects together on disk so they can be retrieved as a group.

Because it does not know the semantics of its objects, ObServer cannot determine the best groupings. Rather, it provides the segment abstraction as a unit of physical clustering whose contents are guaranteed to be stored contiguously on disk.

Traditional database systems provide a competitive transaction abstraction to group primitive database operations. Transactions compete for resources and the job of the database is to keep a transaction from observing the intermediate results of other transactions [3, 29, 36]. Design environments have a different set of criteria. Two designers should be able to cooperate interactively as they work: the designer of a support structure for an aircraft wing should be able to interact with another designer working on the control surfaces. ObServer provides an extended lock set that allows transactions to cooperate in a controlled manner.

2.2 The Persistent Object Management System

The approach ObServer takes to the design of a persistent object store is to strip away unnecessary functionality from a traditional database to yield an efficient, unrestrictive kernel on which advanced applications can be constructed. Another approach is to augment the runtime structures of a high-level programming language to provide persistence and transaction semantics. This is the approach taken by the Persistent Object Management System (POMS) [1].

POMS extends the runtime virtual memory heap into stable storage. Because the database is an extension of a language, the database objects parallel the runtime objects. The objects are structured and typed, unlike ObServer's untyped, unstructured objects. Some issues this approach raises are the need to type-check data that may exist beyond the life of the program that created it and the need to reconcile stable memory locations with virtual memory pointers. ObServer does not face these issues, but anyone designing a language on top of ObServer eventually must deal with them.

POMS provides support for transactions by using shadow paging. As a transaction progresses, the original copies of objects are not modified. Rather, shadow copies of objects are created. Upon completion of a transaction the shadow copies replace the originals through careful pointer manipulations and then the space used by the originals is reclaimed. POMS transactions provide atomicity and recoverability for a single user, but do not confront the issues of multiuser concurrent access.

2.3 The Exodus Storage Manager

The two systems previously described are persistent programming languages with specialized object stores. The Exodus project [7, 39, 5] offers a persistent programming language, E, built on top of the Exodus Storage Manager (ESM), a standalone object store [6]. Several other OODBs use ESM for persistent storage [47, 2]. ESM provides applications with the abstraction of objects, but the unit of transfer between the client and server is the page. For this reason, ESM is a page server rather than an object server, which moves objects or segments between the client and server.

The objects that ESM builds on top of pages are divided into two groups: large and small. Small objects fit within a page. The identifiers of these small objects reference the page and an index in the page, a level of indirection that allows small objects to move with a page. Large objects are built using a B-tree optimized for large insertions and deletions and for appending to the "end" of the tree.

In addition to functioning as the unit of transfer, pages are also the unit of concurrency control and recovery. ESM uses a pessimistic two-phase locking protocol. When a client wants to read or modify an object on a page, it requests the appropriate lock on the page from the server. When a page is initially fetched from the server, a shared lock is implicitly granted, and, at the end of a transaction, all locks are released [10]. ESM generalizes the ARIES recovery system for a client-server environment [15].

2.4 Concurrency Control

These next three sections present algorithms rather than specific systems. This section discusses the optimistic, user-defined concurrency-control protocol used in BOSS. The next section describes the difference between a synchronous and asynchronous cache-coherency protocol (BOSS actually has the ability to use either). The final section presents a standard means of crash recovery.

Traditional database concurrency control provides correct concurrent operation for only one or a small set of system defined datatypes, record, page, file, B-tree, etc. User-defined atomic datatypes allow implementors to design their applications using the best suited datatypes [46], the advantage being that higher levels of concurrency can be achieved than by using only system-defined datatypes.

The correctness of transactions is a global property because it is a constraint across all operations on all objects. However, correctness of an individual datatype is local: it concerns only operations on objects of that datatype. The theory of atomic datatypes separates local from global correctness and thus allows new, user-defined datatypes with local correctness criteria to augment the system.

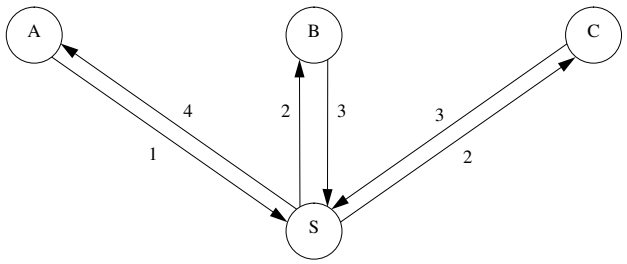
The basic requirement of atomic datatypes is that all types must serialize transactions in the same order. The criteria used in BOSS, dynamic atomicity, serializes transactions in the order in which they commit. Of the several global criteria discussed in [29], we chose dynamic atomicity because it covers both two-phase locking and the optimistic protocol that makes asynchrony possible in BOSS.

BOSS uses conflict-based backward validation to maintain global correctness [20]. This protocol checks committing operations against previously committed operations. The alternative, forward validation, checks committing operations against operations that intend to commit. Under backward validation, if there are no committed operations that could have changed the outcome of the committing operation, then there is no conflict and the operation is allowed to commit. If all operations in a transaction are allowed to commit and there are no system failures, the transaction commits.

2.5 Cache Coherency

Cache coherency is the most notable example of the difference between a synchronous and an asynchronous algorithm. The goal of cache coherency is to keep the states of all objects in the different caches synchronized with each other and with the stable state. The following examples illustrate the messages required to update an object X that is already cached at several clients. The first example (Figure 2.1) follows the callback locking protocol with invalidations [22, 14]. This protocol is synchronous in that all of the copies of an object have exactly the same state. The second example (Figure 2.2) follows the protocol used in BOSSan asynchronous protocol that allows the different copies of an object to drift out of synchronization.

Figure 2.1 shows three clients A, B, and C and one server S. To begin with, all three clients cache the object X. The diagram shows only the messages between the nodes in the network and illustrates none of the contents of the nodes. Under the callback locking protocol, possession of an object implies the ability to read the object. Thus, X can be read by the clients without any messages to the server.

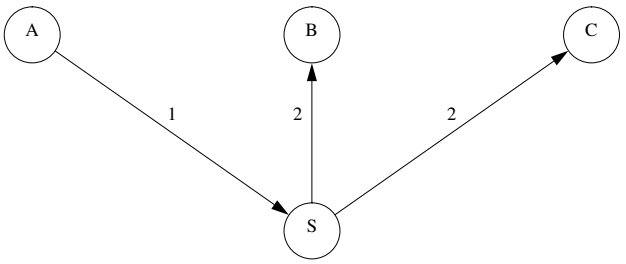


0. Clients A, B, and C possess an object X.
1. Client A requests a lock on X during transaction and waits for response.
2. Server requests an invalidation from the other clients holding X and waits for a response.
3. Clients B and C acknowledge request sometime later.
4. Server grants client A an exclusive lock on X.

Figure 2.1: The messages required for synchronous cache coherency.

In the synchronous case, client A must acquire an exclusive lock on X before an update. The lock acquisition requires one round trip to the server. At the server, two concurrent round trips are required to invalidate the copies of X at B and C. The total cost of acquiring one lock is two round-trip delays and, more importantly, six small messages on the network.

The asynchronous alternative in Figure 2.2 begins with the same initial configuration. There is a copy of X at each client. This alternative tolerates differences between the caches, so that client A simply updates X without any messages to the server. At this point the copies of X at B and C are not synchronized with the copy at A. To commit the update and indirectly invalidate the other copies, A sends a message to the server. Once the server learns that X has been updated, it invalidates the copies at B and C. Since this operation need not happen immediately, it is delayed and combined with other invalidations to avoid unnecessary small messages.



0. Clients A, B, and C posses an object X.
1. Client A notifies server of an update to X sometime after the update.
2. Server notifies clients B & C of the update to X sometime later.

Figure 2.2: The messages required for asynchronous cache coherency.

The total cost for the asynchronous case is two messages, neither of which is time-critical. The second message is guaranteed to be large and therefore relatively inexpensive compared with the numerous small messages it replaces. The first message is likely to be of reasonable size because a transaction usually updates more than one object. Note that, the synchronous case also sends a commit message not shown

in the diagram.

In this example, asynchronous cache coherency is clearly preferable to synchronous cache coherency, but asynchrony introduces a potential risk. The database is not perfectly synchronized; therefore, any transaction that happens to observe an inconsistency could generate an erroneous result. To avoid an erroneous result, a transaction that observes an inconsistency must be aborted and retried. This is not a great problem for low-contention workloads, but for workloads with high contention the cost of the retries is greater than the savings in messages. Chapter 7 discusses this topic.

This example shows the differences between synchronous and asynchronous cache coherency: the asynchronous protocol accomplishes the same task as the synchronous protocol only a little later and using significantly fewer messages.

2.6 Recovery

Recovery is another area in which a significant difference exists between synchronous and asynchronous protocols. Traditional write-ahead logging (WAL) is a synchronous protocol [17] that synchronizes all copies of every object before normal operation begins. BOSS introduces an asynchronous algorithm that does not have to modify any objects to achieve synchronization (See Chapter 5).

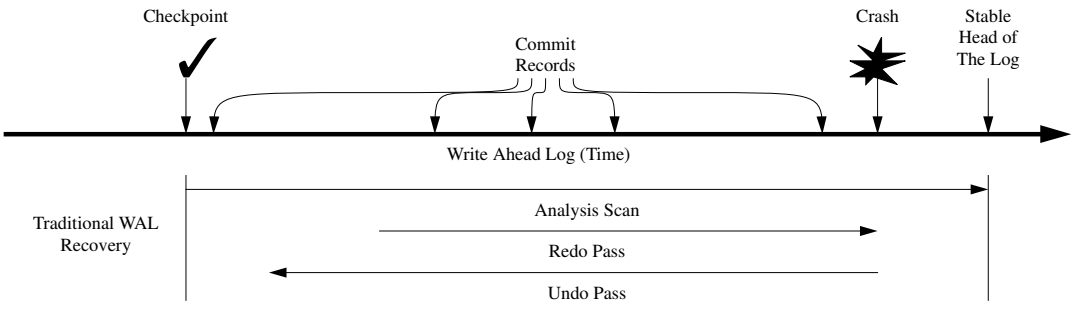


Figure 2.3: Recovery passes in traditional WAL.

Traditional write-ahead logging (WAL) places potential changes (redo records) and a means to correct the changes in the event of a problem (undo records) into a

log before the change is applied to the database. After a failure, the log is analyzed to determine which records need to be redone and undone. Once the analysis is finished, the redo pass synchronizes the entire database with the state at the time of the failure. However, this state may reflect some partial changes from transactions that were terminated as a result of the failure. These partial changes are corrected in an undo pass that synchronizes the database not with any particular wall-clock time, but rather with the internal system time just after the last transaction commit before the failure. All three of these passes require scanning the log and the last pass requires updates to both the log and the database.

ARIES (Algorithm for Recovery and Isolation Exploiting Semantics), a variant of a write-ahead logging protocol, developed at IBM Almaden Research Center [32] supports faster recovery in a pessimistic system using a log sequence number (LSN). An LSN is the index of a log record that is used to efficiently determine the state of a page with respect to the stable state, the net effect of all committed transactions. Each page is tagged with the LSN of the last operation applied to that page. LSNs are assigned in an ascending sequence, so that determining whether an operation has been applied to a record or object on a page requires only checking the magnitude of the LSN of the object and the LSN on the page. If the LSN on the page is greater than or equal to the operation's LSN, the operation is reflected in the state of the page; otherwise, the operation is not yet reflected in the state of the page.

The POSTGRES Storage Manager [45] supports transaction management without using a conventional WAL. POSTGRES uses an update log with a large amount of non-volatile RAM to avoid synchronous writes at commit and attain instantaneous recovery. The algorithm does not require the non-volatile memory; however, its performance degrades to the point that it is significantly more expensive than WAL to operate during normal processing. The BOSS recovery algorithm builds on this general approach, but provides the efficiency of write ahead logging during normal processing and nearly the efficiency of POSTGRES during recovery without any

special hardware (See Chapter 5).

Chapter 3

High-Level Architecture

BOSS takes a new approach to the design and implementation of an object store. BOSS is a complete working system in which all the issues have been addressed, but in many cases in a different manner from in a traditional object store. BOSS has an asynchronous, modular architecture that offers extensibility and high performance in a client-server network of advanced workstations.

3.1 Computation Model

Early object servers and page servers offered support for multi-client, single-server operation. To support more users, provide access to more data, and take advantage of advances in hardware technology, however, object servers have become fully distributed, multi-server systems.

Figure 3.1 shows three end-user applications: a VLSI tool, a CAD tool, and a hypermedia application. Each of these is implemented using a client OODBimplemented on top of BOSS. Applications on the client machines communicate with BOSS via a client stub, one BOSS client stub per client. The client stub runs in the address space of the OODB and the protection boundary between the database and the application is implemented by the OODB. The OODB calls operations on the stub of a storage

class that result in computation at the client or communication with the server. Not all operations result in communication with the server. The client communicates with the server once to fetch the object and once each time a transaction that read the object commits, no matter how many times the object is read or how many transactions read the object.

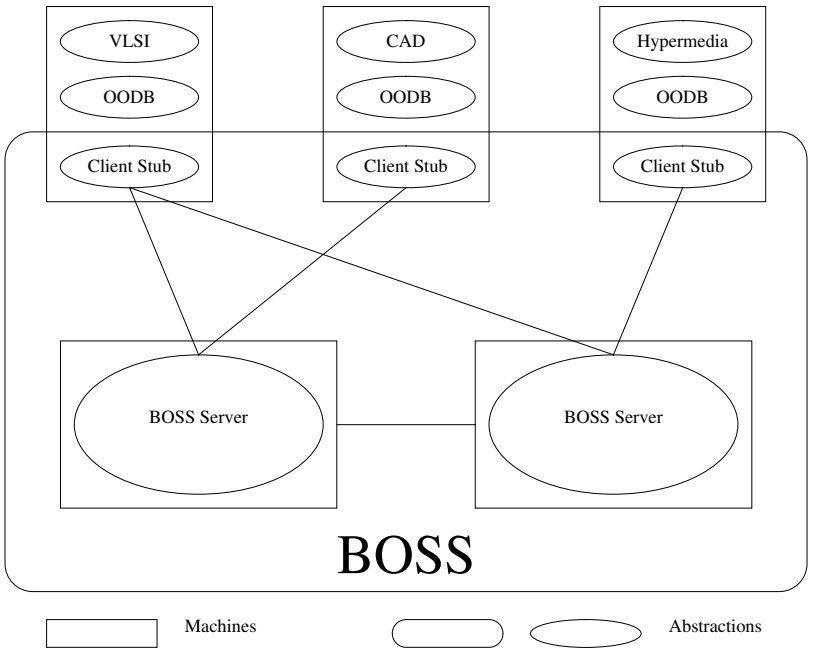


Figure 3.1: An overview of the BOSS distributed environment.

In our computation model, the client explicitly contacts each of the servers from which it needs information and the servers communicate with each other only to synchronize for distributed commit. BOSS's function is to provide persistent storage for objects. It does not aim to support general distributed computation, so server-to-server communication is not required.

3.2 Modularity

BOSS's internal interfaces are fixed and well defined, while the architecture allows extension of the interface BOSS presents to client applications, BOSS lets new storage

structures be added to the system as storage classes. A storage class is a vertical slice through the layers in the memory hierarchy (See Figure 3.2). Each piece of this slice has a well-defined interface. The external interface it presents to the client OODB depends on the semantics of the storage class. The rounded rectangles are abstract data types (ADTs) that span the client and server: each rounded rectangle contains the operations that the ADT provides to the client application.

The topmost layer in Figure 3.2 operates on the client machines. This layer contains only volatile information, as client machines are not fault tolerant. Clients operate on copies of objects in their caches. The server implements fault tolerance on non-volatile storage and maintains the primary copy of all objects.

Figure 3.2 shows an abstract view of BOSS configured with two storage classes: segments and B-trees. Segments consist of any number of variable-sized byte streams that support fetch, read, write, and create operations. B-trees contain associations between values and objects and provide insert, delete, find, and create operations.

