Proc. IEEE 11th Int'l Conference on Data Engineering, March 1995, pages 308-315

Transactions in the Client-Server EOS Object Store

Alexandros Biliris and Euthimios Panagos
AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974
{biliris, thimios}@research.att.com

Abstract

This paper describes the client-server software architecture of the EOS storage manager and the concurrency control and recovery mechanisms it employs. Unlike most client-server storage systems that use the standard two-phase locking protocol, EOS offers a semi-optimistic locking scheme based on a multigranularity two-version two-phase locking protocol. Under this scheme, many readers are allowed to access a data item while it is being updated by a single writer. For recovery, EOS maintains a write-ahead redo-only log because of the potential benefits it offers in a client-server environment. First, there are no undo records, as log records of aborted transactions are never inserted in the log; this minimizes the I/O and network transfer costs associated with logging during normal transaction execution. Secondly, it reduces the space required for the log. Thirdly, it facilitates fast recovery from system crashes because only one forward scan of the log is required for installing the updates performed by transactions that committed prior to the crash. Performance results of the EOS recovery subsystem are also presented.

1 Introduction

Most commercial and experimental database systems of today operate in a client-server environment. The concurrency control and recovery sub-systems employed by most client-server architectures are based on the protocols found in centralized and replicated DBMSs. In particular, the standard two-phase locking protocol is used for concurrency control, while the redo/undo recovery protocol protects the database from transaction aborts and system crashes [2]. In this paper we describe the implementation and performance characteristics of a semi-optimistic two-phase locking protocol and a no-undo/redo recovery scheme, based on our experience in building the EOS client-server system.

In traditional centralized database architectures and most of today's commercial client-server relational database systems, queries and operations are shipped from client machines to the server which processes the requests and returns back the results. In contrast, the vast majority of object-oriented client-server DBMSs follows a data shipping approach where clients operate on the data items the server sends to them (e.g., [7, 9, 11, 10]). Although there is a number of alternatives on the granularity of the data items exchanged between the server and the clients, the majority of the client-server systems employ the page-server model because of its simplicity and potential performance advantages over the other alternatives [8]. In a page-server environment, the server and the clients interact by