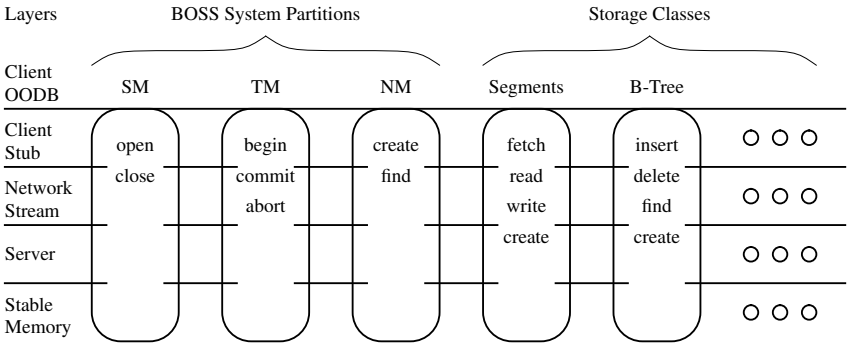


Figure 3.2: Layers and vertical partitions.

Figure 3.2 also shows how BOSS vertically divides basic system services. A session manager (SM) is responsible for maintaining connections to active clients and other servers. The transaction manager (TM) synchronizes the operations on all objects in all the different classes. The name manager (NM) allocates names that the classes use to identify objects and assists in object location. The basic system services are the same in all configurations of BOSS; only the storage classes change.

3.3 Asynchrony

In a distributed system, communication overhead can become a bottleneck. Increasing network bandwidth alleviates the problem to a certain extent; however, network latency is not improving in proportion to bandwidth [8]. The cost per unit of information is significantly greater for a small message than for a larger message. Another important cost measure is the round-trip latency for synchronous communication, which is the basis of many concurrency-control and cache-coherency algorithms. A synchronous communication consists of a remote call and a return message. Such communication is particularly expensive because the sending process is blocked until the communication completes (while blocked, it cannot accomplish any other work).

Failure presents another problem in a distributed system, where many components can fail independently. When one component fails, the rest of the system needs to continue processing. If the failed component is on the path of a synchronous communication, the sender is blocked until the component recovers; that is, the failure propagates to the sender.

Asynchrony is a useful tool for addressing problems stemming from communication latency and partial failure. Asynchrony is the ability to let related elements of the database evolve independently, that is without synchronization. Thus, two or more values that are logically updated together are allowed to physically change at different rates. Asynchrony is particularly advantageous in a distributed system where the communication cost to synchronize remote portions of the database is high. An additional benefit of asynchrony is extensibility: because an asynchronous system has fewer dependencies, greater modularity can be achieved.

BOSS's optimistic concurrency-control protocol does not enforce synchronization, but rather checks for the appearance of synchronization at the end of each transaction. If the transaction observed objects that were not synchronized with the stable state, the transaction is aborted. Several protocols to avoid this problem are explored in Chapter 7.

3.4 The Object

The modular, asynchronous architecture is reflected in the common attributes of all BOSS objects. Every object has a unique object identifier (OID) and a timestamp. The timestamp allows BOSS to detect copies of objects that are out of synchronization with the stable state.

OID s in BOSS have the special property that the object's storage class can be extracted by the name manager. Typical object-oriented systems dispatch messages to methods based on the object the message is sent to. In BOSS the object need not be resident in the cache at the time a message is sent. Further, an object's storage class determines how the OID is dereferenced, so that, the object's storage class must be determined before it is dereferenced. This is accomplished by a tag in the OID that encodes the object's storage class. Since there are few storage classes, this tag is compact. The following chapter describes the internals of the storage class abstraction in detail.

Chapter 4

The Storage Class

New applications such as hypermedia, geographic information systems, and the human genome project require storage structures that are not directly supported by existing object stores. Implementing these structures efficiently on top of an object store poses performance problems because it requires access to most aspects of the object store's implementation: concurrency control, cache coherency, recovery, object location, and memory management.

One of the distinguishing features of BOSS is its ability to extend its interface with new abstractions via the storage class. This ability is important for two reasons: first, it gives clients higher-level access to their data and second, the storage class has access to the above aspects of the BOSS implementation.

Interfacing at a higher level gives BOSS knowledge of what clients are trying to do rather than just how they are trying to get it done. This knowledge opens opportunities for optimization that may be exploited through the interfaces BOSS provides to the storage class.

A storage class contains BOSS objects. All objects have a unique object identifier (OID). However, the interface the object exports is determined by the storage class, and is not restricted by BOSS. This novel feature has implications for concurrency control and recovery.

Each storage class defines its own local type-specific correctness criterion that is mediated by a global timestamp-based optimistic protocol. The advantages of type-specific concurrency control are discussed at length in [46]. In short, higher potential concurrency can be achieved by taking advantage of the semantics of the various operations that can be performed on an object. Concurrency control and recovery are performed on a per object basis within a class. The class determines whether each object was accessed correctly, and the transaction manager determines whether the entire transaction can commit. Recovery is also based on an object's semantics. The transaction manager logs the operations performed on an object, not the physical storage that was modified. Logging operations permits objects to be recovered individually, as discussed in Section 5.3.

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. There is a tradeoff between performing an operation at the client on a copy of an object and performing the operation at the server and sending the result to the client. Sending the object to the client has the advantage that the object can be cached. Performing the operation at the server requires that the object stay in the server buffers for a longer period of time and requires greater server CPU resources. BOSS allows the implementor control over object motion and location so that the best performance is achieved.

BOSS provides greater control over the physical aspects of an object's implementation than an OODB, but OODBsoffer some functionality that BOSS does not. BOSShas no way to compose or extend storage classes to create new storage classes. Further, objects store references to other objects in the form of OIDs, but the operations on storage objects cannot follow these links. This functionality must be provided in an OODB implemented in a layer above BOSS.

While these abilities are not traditionally included in an object store, they were considered for BOSS. Inheritance and composition were rejected because their inclusion would have entailed using an object model. Since one of the major advantages

of an object store is that it does not enforce the restrictions and overhead of one particular object model, these abilities were rejected.

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

This chapter describes the storage class and presents several examples. Section 4.1 presents the interfaces common to all storage classes. Then the dynamics of the interface between that storage class and the transaction manager (TM) are described. Section 4.3 presents the storage class in the base implementation. The final two sections discuss the addition of a new storage class and present as an example a notification storage class.

Extensibility requires that storage classes be easy to design and construct. This implies that one storage class should not be impacted by the existence or implementation of another storage class. Also, the implementation of a storage class should not be tantamount to the implementation of an entire object store. The final section of this chapter demonstrates the feasibility of extending BOSS by the addition of a storage class that exercises all of the major interfaces. This extension does not impact other storage classes or require a significant implementation effort.

4.1 Local Concurrency Control

The interfaces between the storage class and the rest of the system are critical for both performance and functionality. The goal behind separating the storage class abstraction from the object store's implementation is to provide the ability to improve

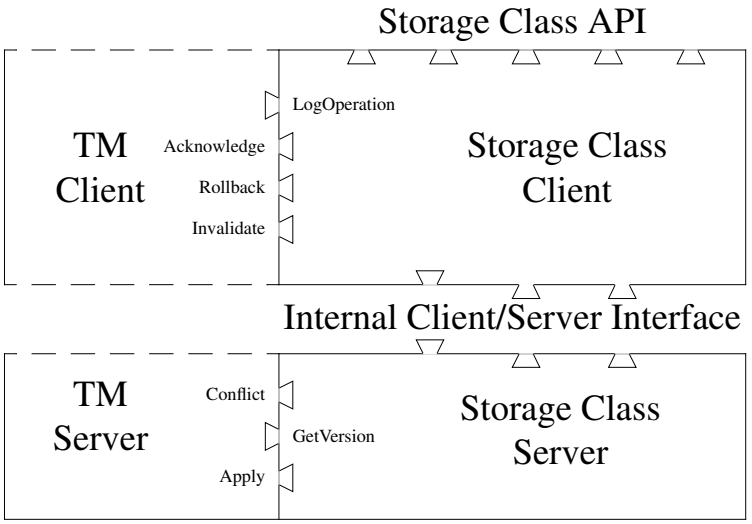


Figure 4.1: The fixed storage class module interface.

performance with specialized storage structures tuned to a particular application's needs. This goal requires a set of interfaces that allow a great deal of flexibility without undue overhead.

The goal is met in large part through an efficient implementation of user-defined atomic datatypes [46, 20]. The storage class cooperates with the transaction manager for correctness. The storage class implements a local policy suited to its semantics for which the storage class designer guarantees correctness. The transaction manager implements a global policy that coordinates the local policies to yield correct executions.

Figure 4.1 illustrates the storage classes interfaces. The boxes correspond to abstractions. A notch into a box is an interface the abstraction supports and a notch extending from a box is an interface the abstraction exploits. Two divisions are shown in the diagram, that between local and global concurrency control and that between client and server. The internals of the transaction manager (TM) are described in Chapter 5. Here we describe four storage-class interfaces: the client API, the internal interface between the client and server, the interface to the client TM, and the interface to the server TM.

4.1.1 Storage Class Client

The storage class client consists of three interfaces. The client API is unrestricted, as are the interface a storage class exports and the internal interface to the server portion of the storage class. The interface to the client portion of the transaction manager is structured to achieve atomicity for the storage classes' operations.

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. In return, the transaction manager informs the storage class client of global events that could affect the correctness of future operations. The remainder of this section describes the protocol by which this is done.

The storage class client implements the abstraction it provides to the client with two sorts of operations: transactional and server. Transactional operations are mediated by the TM for atomicity. Read-object and write-object operations are examples of transactional operations. Server operations do not affect the stable state of the database. A fetch operation that moves an object from the server into the client cache without allowing the client to view the object or modify its state is a server operation. Server operations are described fully in the next section.

The transactional operations performed by the storage class client are not passed directly to the server. Any operation that observes or modifies the stable state of the database must be mediated by the transaction manager. A storage class defines the operations that observe and affect the stable state. These operations are passed into the TM via the `LogOperation` entry point. Within the TM the operations are combined with operations from other storage classes and committed as an atomic group. Section 4.3 describes `LogOperation` for a common storage class.

The TM client informs the storage class about changes in an object's state via callbacks. BOSS does not prescribe the implementation of the callbacks, only their interface and effect on the state of the cache. There are three callbacks in the interface: `Acknowledge`, `Rollback`, and `Invalidate`. If a client's transaction commits,

`Acknowledge` is called once for each modified object. This invocation will contain the object's new timestamp. The storage class can then place the new timestamp in the object so the dirty copy of an old version becomes a clean copy of a new version. If a client's transaction aborts, `Rollback` is also called once for each modified object. However, this invocation must either discard the dirty object or revert it to its previous state in order to leave the cache in a state consistent with the current stable state.

`Invalidate` is called to inform a client that an object has been updated. BOSS does not rely on cache coherency for transactional correctness; and thus, this callback may occur at any time. Once `Invalidate` informs the storage class that an object in its cache is no longer valid, the storage class can then either discard the object or bring it up to date. Chapter 7 explores the tradeoffs involved with this performance-critical operation.

4.1.2 Storage Class Server

The storage class server exports two interfaces: the internal client-server interface and the interface to the TM server. The internal client-server interface at the server is the mirror image of the interface at the client. The storage class and the TM cooperate at the server to determine the atomicity of operations and to update the stable copy after a successful commit.

The TM has no knowledge of the semantics of the individual operations performed by the storage classes, and thus, it cannot determine their correct concurrent semantics in isolation. The TM relies on the `Conflict` operation for this task. Each storage class determines the correct concurrent semantics for its operations and encodes them in the `Conflict` operation. `Conflict` accepts a transactional operation as its one argument and returns true if its argument conflicts with a previously committed operation and false otherwise.

The TM assists the storage class with the `GetVersion` operation, which returns

the most recent timestamp of any object. The timestamp is maintained by the TM in a structure defined in Section 5.1.

If none of a transaction's operations conflict, the transaction commits its updates. Again, the transaction manager does not know the semantics of the individual operations. The operations are passed back into the storage class via the `Apply` entry point. Because BOSS has an asynchronous architecture, `Apply` need not be called during commit processing. A transaction's updates can be applied lazily in the background, as explained in Section 5.6.

4.2 Example Interactions

Two examples illustrate the interactions between the storage class and the transaction manager and the workings of the storage class itself. The first example, Figure 4.2, follows the course of one transaction, **T**, through some of its significant events. The second, Figure 4.3, follows one object, **O**, through a storage class. The boxes in the figures represent system components on both the client and server machines. The top part of each box is the client portion, the bottom part the server portion. The labels in the diagrams correspond to the labels in the text, and the text presents the events in the order in which they occur.

4.2.1 The Storage Class and Transaction Manager

1. **Begin:** Begin is the first message in a transaction. In response, a client stub creates a transaction identifier **T**.

2. **Operations:** The operations that a client OODB calls on the client stub can be different from those that are actually sent to the server. Also, the number of operations sent may differ from the number actually invoked on the client stub. For example, the OODB may invoke many write operations, but only one update is sent to the server.

3. **Intention Rec's:** The operations of **T** are sent to the server in intention records. These records are streamed to the server to reduce I/O at commit.

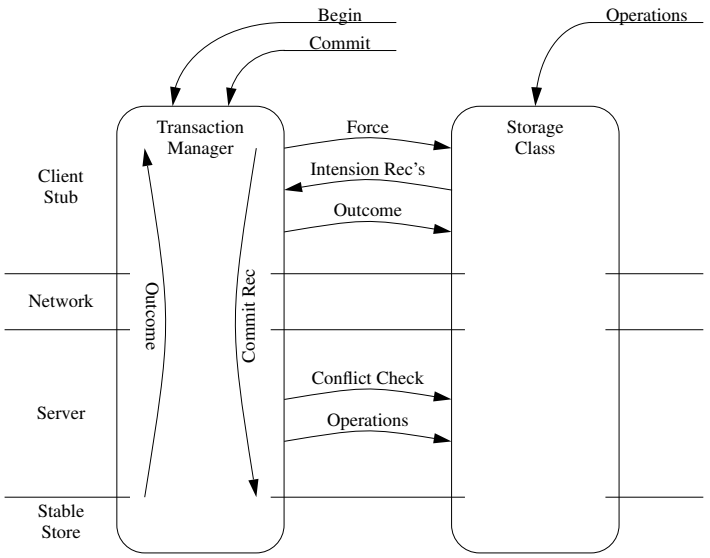


Figure 4.2: The transaction manager and a storage class.

4. **Commit:** To save the modifications made during the transaction, the client issues a commit message to the transaction manager. The client transaction manager then sends a force message to the storage class and a commit record to the server transaction manager.

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

6. **Commit Rec:** The final intention record is combined with a commit request for **T** and sent to the server. The server stores these operations in the transaction history.

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

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

9. **Operations:** After **T** commits, its operations may be applied at the server. These operations are of a restricted form described in Section 4.5.

4.2.2 Storage Class Internals

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

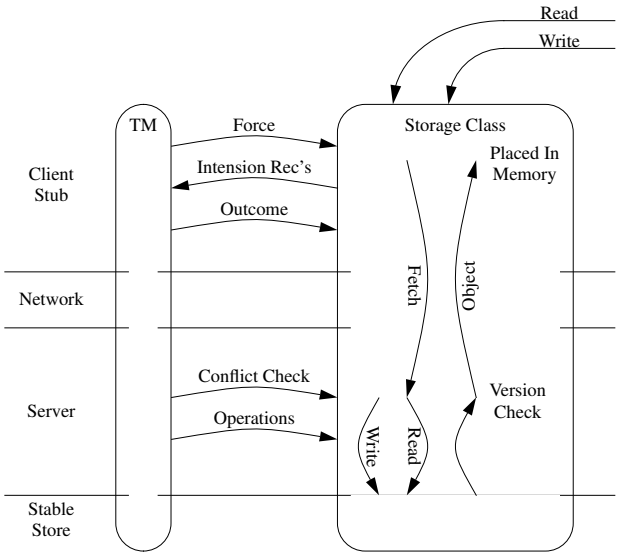


Figure 4.3: Inside a storage class.

1. **Read:** The first operation on **O** is a read from the client OODB. A read operation can be called only while the client OODB has a transaction running. BOSS allows objects to be cached between transactions, but in this example, **O** is not cached at the client stub.

2. **Fetch:** Because **O** is not cached, it must be fetched from the server.

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

4. **Version Check:** After **O** is in the server memory, the storage class checks that the copy of **O** that was read from disk is up to date. If not, it is brought up to date as described in Section 5.3.

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

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

7. **Write:** At a later point in the transaction, the client OODB writes a new value to **O**. The client's buffer now reflects this change. At this time, a record is made of the modification that is eventually included in the intentions list.

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

9. **Conflict Check:** Before the transaction commits the TM in the server calls conflict on **O** to determine whether the update was invalidated. If it was, the transaction is aborted.

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

4.3 The Fixed-Sized-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 [21].

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 as to determine the class of an object. The implementation of the Fixed storage class encodes the identity of the segment in the identity of each object it contains.

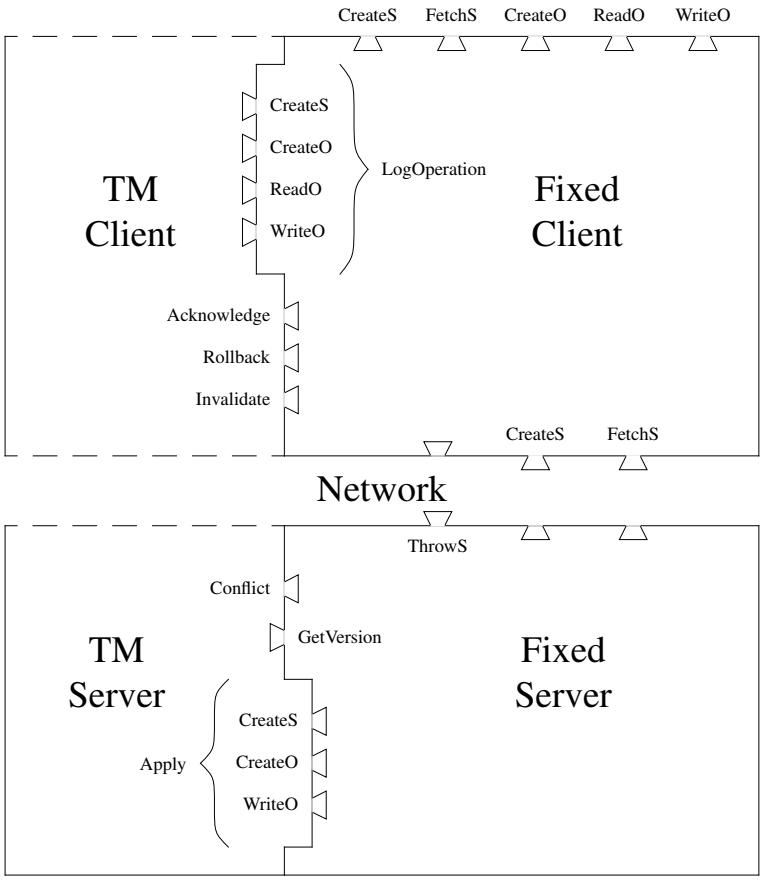


Figure 4.4: The fixed storage class module interface.

Figure 4.4 contains an expanded module diagram shows the operations that Fixed provides. This diagram fills in the client API and the transactional and server operations. The client portion of the Fixed storage class is responsible for mapping the client API to the transactional and server operations. In the diagram, Fixed's transactional operations are illustrated within the `LogOperation` entry point and its server operations are named in the interface between the client and server portions.

Any operation that observes or modifies the stable state of the database must be included as a transactional operation. `FetchS` brings a segment from the server into the client's cache. Since this operation does not affect the stable state, it is not included as a transactional operation. `FetchS` is implemented with one server operation of the same name.

In general, there does not need to be a one-to-one mapping from interface operations to transactional or server operations. For example, `FetchS` does not pass directly through the Fixed client abstraction. The client maintains a cache of segments and if the desired segment is already cached, the request is not passed on to the server. Chapter 7 explores several protocols to ensure this segment is up to date. Also, an object that is read and written several times during the course of a transaction requires only one `WriteO` operation.

The interface between the storage-class client and the storage-class server is actually a peer-to-peer interface. The server is capable of directing the client's actions and providing it with unsolicited information. There is one upcall from the Fixed server to the client, `ThrowS`, that moves a segment from the server to the client asynchronously and may be used to implement segment prefetching.

`ReadO`, `WriteO`, and `CreateO` are not included in the server interface. 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 simply logs the operations. If the target segment is not cached, this call generates a `FetchS` and waits for it to complete before logging the operations.

`CreateS` does not actually have to contact the server. A client stub could be implemented that generates 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.

At the server the transactional operations are passed back via the `Apply` entry point. The application process (see Section 5.6) calls `Apply` with each committed operation. For the Fixed storage class only `CreateS`, `CreateO`, and `WriteO` modify the stable state and require application.

The semantics of the five operations the Fixed client interface provides are as

follows:

`FixedStatus CreateS(ObjID& oid, unsigned size);` This call creates a new segment and returns its unique 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 and makes an asynchronous call to the server requesting the segment. Because it has no impact on transactional correctness and makes no changes to the persistent state, `FetchS` does not need to be logged.

`FixedStatus CreateO(ObjID& oid, unsigned size);` This call creates a new object within a segment and returns its unique 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 in which 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.

The handle returned by this call is implemented as a virtual memory pointer that requires that the object be resident. If the object is not resident, its segment is synchronously fetched from the server. Once this operation is complete, the segment is pinned in virtual memory (see Chapter 6) 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 that let them modify the cached copies directly. However, these modifications do 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.

4.4 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 require new storage structures, such as multidimensional 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. In most cases, the performance of a standard OODB is considered 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 the requirements for adding a new storage class and presents an example of adding a non-traditional database service as a 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?" The designer can use the following 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 improve performance significantly. The BOSS storage-class facility was designed to let applications 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. A further 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 ADTs 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 expand 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. 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 and 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 that 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?

A storage class should serve as wide an audience as possible. 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, string matching, 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 notification — 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, though implementing the employee as a storage class is a less obvious strategy because this abstraction is usually included as an application-level object, not built into the database. However, since an objective of BOSS is to help capture and exploit the semantics of objects, we consider the employee as a storage class. The notification is a very desirable tool in modern database implementations. Triggers and cooperative transactions, neither of which is implemented within BOSS, could be implemented as a layer above the database using the notification as a fundamental building block.

Table 4.1: Evaluation of the criteria for three potential storage classes.

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

Table 4.1 shows the results of applying the criteria A-E to each ADT. As expected, the B-Tree turns out to be a highly attractive 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 does not fare as well as a storage class. Since it can be handled effectively in an existing record-oriented database. The only operation that could be optimized is iterating through all employees, and a B-tree could be used for this operation.

The notification 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 in addition a broad class of applications can take advantage of a notification. For these last two reasons, the notification would make an excellent storage class, and it used as an example in the next section.

4.5 The Notification Storage Class

This section describes the design of a BOSS storage-class implementation of notifications. The notification is significantly different from a conventional database object, and its inclusion in BOSS indicates the power of the storage-class abstraction. The notion of a notification was not considered in the BOSS design, and its later addition substantiates the claim that BOSS is capable of supporting the database requirements of new application domains.

The section begins by documenting the semantics of the notification object, then describes the correct concurrent semantics, and finally presents the detailed design of both the operations on the notification objects and the conflict operator for correct concurrent operation.

The BOSS notification has the following operators:

Create This operator creates a new notification object.

Wait This operator suspends the calling transaction.

Signal This operator signals the notification and wakes up all suspended transac-

tions.

The notifications 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 the intuitive notion of a notification's semantics.

Another problem is that a transaction can be awakened by an transaction that aborts, so that the transaction appears to have been 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 never to have run, so W appears to have been awakened by nothing at all.

Both the previous problems arise from a transaction is signaled and commits before the transaction that signaled it. We correct these problems by enforcing an ordering on the notification's operation: a transaction that signals a notification must commit before a transaction that is awakened by the signal. This constraint creates a well defined notion of time for the notification.

The constraint is implemented using timestamps. Wait keeps track of the timestamp of the notification object when it was called. Signal increments the timestamp when it commits. Wait cannot commit successfully if the timestamp is the same as when the notification object was first observed. The first column in Table 4.2 shows the timestamp of the notification object and all operations are tagged with the timestamp of the version that was observed.

Table 4.2 shows three possible interleavings of two transactions. The first two interleavings have a semantic error that causes an abort. The last interleaving does not have an error, so both transactions commit successfully. In the table "@57" is a timestamp. In the first column, the timestamp indicates the current time. In the other two columns a timestamp from the version of the object that was observed is coupled with each operation.

Table 4.2: Correct interleavings of transactions on notifications

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

The first interleaving has 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 interleaving has the problem that transaction W seems to wake 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 interleaving, transaction S notifies and commits before transaction W thereby incrementing the timestamp and allowing transaction W to commit.

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.

There are several important distinctions between the notification storage class (Note) and the Fixed storage class. One is that a client never requires access to the representation of a Note object. Another is in the size of a Note object: a Note object

consists of only the OID and timestamp that all BOSS objects contain. Buffering these small objects is not much of an issue, so the Note storage class has no explicit fetch operation. The operations on Note objects are:

`NoteStatus Create(ObjID& oid);` Create a new notification object 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.

`NoteStatus Signal(ObjID& oid);` Wake up all transactions waiting on this notification.

As the transaction examples illustrate, waking up another transaction before the notifier commits is likely to cause aborts, so the implementation of `Signal` 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.

`NoteStatus Wait(ObjID& oid);` Block until the notification is signaled.

`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.

The conflict operator has significantly different semantics, as presented in Table 4.3. The creator succeeds so long as the OID has not already been allocated and `Signal` succeeds so long as it observed the most recent state of the object.

This raises the issue of `Signal` failure. The previous examples do not show the failure of an invocation of `Signal`, because the only semantic constraint on the notification is that a transaction cannot be awakened before another transaction notifies

Operation	Timestamp = Current	Timestamp ≠ Current
Create	no conflict	no conflict
Signal	conflict	no conflict
Wait	no conflict	conflict

Table 4.3: Conflict table for operations on notification object

it. There is no semantic constraint on when Notify can be called. There is, however, a physical conflict, which 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 update operations 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 called a *blind operator* [46]. `Signal` 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 [46], modifiers cannot commute. For Note, this means that `Signal` can cause an abort. This constraint is not as bad as it might seem at first. First, any physical conflict that occurs in BOSS would occur in a system that offered only physical concurrency control; thus, BOSS is strictly better with respect to false conflicts than a physical system. Second, observers can commute over modifiers and modifications tend to be less frequent than observations, which implies that 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.

Chapter 5

Global Transaction Management

The purpose of global transaction management is to guarantee the necessary invariants the storage classes need to implement the concurrency-control criterion specific to their semantics. The maintenance of these invariants can impose significant costs above the basic costs of object motion and manipulation. BOSS introduces two new techniques that exploit optimism and asynchrony to reduce these costs. The cost of recovery after a system failure is reduced by using a new algorithm, no-redo write ahead logging (NR-WAL), and the cost of maintaining transaction atomicity is reduced by using a new cache coherency protocol, invalidate on abort.

No-redo write ahead logging is an improvement on an efficient write-ahead logging protocol that uses intention lists to eliminate undo processing during recovery. It exploits the fact that objects in the BOSS system need not be in a globally consistent state for correct operation. Instead, normal processing can begin quickly after a failure and objects are brought into a consistent state asynchronously in the background.

Invalidate on abort also exploits the fact that objects need not be in a globally consistent state. It is a relaxation of the common protocol of delivering invalidation messages during or immediately after an update. Invalidate on abort is one of many cache coherency protocols that can operate within BOSS because cache coherency is completely decoupled from transaction correctness. Cache coherency is an issue

critical to performance, and in BOSS the protocol yielding the best performance for each storage class can be chosen independently of its correctness properties.

Global transaction management maintains the following two invariants:

- Committed updates persist
- Only observations consistent with all committed updates commit

These invariants place no hard constraints on the states of copies of objects. This freedom allows copies of objects to drift from a globally consistent state and introduces the opportunity for optimizations. The freedom comes at the price of potential aborts due to observations of out-of-date copies, as investigated in Chapter 7. The present chapter describes these contributions and the technology BOSS uses for global transaction management.

5.1 Concurrency Control

In the BOSS computation model, clients perform operations on objects directly in their caches. These operations must be gathered into transactions and their effects atomically exposed to other clients and recorded in persistent storage.

Operations from different storage classes are collected into intention records by the client portion of the transaction manager. At commit time, the intention records are shipped to the server portion of the transaction manager. If there are no conflicts, the operations are atomically exposed to other clients by inserting them into the version table. The version table maintains an ordered list of every object's operations and a related structure, the transaction table, maintains an ordered list of every transaction's operations. These two lists form the grid structure shown in Figure 5.1.

The version table is a critical structure in an database using optimistic concurrency control. It is the one point at which the consistent committed state of every object can be determined and is used to ensure that the most recent copy of an object is sent to the client on a fetch request. It is used again to determine if an update invalidated

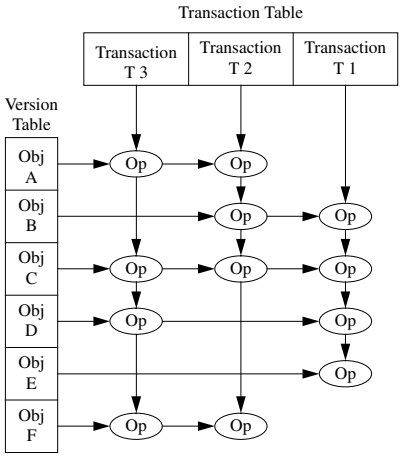


Figure 5.1: Operation records in the version table and transaction table.

an object observed by a transaction. The algorithms for these two processes are given in sections 5.8.1 and 5.8.2, respectively.

5.2 Cache Coherency

After a transaction commits, its effects must be propagated to other clients to maintain cache coherency. Although keeping the client caches as up to date as possible is good for performance, there are cases in which slavishly maintaining this constraint can actually slow the system. BOSS introduces a new protocol "invalidate on abort" that relaxes cache coherency to improve network utilization.

In many systems, cache coherency is required for correctness, and the synchronous communication required becomes a bottleneck [14]. BOSS informs client caches of updates asynchronously. This approach allows BOSS to construct larger messages containing more updates and to make more efficient use of the network than even the best of the synchronous call back locking protocols.

The flexibility of cache-coherency protocols has allowed us to experiment with new policies. Invalidate on abort, which takes a lazy approach to cache coherency, sends no invalidation messages when a transaction commits, but rather waits until a

transaction aborts and then sends all of the invalidation messages to all the clients. The rationale behind invalidate on abort is to let the client caches diverge until the divergence becomes a problem and then bring all the caches into a consistent state all at once. The performance characteristics of invalidate on abort are explored in Chapter 7.

5.3 Persistence

As part of the commit process, the effects of the operations are made persistent. BOSS relies on the service of a persistence layer to achieve stability for data and transactions. This layer consists of two fundamental components: a write-ahead log and one or more disk partitions that contain the base versions of the data (see Chapter 6). Modifications made by transactions are made durable by entering an intentions record in the log. The coordination of the log and the heap of base versions provides full persistence. BOSS does not require the existence of a current copy of an object.

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

Figure 5.2 shows the history of a series of update operations on an object. The update operations are stored in a list, with the most recent operation at its head. The base version contains a pointer to the last operation applied. In normal operation, the base version can fall behind the current state, and the object is brought up to date by applying the appropriate operations from the history. In Figure 5.2, the base version is two operations out of date and applying Op4, then Op5 brings the object up to date.

Applying the appropriate operation to a copy of an object brings that copy up

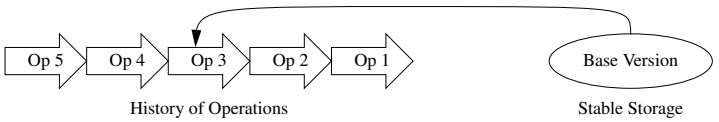


Figure 5.2: Components of the current state of an object.

to date, but it modifies not the base version, but only the copy in volatile memory. The base version is modified asynchronously sometime before the buffer manager at the server needs more space. The buffer manager tracks which objects have been modified and sends them to stable storage before it discards a volatile copy.

5.4 Recovery

BOSS introduces a recovery algorithm called no-redo write-ahead logging (NR-WAL) as a means of reducing recovery overhead. Intentions lists are used to eliminate the undo phase of recovery [3]. There is no redo phase; NR-WAL does not require the current state of an object to be stored stably, so the stable state need not be recovered immediately after a crash: the most recent state before the crash can be reconstructed later, on demand, from the last stored version and the intentions list.

Both the history of operations and the base version of an object are stored stably and are not affected by a system failure. Recovery is instantaneous if a small amount of fast non-volatile memory is available to maintain the version and transaction tables. Otherwise, periodic checkpoints of these data structures reduce recovery time.

Figure 5.3 shows the portion of the log that is scanned during recovery by NR-WAL and WAL. "Crash" indicates the position in the log that was being written when the failure occurred. The recovery pass begins at the last complete checkpoint and replays only updates to two critical data structures: the version table and the transaction table. The stable tail of the log is kept ahead of the actual tail of the log at a well known location. Normal operation begins upon completion of the scan. A volatile copy of an object that was lost is recreated on demand from the object's base

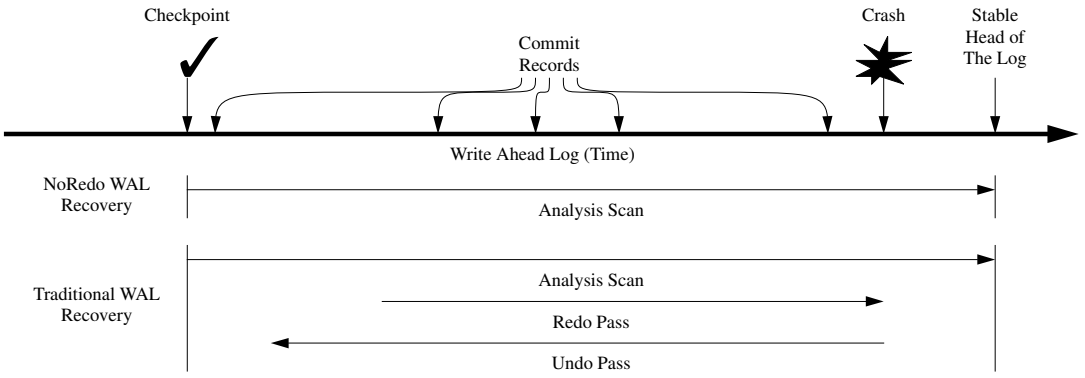


Figure 5.3: Asynchronous versus synchronous recovery.

version and history.

The asynchronous protocol used in BOSS requires neither a redo phase or an undo phase to synchronize the database with any wall clock or internal system time. After a fast analysis scan, the system is ready to begin processing. Any updates lost from the buffers are redone later in the background or as needed.

The NR-WAL algorithm is fundamentally the same as write-ahead logging using intentions lists with one added feature. BOSS has the ability to extract update operations from the log during normal processing. If an object is out of date with respect to committed state, the pending operations are located using the version table and the object's timestamp. The operations are then applied to the object, bringing the object up to date. This ability allows BOSS to postpone the redo phase of recovery until normal processing begins, where it is handled asynchronously, in the background.

The correctness of this extension relies on ensuring that all updates are applied to an object before it is used in a transaction that commits. This property is ensured by the validation phase of BOSS's optimistic concurrency-control protocol. If the object has pending operations, the object's timestamp is behind the timestamp maintained by the version table. In this case, during validation the conflict operation detects the discrepancy and returns true, which causes the transaction to abort.

The previous paragraph argues the correctness of NR-WAL; however, correctness does not guarantee that any progress is made after a failure. An object with pending

operations that are never applied would cause every transaction in which it participated to abort. There are two separate means by which pending operations are applied to ensure progress after a failure. First, each time an object is moved from the server to the client, the object's timestamp is checked against the version table and the object is brought up to date if necessary (See Section 5.8.1). Second, a background process at the server scans the log for pending updates and applies them to objects (see Section 5.6). As a result, pending operations are applied to objects, allowing transactions to commit and guaranteeing that progress is made.

5.5 Data Structures

Concurrency control, persistence, and recovery are implemented at the BOSS server using a write-ahead log. The transaction table and version table are two views onto this log. In addition to providing different abstract views, both of these structures accelerate access to particular parts of the log. This capability allows the BOSS log to be used for both recovery and online processing.

5.5.1 The Log

BOSS's novel implementation of the log allows quick recovery quickly after failure. BOSS implements a circular log on top of a memory-mapped partition (see Figure 5.4). Mapping the log allows fast access to log records through virtual memory.

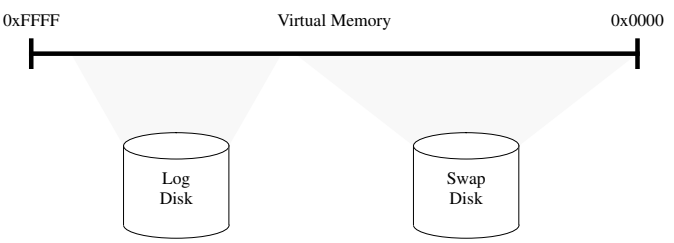


Figure 5.4: Mapping the log into virtual memory.

A log sequence number (LSN) is a reference to a record in the log. The implemen-

tation of LSNs takes advantage of direct mapping to make possible fast location of log records. An LSN is a tuple in which the low order bits encode a memory address in the log. Interpreted as an integer, this number strictly increases as the log is written until the log wraps around. Our concurrency-control algorithm [20] requires a timestamp that increases strictly over all time, not in a piecemeal fashion. To correct this problem, the memory address is augmented with a count of the number of times the log has wrapped around. This count results in an LSN that consists of a 32-bit virtual memory pointer and a 32-bit integer as a wrap count, for 64 bits total.

5.5.2 The Version Table

The version table which implements an object's history (see Figure 5.2) is a main memory index that maps from OID to the LSN of the object's last update operation. Each update operation maintains the LSN of the previous update operation. These links form the list of operations that is an object's history. The version table also maintains several status bits to maintain the state of the object for two-phase commit (see Section 5.8.2).

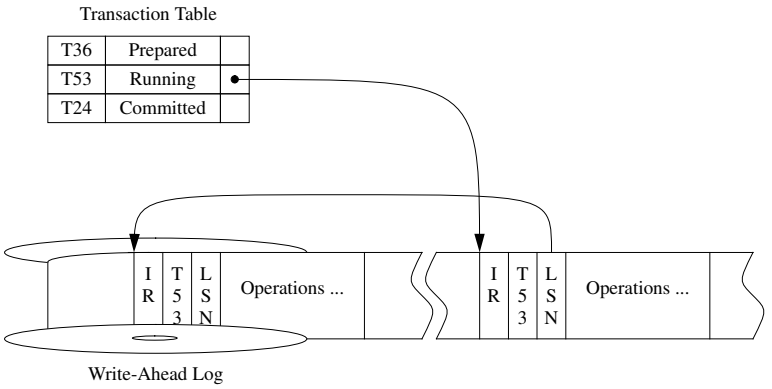
5.5.3 The Transaction Table

The purpose of the transaction table is to maintain the grouping operations into transactions for commit and recovery processing. This task is complicated by the fact that BOSS supports multiple storage classes and each storage class can define its own operations. BOSS defines a general operation record that storage classes extend by subtyping to suit their own needs.

Operation records are the basic building blocks of the transaction table, but an intermediate structure is needed: an intention record groups all the operation records for an individual transaction. The transaction table provides fast access to the intention records through a main memory index.

The transaction table is implemented in much the same manner as the version

table. It is a main memory index that maps from the transaction identifier (TID) to the LSN of the transaction's last log record. It also maintains status flags for each transaction that indicate whether it is running, prepared, committed, or aborted. Figure 5.5 illustrates a small transaction table and two intention records and also contains definitions of their structure.




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

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

Figure 5.5: Chaining intention records.

The first column of the transaction table is the TID. The table is keyed on this field. The second column indicates the status of the transaction. This field is modified by commit processing as described in Section 5.8.2. The third column is a reference to a log record implemented as an LSN.

The intention records in the diagram are chained. Chaining allows clients to stream the records down to the server in advance of a commit, thereby reducing communications overhead during the commit. The contents of the intention records are operation records.

5.5.4 Operation Records

The purpose of an operation record (see Figure 5.6) is to store a transition persistently from one state of an object to the next. It also is used during the commit process to determine if there are any conflicts with other operations. An operation record also maintains a link to the last previously committed operation record. These links implement an object's history.

All operation records store the OID of the object and the LSN of the copy that was read. The LSN acts both as a timestamp for concurrency-control validation and a link back to the previously committed operation. In addition to these two fields, an operation record has a status flag that indicates the general nature of the operation: creator, observer, or modifier.

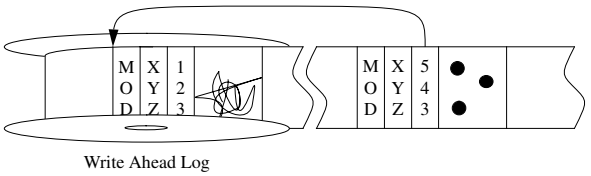


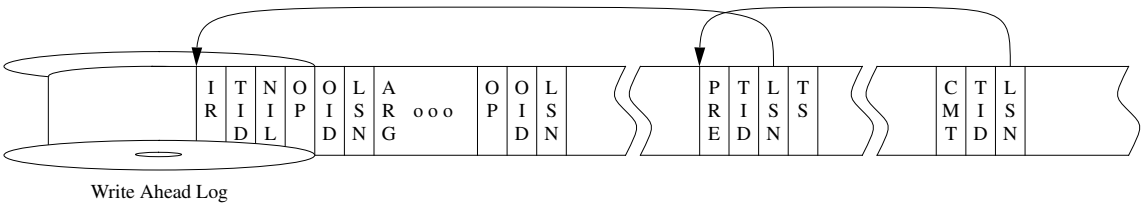
Figure 5.6: Operations in the log.

The remainder of the operation record is storage-class-specific. In the case of a modifier, it contains the information necessary to transition the object from the previous state to the next state. The data stored at this location could simply be the entire new value of the object; a more compact representation might be a delta or the arguments to some function from the old state to the new state. (In Figure 5.6 the dots and squiggle are a rendition of such information.)

5.5.5 Prepare and Commit Records

The two remaining types of records in the log, prepare and commit, are used in the standard manner by the two phase commit algorithm described in Section 5.8.2. Figure 5.7 illustrates their use in a simple but complete transaction. The leftmost

record is an intention record with two operation records, the middle record is the prepare and the rightmost is the commit record.




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

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

Figure 5.7: A complete simple transaction in the log.

5.6 Application Process[1]

Several issues have yet to be resolved with the design as presented thus far.

- The number of updates that must be applied to an object before it is sent to a client could grow large and cause delays.
- The version table could become so large that it infringes on the buffer pool and needs to be paged.
- The circular log could fill up, causing transactions to be aborted.

These problems are all artifacts of letting the database fall too far out of date:

- An up-to-date object requires no applications on a read and therefore causes no delays.
- The version table does not need to keep an entry for an object whose stable state is up to date with its committed state.

[1]The term "application process" refers to the server process that applies updates to the base versions of objects. It is not to be confused with the process under which the client application runs.

- Operation records that have already been applied to an up-to-date object consume log space that can be reclaimed.

The application process corrects these problems by applying operations in the log to stable storage. It chooses which operations to apply based on an application pointer: an LSN that satisfies the invariant that any operation with a lower LSN is reflected in stable storage. The application process scans forward from the application pointer looking for the oldest operation on an object with an entry in the version table. If the version table contains an entry, the operation may not have been applied to stable storage. The application process then checks the LSN of the object and brings it up to date if necessary. When the object has been returned to stable storage, the application pointer moves forward.

Applying the operation reduces the amount of work that needs to be done on a read and frees that slot in the version table, thus correcting two of the problems mentioned earlier. In addition, the application pointer is an upper bound on the head of the log, so moving the application pointer forward effectively frees space in the log.

5.7 Examples

This sections contains two related examples of update processing in BOSS. The first follows the course of a modification from the client to the server: the second follows the update into the database partition and other clients caches.

5.7.1 Update

Figure 5.8 shows a sample system with a single class and two clients. It follows the evolution of an object X through the stages of a transaction.

Stage 1: The state of X is consistent throughout the system: all copies of X are of version 123.

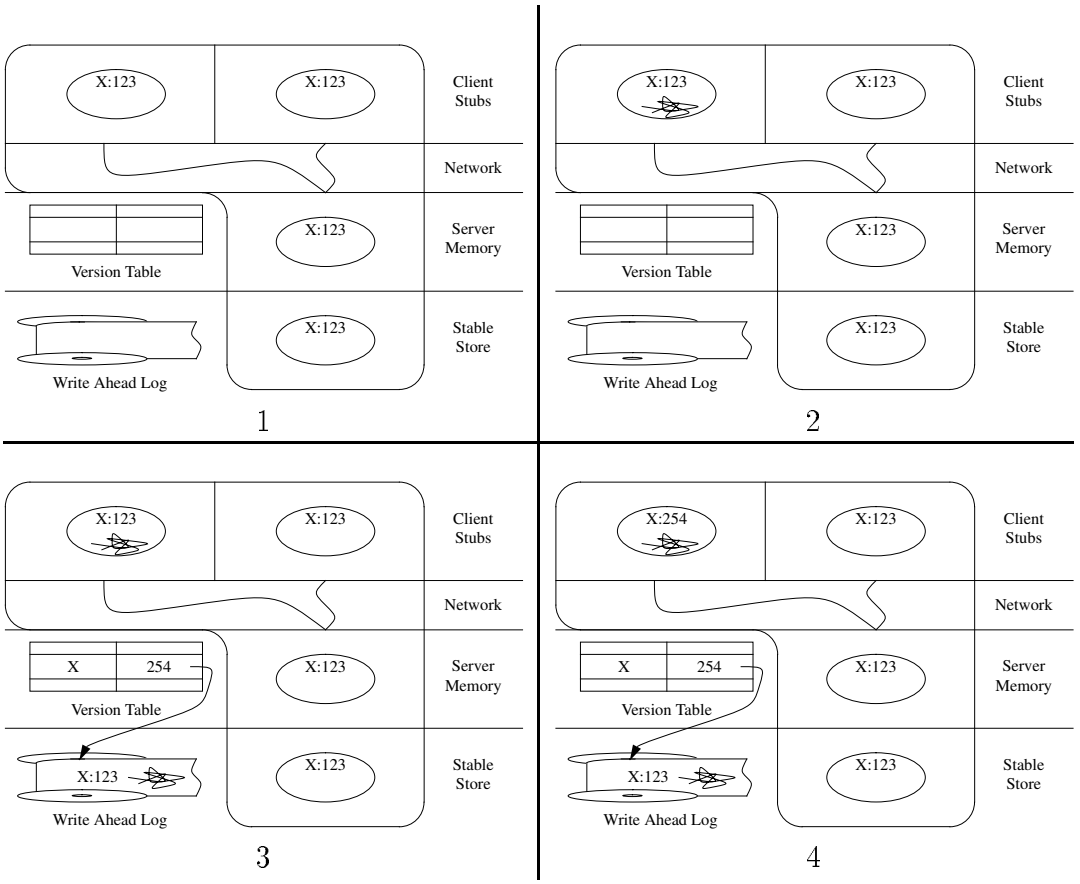


Figure 5.8: A sample update

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

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

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

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

5.7.2 Background Processing Example

Figure 5.9 continues the example of an object in the server. Section 5.7.1 left off when the client that modified X had been informed of X's new LSN, 254. This example follows the new copy of X to the other clients, into the stable store and out of the log.

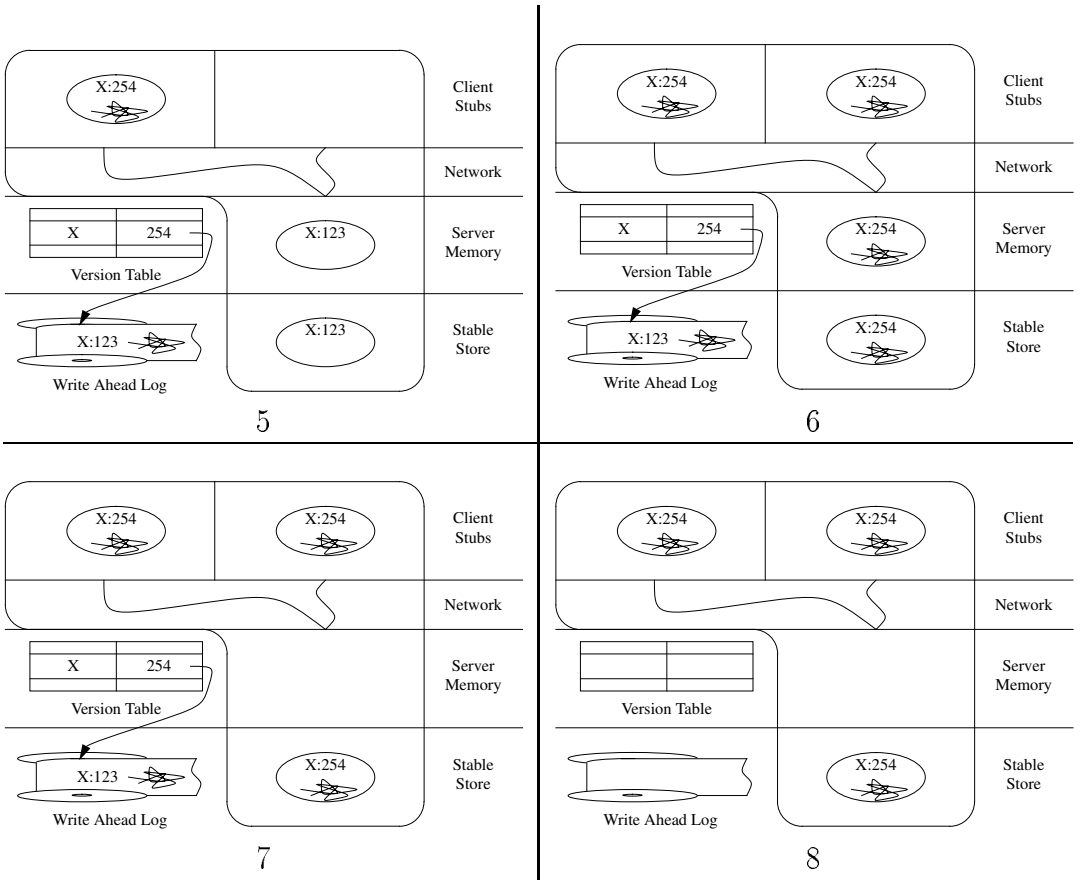


Figure 5.9: Background processes brining system to a coherent state.

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

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

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

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

5.8 Algorithms

BOSS has a unique protocol for update processing. The section contains examples; Section 5.8.1 contains the core of the algorithm for lazy updates and Section 5.8.2 presents the integration between the standard two-phase commit algorithm and BOSS data structures.

5.8.1 Lazy Updates

An operation is not applied at the server immediately after it is committed, since BOSS allows an object to remain out of date. However, to avoid aborts, it is important that clients fetch up-to-date copies of objects. BOSS brings the copy of the object in the server cache up to date as part of the fetch operation. The pseudocode is:

Fetch(OID)

Locate(OID)

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

Apply(Version_Table(OID).LSN, Copy)

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

MoveUp(OID)

The up-to-date copy is moved up to the client's buffer space.

End Fetch

Apply(Operation, Copy)

if Lsn $\neq$ Copy.Lsn

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

then Apply(Operation.Lsn, Copy)

Otherwise, at least one operation needs to be applied to the object. Apply recurses to determine if other operations need to be applied. Each modification in the log contains the LSN of the previous operation. The arguments to the recursive call are the copy and the LSN of the previous operation.

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

endif

End Apply

5.8.2 Prepare and Commit

This section describes what happens in the prepare and commit phases at the server. The general flow of information during a transaction is presented in Section 4.2, and we use the two-phase commit protocol defined in Grey [17].

BOSS executes the following pseudocode during the prepare phase. This implementation is not reentrant, so only one transaction can prepare at a time.

Prepare

if Conflict

In the prepare phase, the server transaction manager uses the Conflict operation defined by the storage class (see Section 5.3) to determine if any operation from the transaction's intentions records is invalidated by a transaction that prepared earlier.

then Abort Transaction

The transaction aborts and its intention records are ignored. (See Abort.)

else Update tables and force a prepare record

The transaction manager forces a prepare record to the log, updates the transaction table, and updates the version table. All three of these actions occur as an atomic operation with respect to other prepares. The current implementation allows many transactions to commit concurrently within a server. However, only one transaction prepares at a time, so atomicity is achieved by default. The transaction table needs to reflect the current state of the transaction and hold the LSN of the prepare record. The version table needs to contain the LSN of the most recent prepared operations for

each object. If the transaction is distributed over multiple servers, the version table also notes that the modification has only prepared and not yet committed. The prepare record contains the LSN of the transaction's last intention record, the transaction identifier, and a timestamp used by the two-phase commit process (see Figure 5.7).

endif

End Prepare

If the prepare phase completes successfully, the coordinator initiates the commit phase. Each participant executes the following pseudocode.

Commit

Committing a transaction requires modification of the tables. At commit, the transaction manager modifies the version table to show that the operations are now committed. The transaction table notes that the transaction has committed and stores the LSN of the commit record. The commit record contains only the transaction identifier and the LSN of the prepare record.

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

End Commit

If the prepare phase does not complete successfully, the participants are notified that the transaction must be aborted.

Abort

If a transaction aborts in the second phase of two-phase commit, the version table reverts to its previous state and the transaction table must be updated.

An abort record is forced to the log.

The client is notified of the abort and must flush all objects that the transaction modified. In addition, the client is notified of copies that are no longer up-to-date because of updates by other transactions. The client then removes those copies, thereby avoiding future aborts.

End Abort

At the instant a transaction prepares, the state of the database must be consistent with the one which the transaction observed, because transactions are serialized in the order in which their prepare records enter the log. This is not to imply that the entire database needs to be in a consistent state — only the small portion that the transaction observed. Further, the database is not in a consistent state immediately after the transaction prepares and commits.

When the server receives an intention record, the server assigns it an LSN and places it in buffers that are eventually forced to the log. Intentions are asynchronous and do not force the log. The buffers that hold the intention records are not pinned, so they can be written to disk if more buffer space is needed.

The client transaction manager maintains the invariant that all intention records are sent to the server before the client sends a commit record. Our transport protocol guarantees in-order delivery. Therefore, by the time the server transaction manager begins to prepare a transaction, the transaction's intentions are at the server.

Chapter 6

Memory Object Management

Extensibility in BOSS is more than just the ability to add a new storage class. BOSS supports the storage class developer with modules for naming, object motion, persistence, and other needs common to most storage classes. This chapter explores memory-management support for new storage classes.

Memory management in the context of an object store is a critical issue. At a higher level, the object store itself can be considered a memory manager. The pivotal question in the design of a memory-management service for an extensible object store is: how much functionality can be provided without restricting the implementation of storage classes? Memory object management (MOM) makes the addition of a storage class easier, since MOM hides much of the complexity of dealing with the operating system and mediates among conflicting demands of storage classes.

MOM is a tool for building a caching policy. It contains a default policy, but storage classes may exploit their semantics by building their own policies. Objects may be fetched or discarded, before or after the default policy would relocate them. For example, a storage class could implement intelligent prefetching by learning access behavior. If a familiar pattern is recognized, MOM would be instructed to move objects into the cache before they were explicitly requested by the client, so as to yield better response time.

This chapter begins with a description of the MOM architecture and then in Section 6.2 presents the memory object abstraction. Section 6.3 describes how MOM handles multiple memory management policies, and the interfaces to MOM and memory objects are presented in Section 6.4. Finally, Section 6.5 demonstrates how MOM can be used implement an efficient memory management policy for a B-tree.

6.1 MOM Architecture

We considered many architectures for this service and experimented with several. The one we settled on places one MOM on each BOSS host: client or server. This architecture provides the storage-class developer with a consistent interface, reuses the MOM source code, and gives BOSS intelligent client-disk caching.

Figure 6.1 shows a macro view of a storage class distributed across the system layers. The storage-class client communicates with the server via a special RPC protocol suited to multithreaded, asynchronous operation. Through this connection, the client receives objects and control messages from the server and immediately places the objects it receives under MOM control.

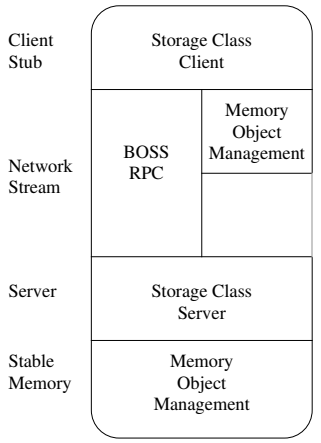


Figure 6.1: A layered view of a storage class.

The MOM at the server is a different instance of exactly the same abstraction at the client linked into a different executable image. MOM use differs minimally at the

server. The level of multithreading at the server is very high; one thread per pending request runs through MOM to fetch objects for clients and to update objects after transactions, whereas at the client, usually only one or two threads run through MOM. Another difference is that at the client, MOM performs replacement on nonvolatile objects while nonvolatile objects at the server are never replaced.

6.2 The Memory Object

The basic unit of allocation and transfer in MOM is the memory object (MO), which is a contiguous, variable-sized, persistent chunk of memory. Several factors influenced the choice of whether to make MOs page-sized or page-aligned versus truly variable-sized. Many of the objects that storage classes implement are not page-aligned. Thus, storage classes would have to implement variable-sized objects on top of large fixed-sized objects, which is tantamount to implementing a simple memory manager. Another factor is that variable-sized objects not require the additional runtime overhead over fixed-sized pages.

The MO is implemented as a portion of an extent, which is a contiguous piece of secondary storage implemented as either a container file or raw partition. A MO's location and status information are kept separate from the MO body itself. When the storage class requires access to the MO, it pins the MO to virtual memory as illustrated in Figure 6.2. Pinning a MO returns a handle to the body of the MO that is guaranteed to be valid until the MO is unpinned. The MO need not be read into VM at this time; rather the VM system loads the object on demand.

6.3 Memory-Management Policies

The main reason for building a storage class on MOM rather than the underlying OS is that the storage class often can manage memory more efficiently than the OS

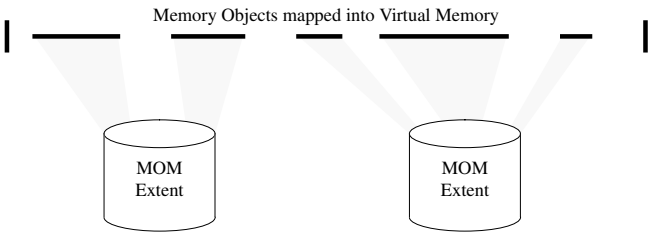


Figure 6.2: MOM object mapping.

could. The storage class can track the access history to predict future behavior, and can also exploit invariants in access behavior, such as the invariant that a tree root is always accessed before a tree leaf. Section 6.5 describes a policy that exploits this invariant.

MOM provides a general-purpose memory-management policy that storage classes can specialize to suit their needs. The least-recently-used (LRU) replacement policy, with fetching on demand, is the most common general-purpose memory-management-policy [25]. MOM uses a variant of LRU as its general-purpose policy and provides the storage class with high-level control to change the behavior of the default policy.

If contention occurs, a default global replacement policy based on a two-handed clock algorithm [25] arbitrates, as illustrated in Figure 6.3. A cleaning hand moves through the cached objects marking the objects unread; this hand is kept some fixed distance ahead of the allocation hand. Given a request for a new space, the allocation hand moves through the cached objects, discarding those that have not been read since the cleaning hand passed them. It stops as soon as enough space has been reclaimed to satisfy the request.

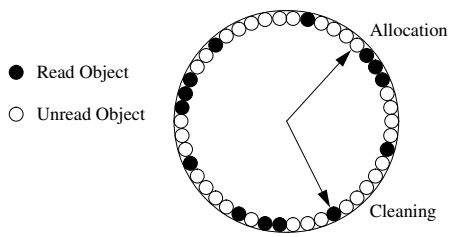


Figure 6.3: A two handed clock for object replacement.

There are two important differences between BOSS's memory management policy and BSD UNIX virtual memory. First, classes must be able to take advantage of the semantics of their objects to increase performance. Second, BOSS is a distributed client-server system. At the client, the non-volatile store is also just a cache and must be managed as such.

A caching policy consists of two subpolicies that need not be independent. The fetch policy determines which objects to bring into the cache. UNIX uses a demand-fetch protocol in which pages are not moved into memory until they are needed. In BOSS, storage classes have complete control over which objects are cached in the first place. The replacement policy determines which objects to discard when more space is required. UNIX uses the clock-based algorithm. The MOM replacement policy is driven by the clock and defaults to the clock if no free space is available, but several hooks give storage classes significant influence over which objects are kept and which discarded.

Local disks are used for caching on client machines, and this too must be managed. Managing a disk cache is somewhat different from managing a virtual-memory cache. One major difference is that it is expensive to look at the object when making decisions. There is a location operation to determine whether or not an object is resident before accessing it.

MOM memory objects are actually memory-mapped portions of a container file or raw partition [41]. Memory mapping has numerous advantages. Using memory mapping completely avoids the problem of double buffering (writing an object to a second stable storage location for temporary storage). The operating system keeps track of modifications to memory objects. When an MO is pinned it is assigned a virtual-memory address. The MO need not be immediately read into memory; the operating system's virtual memory subsystem delays the read until the object is actually used.

At the client, a memory object passes through four states with regard to the

replacement algorithm:

vread This state indicates that the object is mapped into virtual memory and has been read since the last pass of the cleaning hand. This is the state in which memory objects are created and into which memory objects are put after they are read.

vunread This state indicates that the object is mapped into virtual memory, but has not been read since the last pass of the cleaning hand. The cleaning hand marks vread objects as vunread. If an object remains in this state (is not read) until the allocation hand comes around, it is unmapped and moved down to disk.

sread This state indicates that the object is not mapped into virtual memory, but has been mapped since the last pass of the cleaning hand. This is the state that objects enter after they have been discarded from virtual memory. The default replacement protocol considers an object that was just discarded from virtual memory to be the most desirable object in stable memory.

sunread This state indicates that the object is not mapped into memory and has not been read since the last pass of the cleaning hand. The cleaning hand marks an sread object as sunread. If an object remains in this state until the allocation hand comes around, it is discarded from the client entirely.

Figure 6.4 shows the transitions between these states. An MO is created in the vread state. When the cleaning hand comes around it marks the MO vunread. Any vunread MO is a candidate for replacement. If the space it occupies is needed, the MO is swapped out to secondary storage. If a vunread MO is pinned, it returns to the vread state and cannot be swapped out. A MO that has just been swapped out is marked sread. Essentially the MO has just finished being read by the layer above. The cleaning hand marks sread MOs as sunread. If the space an sunread MO occupies

is needed, it is discarded entirely from the client cache, so that if required in the future it must be fetched from the server. Again, if the object is pinned it returns to the vread state.

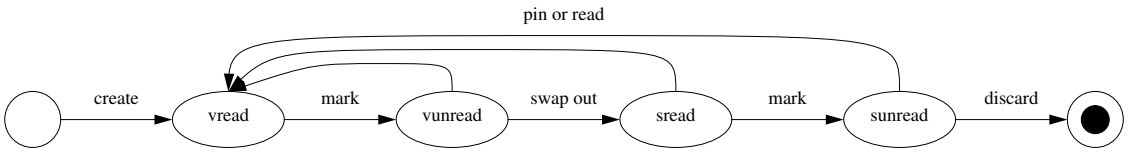


Figure 6.4: The states and transitions of a memory object

In the preceding description the cleaning hand actually serves two purposes: marking objects in virtual memory and marking objects in stable memory. Virtual memory and stable memory differ greatly in size and access behavior, which causes problems with one clock managing all objects. For this reason, there are actually two two-handed clocks at the client: one clock for virtual memory and one clock for stable memory. These clocks are free to move at different speeds through their objects and to maintain different distances between their cleaning hand and allocation hand. By adjusting these parameters, MOM can adapt to diverse access behaviors.

6.4 Application Interface

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, as discussed later. Most interactions are with memory objects themselves (see Figure 6.5).

The MO interface is straightforward. It provides seven methods. In the syntax below the MO is implicit in the method invocation.

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

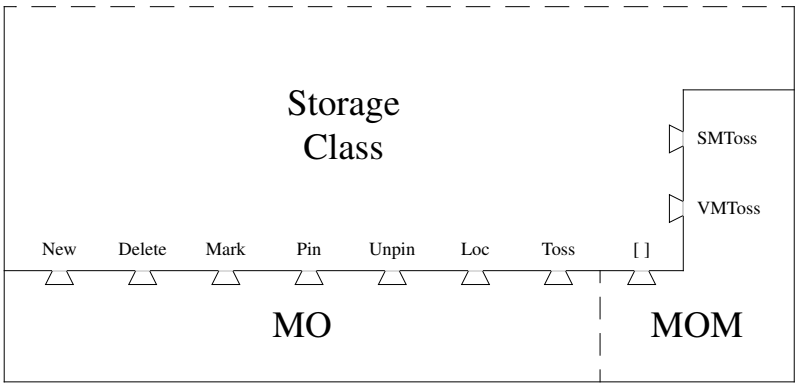


Figure 6.5: The storage class interface to the MOM module.

`delete *Mo` Delete frees the nonvolatile memory and any virtual memory associated with the MO.

`int mark()` 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 returns its address without pinning the object or allocating virtual memory. Storage classes use this call to determine whether 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 thus should be discarded.

There are no interface methods to mark a MO as dirty or to write its contents out to disk. MOM handles these tasks, notifying the storage class when it takes an action via two callbacks:

`int VMToss(Mo *mo)` MOM notifies the storage class that it intends to discard an object via this call. The storage class can refuse the request, but it should do so only if the MO will be used in the immediate future, since consistently refusing to discard MOs quickly causes the entire system to livelock.

`int SMToss(Mo *mo)` SMToss serves the same function as VMToss but in the domain of nonvolatile memory. This call is made only at the client, where nonvolatile memory is used as a cache. The server nonvolatile memory is used only for persistent MOs, so if it fills up, New simply fails.

MOM provides a mapping from OID to the MO that contains the object. The function is simple but has implications for the sorts of storage classes that can be constructed: it does not restrict the interface of the storage class, but it does have performance implications.

A performance-critical operation in an OODB is the mapping of logical OIDs to physical data locations, and this mapping can occur many ways [33, 9]. Previous MOM designs did not include this mapping, on the rationale that a storage class should implement whatever logical-to-physical mapping best suits its semantics. For instance, a storage class that implements pages might compute the physical location from the logical identifier and save a table lookup. However, this method is not of much use when the storage class cannot choose the MO's physical location. A storage class that requires this sort of mapping could be built without using MOM, but at the cost of a greater implementation effort.

The design decision for this iteration was that, because MOM determined the physical location of MOs, it should also perform the logical-to-physical mapping. This decision does not significantly complicate the implementation of MOM and greatly simplifies the implementation of the existing storage classes. A future design may allow specialization of the association operator to let a storage class choose its implementation; however, static binding in C++ makes this sort of specialization difficult to accomplish.

6.5 Example B-Tree Memory Management

Storage classes are expected to be good neighbors and not aggressively acquire as many resources as possible, since doing so causes 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, such as memory corruption, 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, which 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 downward, so there is no point to keeping a child of a node that is being discarded. The VMToss and SMToss calls are augmented to discard as well children of nodes being discarded. Whenever a node is discarded, the storage class recursively descends the tree, using the loc operation, and discards all resident child nodes.

Chapter 7

Cache Coherency Experiments

This chapter explores the interaction between client-disk caching and asynchronous invalidation protocols. The experiment records the time to commit and the frequency of aborts for several sizes of client disk cache and under several workloads. We expect that increasing the cache size would lighten the load on the server and thus decrease time to commit; however, expanding the cache increases the probability of observing an out-of-date object and causing an abort. We find that an appropriate invalidation protocol allows clients to benefit from large disk caches without a high rate of aborts. This result indicates that optimistic concurrency control is a viable substrate for an OODB.

The chapter begins with a description of the workloads we expect BOSS to encounter. Following this, Section 7.2 details the experiment's hardware environment, Section 7.3 defines the benchmark, and Section 7.4 contains the results of our initial experiments. Section 7.5 introduces the new protocol that we invented as a result of these experiments, and finally, Section 7.6 explores the influence of longer transactions on the different invalidation policies.

7.1 Assumptions About Contention

One characteristic of optimistic protocols is increased aborts. This feature is of particular concern in a design environment because engineers are not particularly tolerant of lost work. Concurrency-control semantics for engineers are studied in [34]. A serializable or atomic transaction is meant to be a single indivisible unit of work, such as a bank deposit or a flight reservation, and thus the work done by the transaction must be isolated from other concurrent transactions. A design activity, such as writing a book or designing a new airframe, on the other hand, is composed of activities that can last for hours, days, weeks, or even months. During this time, various users cooperate, building on each other's work, and full isolation is not desirable in this context because designers need access to the intermediate results of their coworkers.

The BOSS architecture assumes that a layer outside of the system maps the semantics of design transactions down to a series of short atomic transactions [34, 24]. This layer coordinates the activities of the designers to provide the desired semantics, but it has several other effects as well. By coordinating activities, the layer acts as a scheduler on top of the database, which has the effect of reducing contention in the workload as perceived by the database. For example, a high-level constraint requiring that analysis complete before design begins forbids contention between the analysis and the designer. In a design transaction, events happen slowly, and even on a large system there are rarely more than 100 users working. By reducing the level of contention, the layer further reduces the number of aborts initiated by the transaction manager. Another effect of the layer is to automate resubmission of transactions, thereby hiding aborts from end users. Thus, all the workloads in our tests have low levels of contention.

7.2 System Configuration

Previous chapters described the BOSS architecture in general; this section describes the implementation used in the experiments. The experiments use one server equipped with a 60-MIPS CPU and 32M of RAM, of which 1M is used for a server database cache. The server has three file systems, one each for virtual memory, the database partition, and the log.

Three client machines also have a 60-MIPS CPU and 32M RAM, but the client machines run two instances of BOSS, for a total of six BOSS clients. The reason for this choice is that in an actual system, BOSS will not be the only process running on a client. Placing two BOSS clients on each machine ensures that a single BOSS client takes no more than half the resources.

The cache size of the clients is a variable in the experiment. The clients have a 2-, 4- or 6-Mbyte local disk cache. Up to 1 Mbyte of this cache may be mapped into virtual memory at any one time. The BOSS client disk cache shares the local disk with the virtual memory paging system, but is maintained in a separate partition.

7.3 Client Disk Caching: Experiment #1

This benchmark was designed to be a worst-case stress test on the database. The database and workloads are modeled after those used in simulations conducted by Franklin [14], through because Franklin's simulations make different hardware assumptions the results are not directly comparable. The database consists of 40,000 objects, each 128 bytes long, clustered into 1,250 segments of 32 objects each. The resulting database is 6.5M including overhead.

The workload is divided into hot and cold regions. Each client has its own hot region, which contains 1,600, objects or 50 segments, and receives 80% of the accesses, four times those of the cold region. For every access to an object in the hot region, there is a 20 percent chance that the object will be updated.

The characteristics of the cold regions, which are shared by all clients, are varied to produce three different workloads. In the first workload, the cold region does not overlap the hot regions and the cold region is never updated. This workload models a case in which each designer in a design group has a private workspace and draws from a static library. Because there is no possibility of an update conflicting with a read in this workload, there is no contention. The no-contention case in Figure 7.1 shows this workload. The hot regions are separate and dark. The dark shading indicates that the region is both read from and written to. The cold region is light and because it does not overlap the hot region, gaps occur. The light shading indicates that the cold region is only read and not written. No shading indicates that the region is neither read nor written.

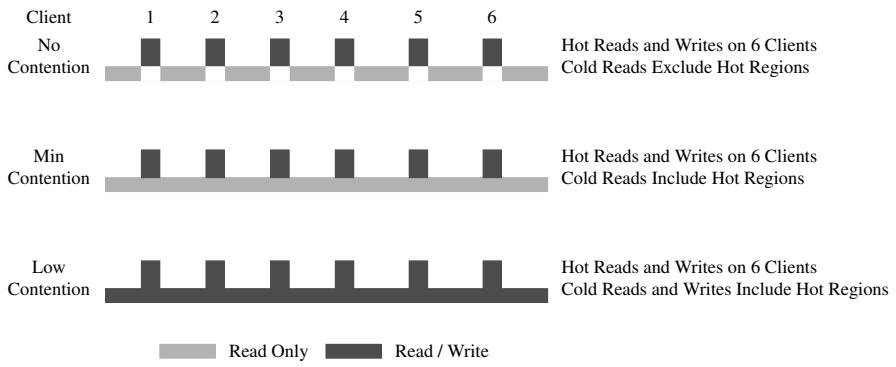


Figure 7.1: Three benchmark workloads.

In the second and third workloads in Figure 7.1, the cold region overlaps the hot regions. In the second workload, min contention, the cold region is read only. This workload models the case in which the designers in a group occasionally read from one another's workspaces as well as from the static library. This workload introduces a small amount of contention because because other clients' hot regions are read as part of the cold region. The third workload, low contention, allows writes to the cold region. The probability that an write will occur given that a read has occurred is 20 percent. This workload models the case in which the designers occasionally update the library and possibly other designers' workspaces, a workload that introduces a

higher level of conflict.

Each client submits 1000 transactions, for a total of 6000 transactions to be completed. Each transaction performs 20 operations dictated by the workload. A new transaction is submitted as soon as the previous one completes, and there is no think time during a transaction and no wait time between transactions. These conditions push the server as hard as possible: it is designed to use idle time to catch up, but no idle time is available in this sort of worst-case workload. The times measured in the tests are for the successful completion of a set number of transactions. Aborted transactions are handled by immediately resubmiting the same transaction.

7.4 Results of Experiment #1

The first experiment explores the utility of client disk caching and the effect of invalidation protocols on time to commit and the number of aborts. Each graph illustrates three cache sizes under different workloads and invalidation policies.

7.4.1 No Invalidations and No Contention

Figure 7.2 shows the performance of the system running under the ideal condition of no contention. The Y axis on the graph is the time to process and commit the number of transactions on the X axis. Under this condition, the system is I/O-bound. The graph clearly shows the advantages of client-disk caching: time to commit is reduced by more than half from the 2M cache test to the 6M cache test.

7.4.2 No Invalidations and Minimal Contention

It is not surprising that performance improves as the cache size increases when there is no contention. The primary cost of caching is maintaining consistency between the caches, which is not an issue without contention. This next experiment introduces some contention. The result of the contention is an increased number of aborts.

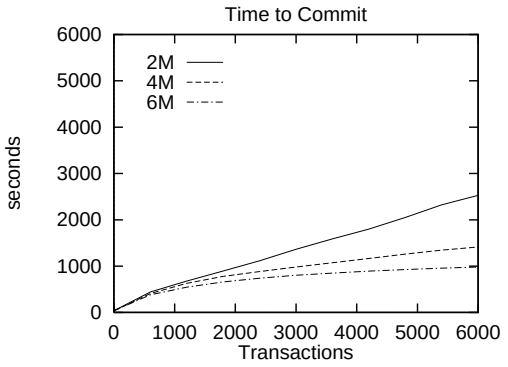


Figure 7.2: No invalidations with no contention.

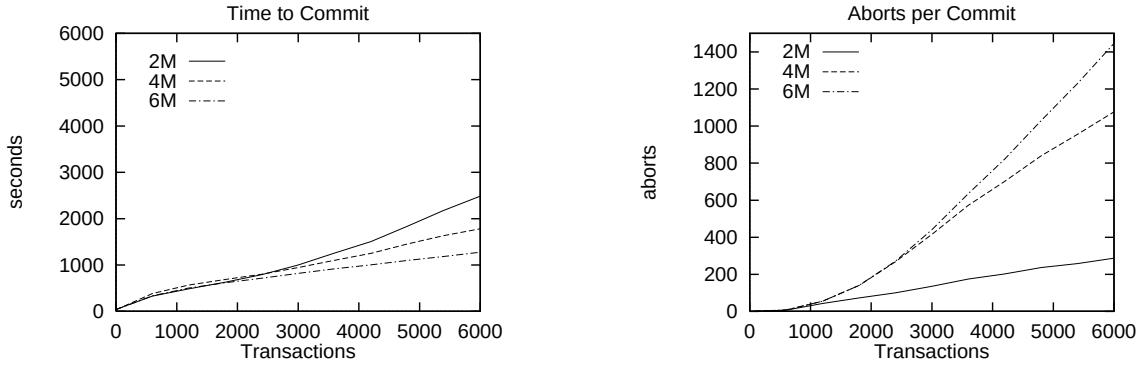


Figure 7.3: No invalidations with minimal contention.

The number of aborts increases from zero in the no-contention case (not shown) to over 1400 for the 6M cache in the course of completing 6000 successful transactions. The reason for the very high number of aborts is that, without an invalidation mechanism, whenever a cached object becomes out of date it triggers an abort.

The impact on performance is not severe because the cost of retrying a transaction is very low in our workload model. After an abort, the offending object is discarded and the aborted transaction is resubmitted (See Chapter 5). Except for discarded objects, all the objects used by the transaction are cached locally, so only discarded objects need to be fetched. Once this occurs, transactions may be committed. Section 7.6 describes an experiment with a higher cost for restarting a transaction.

7.4.3 Invalidate on Commit and Minimal Contention

While the time to commit is tolerable in the previous test, the number of aborts is high. It is important to remember that these aborts are not user-level events: they would be handled by the cooperative transaction manager in a layer above BOSS.

The goal of this experiment is to decrease the number of aborts by sending invalidation messages to the clients for objects that have been modified. Invalidate-on-commit sends to each client a message containing the OID and the new version number, immediately after each transaction that modifies an object.

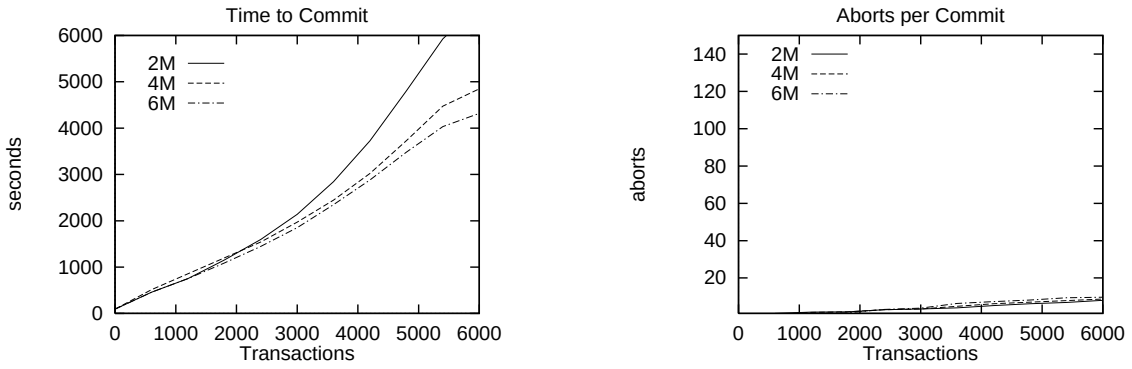


Figure 7.4: Invalidate on commit with minimal contention.

This protocol is quite successful at reducing the number of aborts. Note that the scale on the rightmost graph is one tenth that of Figure 7.3. However, the cost of this improvement is high: throughput is less than half that of the case without invalidations. The reason for the decrease is the cost of invalidation messages. Collecting, sending, receiving and processing the invalidation messages in this manner is expensive.

7.4.4 Invalidate-on-Commit and Low Contention

The low number of aborts for the previous test opens up the possibility of experimenting with increased contention. This experiment uses the same invalidation protocol but now on the low-contention workload. The time to commit increases but not

significantly. However, the number of aborts on average more than doubles. An increase caused by updates to hot objects in other clients' caches, since the invalidation messages do not always reach the other clients in time.

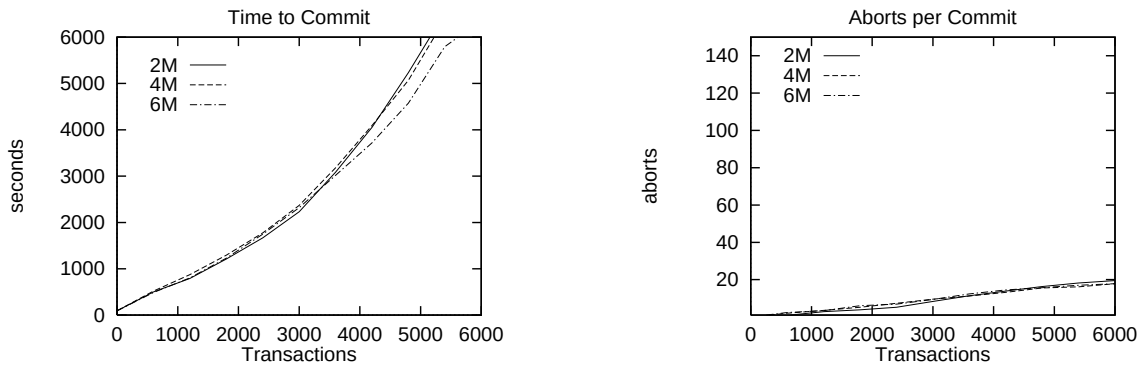


Figure 7.5: Invalidate on commit with low contention.

7.5 Invalidate-on-Abort

To reduce the cost of invalidations, we invented a new protocol whose purpose is to reduce the cost of invalidations without seriously affecting the number of aborts per commit. This protocol again applies the optimistic principle of apologizing rather than asking permission.

The invalidate-on-abort protocol sends out an invalidation message only when an abort occurs, rather than each time a commit occurs. By exploiting the extended delay between invalidations larger messages may be constructed that make more efficient use of the network and require less processor overhead. This is an example of introducing and exploiting asynchrony in an asynchronous system. Under the new protocol, the client caches slowly diverge from absolute correctness until a problem occurs, and then all the caches are brought back into a coherent state.

7.5.1 Invalidate on Abort and Minimal Contention

The invalidate on abort protocol trades aborts for decreased time to commit. (The corresponding experiment for invalidate-on-commit is shown in Figure 7.4 and the no-invalidations experiment is shown in Figure 7.3.) The new protocol offers almost the same time to commit as the no-invalidations experiment with a fraction of the aborts. The number of aborts is not relevant in the final analysis, but it helps better account for the underlying activities. In comparison to the invalidate-on-commit experiment, the time to commit is almost half; however, the aborts triple.

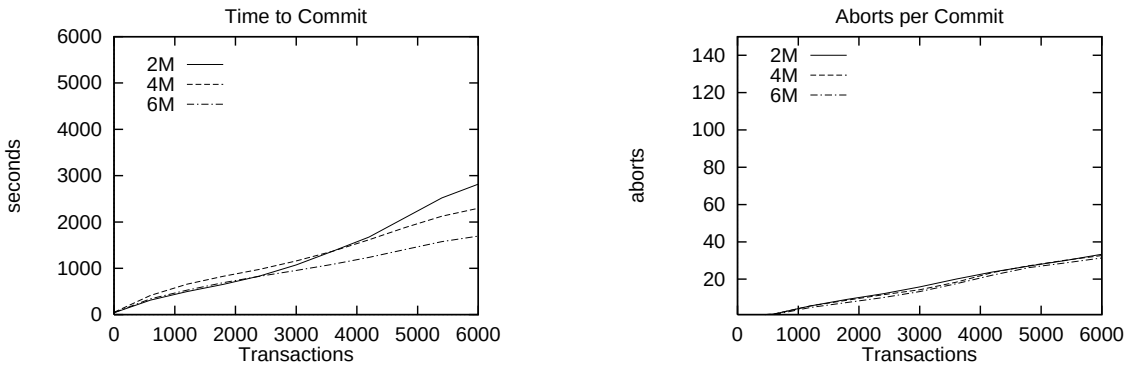


Figure 7.6: Invalidate on abort with minimal contention.

7.5.2 Invalidate on Abort and Low Contention

The invalidate-on-abort protocol is an intermediate point between no invalidations and invalidate-on-commit. This optimism decreases time to commit at the cost of higher abort rates. As contention increases, however, optimistic algorithms do not perform as well. For the low-contention workload, invalidate-on-abort again offers half the time to commit as invalidate-on-commit, but the cost has increased: invalidate-on-abort has five times the number of aborts of invalidate-on-commit. This is still a fraction of the no-invalidations case (not shown for this workload), but is significantly worse than the minimal-contention workload (Figure 7.3).

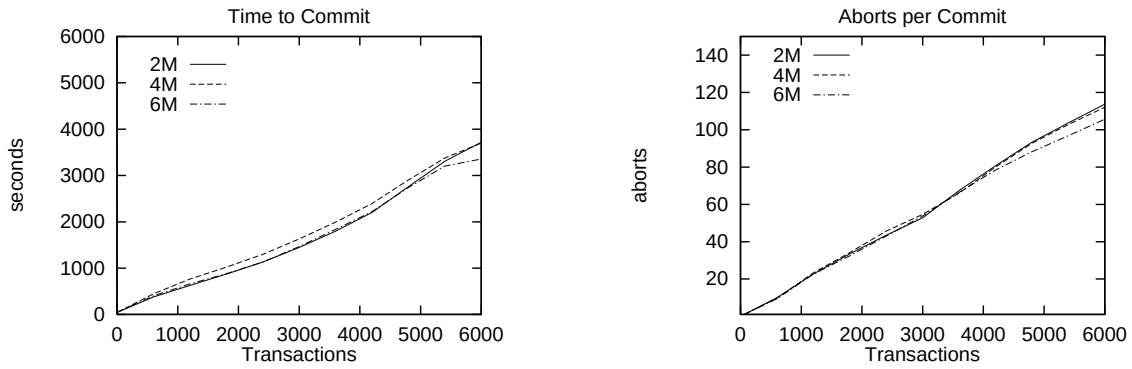


Figure 7.7: Invalidate on abort with low contention.

7.5.3 Conclusion of Experiment #1

The experiments measured the time to complete a fixed number of transactions and the number of aborts that occur. There were two variables: the client disk-cache size and the invalidation protocol. The client-disk cache ranged from 0M to 6M in 2M intervals. The invalidation protocols were no-invalidation, invalidate-on-commit, and the new invalidate-on-abort.

Our hypothesis that increasing client disk-cache size would decrease time to commit at the cost of increased aborts, holds true in the tests without invalidations. Adding invalidations significantly reduces the number of aborts irrespective of which invalidation protocol is used. We attempted to reduce the cost of the original invalidation protocol, invalidate-on-commit, with a new protocol, invalidate-on-abort. This new protocol reduces the cost of invalidations by a factor of two by allowing more aborts than invalidate-on-commit but an order of magnitude fewer than the number of aborts without invalidations.

7.6 Higher-Cost Transactions: Experiment #2

The transaction workload for which BOSS is designed consists of short, low contention transactions that are actually small groups of component operations from a higher-level cooperative transaction manager. However, we experimented with longer transactions to determine if the protocols could deal with delays.

Introducing delays has a number of potential negative impacts. Each transaction holds objects for a longer period of time. During this time another transaction can attempt to update one of the objects, a situation that would result in an abort. Further, the transactions are longer and restarting an aborted transaction doubles the time it takes to commit (assuming it is successful on the second try).

Introducing delays has a mitigating effect as well. In our model we maintain the same number of clients. With delays, each client submits transactions at a lower rate, Which decreases the peak loads on the server and decreases contention to a certain extent.

The base case for this experiment is no delays, as was used in Experiment 1. We introduce random delays approximately equal to the total time a transaction requires to commit: up to 1 second and between 1 and 2 seconds. The delays are introduced by sleeping after a transaction acquires its resources.

7.6.1 Minimal Contention

In the invalidate-on-commit tests, the introduction of delays has no significant effect on the rate of commits or aborts. There is a slight effect in the invalidate-on-abort tests: The 2-second delay test has a slightly higher time to commit than the 1-second and no-delay tests. In this test the average transaction requires 2.3 seconds to complete of which on average 1.5 seconds is sleep time. This number is almost reached by the 1 second and no delay tests, which require on average 1.75 seconds to complete a transaction. The number of aborts is impacted slightly in the invalidate-on-abort tests.

Minimal-Contention Delay Experiments

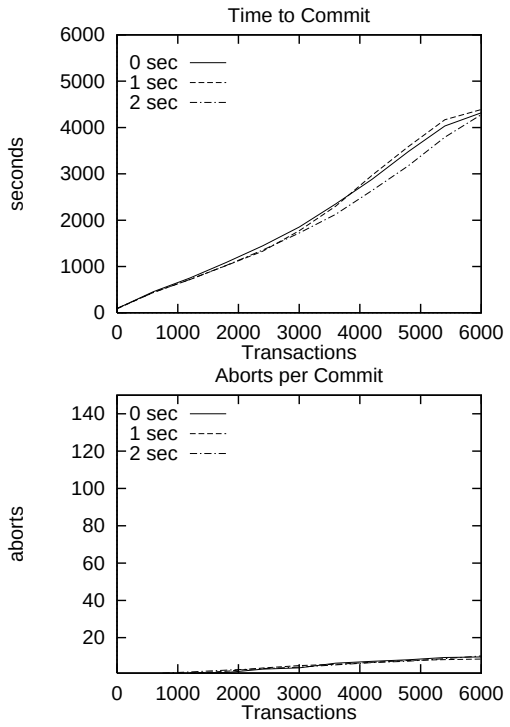


Figure 7.8: Invalidate-on-commit.

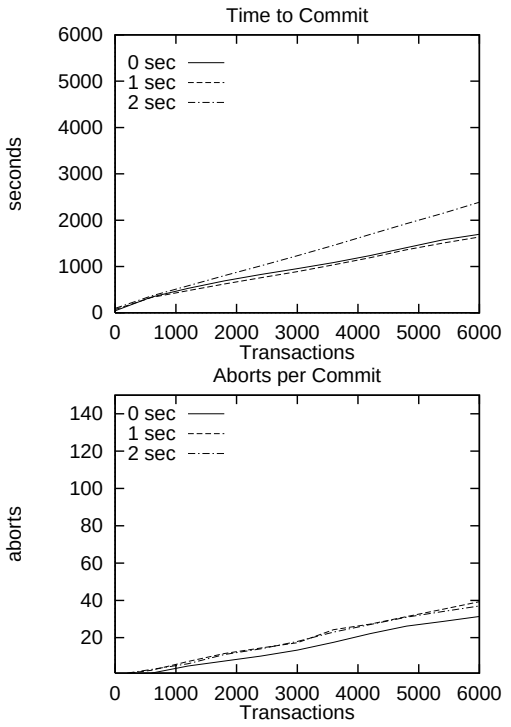


Figure 7.9: Invalidate-on-abort.

7.6.2 Low Contention

Introducing delays increases the cost of aborts and potentially increases the number of aborts, and increasing the level of contention aggravates this problem. However, as with the minimal contention workload, the low-contention workload is not seriously impacted by increased delays under the invalidate-on-commit protocol.

Transactions require 3.25 seconds to complete on average under the invalidate-on-abort protocol with no delays. This is larger than the 1.5 second average delay, so this test does not exhibit the increase in time to commit that the minimal contention test did. There is again a small increase in the number of aborts under this protocol.

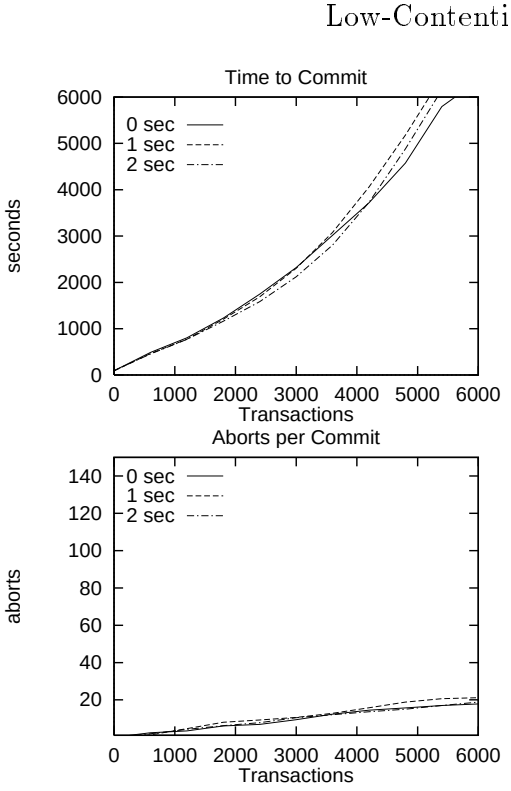


Figure 7.10: Invalidate-on-commit.

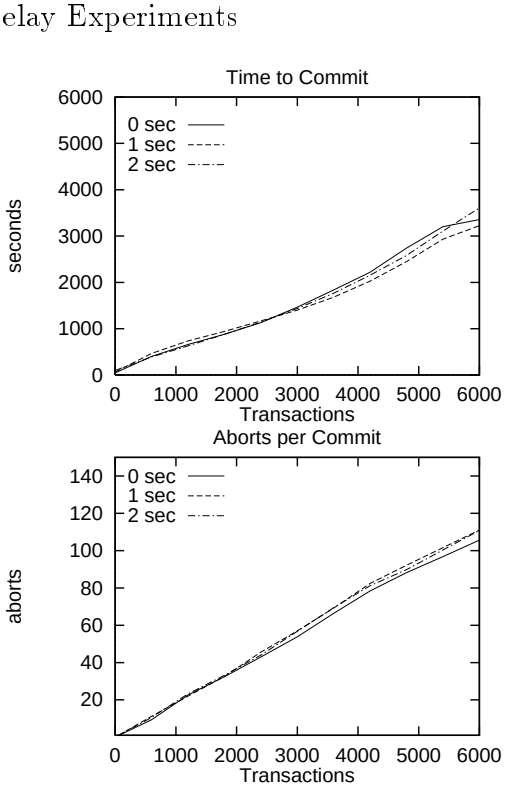


Figure 7.11: Invalidate-on-abort.

7.7 Experimental Conclusions

The first experiment demonstrates that client disk caching improves time to commit but results in an increased number of aborts without an appropriate cache coherency protocol. The invalidate-on-commit protocol is effective in reducing the number of aborts: Figure 7.4 shows a reduction of a full order of magnitude. The cost of this improvement is an increased time to commit.

Asynchrony proved to be a useful tool in developing a new cache coherency protocol, invalidate-on-abort, which reduces time to commit. Invalidate-on-abort introduces asynchrony by delaying invalidation messages from commit time until an abort occurs. This delay causes objects in the client caches to diverge further from the committed state than they would in invalidate-on-commit. Invalidate-on-abort exploits this asynchrony by building larger messages that make more effective use of the network and require less processing overhead. Invalidate on abort reduces the time to commit by more than 50% in both cases, but at the cost of increased aborts.

Chapter 8

Related Research

This section compares BOSS to other object and page servers, extensible databases, and distributed file systems. A number of other systems serve objects or pages: for example, Mneme [33], ESM [6], and the original ObServer [21]. Postgres [45] and Starburst [42] are two extensible relational database systems. Object servers draw from file system technology for persistent storage and cache coherency; this chapter closes with a comparison between BOSS and the Andrew File System [22] and the Sprite Log-Structured File System [40].

8.1 Object and Page Servers

The basic service BOSS provides is transaction support for persistent objects in a distributed client-server environment. Page servers and object servers both provide this service. These systems differ from traditional database applications in that the data is moved to a client machine's work space, operated on for some time, and then returned. A traditional database sends an operation to the database, the operation is performed locally, and then the result is returned.

Page servers act much like a file system. Pages are provided to clients and updates are performed on those pages at the client. An advantage of such a system is that

a data reference is a direct address and locating the appropriate page is also simple. Because the page is a fixed-size unit, moving it from the server to the client is simple. Page servers require that applications implement object abstractions on top of the raw pages, which leads to some problems. Because a reference to an object is a direct address, expanding or moving an object is difficult. Also, concurrency control is based on the physical layout of the data, not on the semantics of the operation that read or modified the data.

Object servers correct many of these problems by mapping application objects directly to storage objects. Concurrency control can be performed on application-level objects, so updates to distinct objects do not cause false aborts. Further, an object store takes advantage of the semantics of operations, so that concurrent application operations that do not conflict semantically do not cause unnecessary aborts. In an object store, the identity of an object is fully independent of its value, size, or location, so that an object can expand and move easily. However, this flexibility comes at the cost of added complexity in dereferencing an object.

ObServer [21], the precursor to BOSS, introduced several features: semantic clustering of related objects, a novel lock set for cooperative work, and a notification system that also supports cooperative work. BOSS improves on ObServer in the areas of distribution, extensibility, and performance. ObServer offered only multi-client, single-server distribution. Some work was 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 in BOSS. In addition, the ability to add new storage structures allows BOSS to be specialized for high performance for specific applications.

The Exodus Storage Manager (ESM) [6], a page server developed at the University of Wisconsin, provides support for values and indices. It generalizes the Aries transaction recovery system [32] 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 log sequence number is a strictly increasing address into the log. The BOSS implementation uses memory mapping to allow faster access to a record in the log.

The Wisconsin simulations show that caching locks across transactions significantly improves performance. The benefit of lock caching is that clients can keep data across transaction boundaries and access that data without having to suffer a round-trip communication delay to the server. ESM requires clients to remain in contact with a server to keep data cached. The BOSS algorithm allows clients to keep objects for an indefinite amount of time without accessing the server. This feature becomes more advantageous as clients perform more optimizing transformations, such as pointer swizzling or compression on data in the cache, and as architectures become heterogeneous, requiring translations between different caches.

SHORE (Scalable Heterogeneous Object REpository) is a second-generation descendent of ESM and Exodus from Wisconsin [12]. SHORE shares BOSS's goal of increased extensibility but realizes it in a different manner. The means to extend SHORE is the Value-Added Server (VAS), which allows sophisticated users to implement new abstractions directly on SHORE servers, bypassing the type system and standard communication overhead. A VAS is provided to allow backward compatibility with the UNIX file system. A VAS could potentially implement a relational database server, but VAS does not have the ability to define the concurrent semantics for the abstractions it creates as the BOSS storage class does. Concurrency control is performed on the underlying physical structures using read/write locking.

The Mneme Persistent Object Store [33] explores the integration of an object store with a persistent programming language. It provides a persistent heap of objects that are available in a distributed system. The object model for Mneme is fixed, but it does offer some support for specialized buffering policies. A general policy is called a

strategy, and an instantiation of a strategy is called a pool: for example, a strategy might be LRU paging and a specific pool might be an LRU cache with 2000 elements. Every object belongs to a pool, and this pool can change.

The intention of the strategy to provide specialized support for different classes of objects, as the BOSS storage class does. The BOSS storage class, however, provides not only different buffering policies but also different storage structures and object location methods. Perhaps the most important difference is that a BOSS storage class actually extends the BOSS interface: new access methods can have different abstract interfaces. BOSS enhances performance and increases concurrency using the semantics of this interface.

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

Camelot [11] is another system that could be classified as a page server, but it has a slightly different computation model. Camelot provides the abstraction of persistent virtual memory to data servers that run on the same node storing the persistent data. 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 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.

Camelot offers several choices for logging. A data server can choose either local or remote logging, and each transaction can choose new-value, old-value or new-value-only logging. Log sequence numbers facilitate recovery in a way similar to BOSS. Camelot uses a structure called the grid that is similar to the update and intentions list in BOSS; however, there is a major difference. The Camelot grid is an index structure stored in volatile memory and must be reconstructed after a failure. The

update and intentions lists in BOSS are stored persistently in the log and do not need to be recovered after a failure.

The Avalon [11] 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.

8.2 Object-Oriented Databases

The THOR (The Object Repository) system developed at MIT consists of a new programming language called θ (Theta), interfaces to popular programming languages called veneers, and an OODB as an integral part of θ's runtime environment called the OR (Object Repository).

We cooperated with the THOR group at the early design stages and BOSS shares several common attributes with the THOR OR. BOSS uses the same computation model: computation occurs on client workstations and shared servers provide persistence. THOR uses the same backward-validating optimistic concurrency control protocol as BOSS. Although THOR's current implementation is not extensible, this is a feature its designers would like to provide in the future.

BOSS differs from THOR in its support for extensibility and its approach to recovery. BOSS provides the storage-class abstraction, which allows implementors direct control over memory management and concurrency control. THOR plans at some point to allow client code to operate at the servers and to allow objects to manage their own concurrency control, but neither of these features has been fully designed, and their would provide less control than the BOSS storage class abstraction provides.

THOR's designers plan to offer highly available, replicated servers using a primary copy replication scheme [35]. The group has implemented this algorithm in the HARP

replicated file system, which provides near-UNIX semantics in a highly available file system [27]. A critical issue in a primary copy replicated system is the time required for the secondary machine after the primary fails. Techniques from BOSS's NR-WAL might prove useful in reducing the fail-over time. Of specific relevance is BOSS's use of log sequence numbers rather than clocked timestamps or operations counts.

GemStone operates in a distributed client-server system and extends SmallTalk with persistence, concurrent access, transactions, and indexing over large collections [30, 31, 38]. The original clients were PCs and the original servers were VAXs running VMS; the system is still in existence and now supports a wide variety of clients and servers as well as the C++ language.

GemStone is built on a query shipping model. The client machines send instructions to the servers and a result is returned. Most data exists on the server and all GemStone computation occurs at the server. The Server runs two types of processes, Gem processes, of which there is one per active client, and Stone processes, of which there is only one per server. Stone is GemStone's object store.

Stone manages concurrent access and provides fault tolerance by using shadowing. Each Gem process has its own virtual copy of the entire database. A true copy of any part of the database is made only when that part is modified. Upon successful termination of a transaction, changes are made permanent atomically by careful pointer manipulation.

8.3 Extended Relational Databases

Starburst [42, 19, 26] extends relational databases in five areas: external data storage, storage management, access methods, abstract types, and complex objects. Many of the issues Starburst addresses are at the data-model level and thus 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 are "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."

The Starburst architecture differs from that of BOSS, which is designed to add new functionality in a new storage class. In BOSS as in Starburst, the storage class can define its own storage layout and access methods. In addition, however, 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 at the client or the server.

The goals of the Postgres Storage Manager are instantaneous recovery, historical access, and utilization of new technology [45, 28]. Postgres provides a linear history and allows more than one version of an object to be viewed in a transaction. Because 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. The Postgres 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 requiring of large amounts of nonvolatile memory.

8.4 File Systems

A page server is similar to a file system with added support for transactions. Two interesting developments in file systems influenced 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 [40] 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 this information 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 [23].

AFS is interesting for its distribution properties. AFS 3.x [22] is a distributed file system that serves a purpose similar to Sun's NFS. However, the scale of the distributed system within which AFS operates is much larger than that of Sun's NFS: NFS was meant to operate within a single organization, whereas AFS can handle global distribution. One interesting feature of AFS that BOSS shares is nonvolatile client caching. Data can be cached on a client's local disk for extended periods of time. However, 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 [14] shows that this protocol is desirable 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, on the other hand, does not require that the cache be kept perfectly coherent. The version number associated with each object indicates whether the object is up to date, so BOSS can tolerate failures without any special means.

Chapter 9

Summary

The Brown Object Storage System (BOSS) makes contributions to the areas of cache coherency, recovery, and concurrency control.

- BOSS does not rely on cache coherency for transactional correctness, so that the cache coherency policy offering the best performance for a storage class can be chosen without regard to its correctness properties.
- No-redo write-ahead-logging (NR-WAL) improves upon traditional write-ahead logging by eliminating the redo phase from recovery processing. This improvement allows normal processing to begin sooner after a system failure.
- BOSS applies the theory of local concurrency control to separate concurrency control and storage management concerns. This model supports all existing database access methods as well as many that may be invented in the future.

We consider BOSS a truly asynchronous system because it has few restrictions on how or when internal operations occur. The database is never required to be in a consistent state. The only requirement is that the small portion of the database that a transaction observes must appear to be consistent at the instant the transaction prepares to commit.

Asynchrony in BOSS has two main effects. BOSS introduces a new recovery algorithm, NR-WAL, that requires no undos or redos during recovery, thus significantly reducing recovery time. The invalidate-on-abort cache-coherency policy offers performance advantages under the workload BOSS expects. BOSS does not actually require

cache coherency for correct operation. Transactions that observe old objects will abort, so it is up to the storage class to determine the best way to keep its objects up to date.

Asynchrony and modularity are complementary in a concurrent system. Even with a well-defined module structure, constraints on the timing of external events restrict the implementation of the module and therefore stifle experimentation and innovation.

BOSS achieves extensibility via the storage class, a novel concept that separates many database internals from the abstraction that the database provides. The storage class can extend the interface of the database by providing new functionality, and can also be used as an experimental testbed.

The ability to add new functionality at the storage level allows BOSS, and therefore applications built using BOSS, to take advantage of new developments in storage technology and move into new domains. New developments have also occurred in multidimensional range searching in secondary storage [16]. Any of these new structures could be tested as a storage class within BOSS. In addition, the storage class has already allowed BOSS to be used in application domains beyond those for which it was designed (See 4.5).

An algorithm can be incorporated into a storage class and tested in a common framework. The advantage to testing an algorithm within a complete, running database is that unexpected interactions can be discovered. Any simulation makes simplifying assumptions about the environment in order to focus better on the current problem. If these assumptions are not correct, the results of the simulation will not be valid. An experiment with a full running database does not suffer from this flaw.

Bibliography

[1] M. Atkinson et al. The Persistent Object Management System. Technical Report PRRR-1, The Universities of Glasgow and St. Andrews, 1983.

[2] François Bancilhon, C. Delobel, and Paris Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O2.* Morgan-Kaufmann, 1992.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[4] Peter Buneman and Malcolm Atkinson. Inheritance and persistence in database programming languages. In *ACM SIGMOD Proceedings*, pages 4–15, 1986.

[5] M. Carey et al. The architecture of the EXODUS extensible DBMS. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, pages 52–65, Asilomar Conference Center, Pacific Grove, CA, September 1986.

[6] 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.

[7] M. Carey et al. The EXODUS extensible DBMS project: An overview. Technical Report 808, Computer Science Department, University of Wisconsin - Madison, 1988.

[8] David Clark. as quoted in Chapter 9 of J. Hennesy and D. Patterson, *Computer Architecture a Quantitative Approach*, 1990, Morgan Kaufmann.

[9] G. Delott. Performance improvements in the ObServer Object Server. Masters thesis, Department of Computer Science, Brown University, 1989.

[10] EXODUS Project Document. EXODUS storage manager v3.0 architecture overview. Computer Sciences Department, University of Wisconsin-Maddison, (available by ftp from ftp.cs.wisc.edu), April 1993.

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

[12] M. Carey et al. Shoring up persistent applications. In *ACM SIGMOD Proceedings*, 1994.

[13] M. Fernandez, S. Zdonik, and A. Ewald. ObServer: A storage system for object-oriented applications. Technical Report CS-90-27, Brown University, Department of Computer Science, 1990.

[14] M. Franklin and M. Carey. Client-server caching revisited. In *International Workshop on Distributed Object Management*, Edmonton, Canada, 1992.

[15] M. Franklin, M. Zwilling, C. Tan, M. Carey, and D. Dewitt. Crash recovery in client-server EXODUS. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1992.

[16] Michael T. Goodrich, Jyh-Jong Tsay, Darren E. Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 714–723, November 1993.

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

[18] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormack, editor, *ACM SIGMOD Proceedings*, pages 47–57, Boston, MA, June 1984.

[19] L. Haas et al. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990. Also IBM Almaden Research Center Research Report RJ 7278 (68535).

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

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

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

[23] E. Kolodner, B. Liskov, and W. Weihl. Atomic garbage collection: Managing a stable heap. In *ACM SIGMOD Proceedings*, 1989.

[24] H. Korth and G. Speegle. Formal model of correctness without serializability. In *ACM SIGMOD Proceedings*, 1988.

[25] S. Leffler, M. McKusick, M. Karels, and J. Quaterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison Wesley, 1989.

[26] T. Lehman and B. Lindsay. The Starburst long field manager. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 375–383, Amsterdam, The Netherlands, August 1989. Also IBM Almaden Research Center Research Report RJ 6899 (65725).

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

[28] C. Lynch and M. Stonebraker. Extended user-defined indexing with application to textual databases. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 306–317, Los Angeles, CA, 1988.

[29] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions.* Morgan Kaufman, 1994.

[30] D. Maier and J. Stein. Indexing in an object-oriented DBMS. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, pages 171–182, Asilomar Conference Center, Pacific Grove, CA, September 1986.

[31] D. Maier and J. Stein. Development and inplementation of an object-oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[32] 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.

[33] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, April 1990.

[34] M. Nodine. A cooperative transaction model for design databases. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications.* Morgan Kaufman, 1990.

[35] B. Oki and B. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Systems*, 1988.

[36] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[37] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, June 1988.

[38] A. Purdy, B. Schuchardt, and D. Maier. Integrating an object server with other worlds. *ACM Transactions on Office Information Systems*, 5(1):27–47, January 1987.

[39] J. Richardson and M. Carey. Programming constructs for database system implementation in EXODUS. In *Proceedings of the 1987 ACM SIGMOD International Conference*, San Francisco, CA, May 1987. Also University of Wisconsin - Madison Computer Science Department Techreport 680.

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

[41] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, February 1994.

[42] P. Schwaarz et al. Extensibility in the Starburst Datbase System. In *International Workshop on Object-Oriented Database Systems*, 1986.

[43] G. Shaw and S. Zdonik. A query algebra for object-oriented databases. In *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, CA, February 1990.

[44] K. Smith and S. Zdonik. Intermedia: A case study of the difference between relational and object-oriented database systems. In *Proceedings of the ACM Object-Oriented Programming Systems, Languages and Applications Conference*, 1987.

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

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

[47] D. Wells, J. A. Blakeley, and C. W. Thompson. Architecture of an open object-oriented database management system. *IEEE Computer*, 25(10):74, October 1992.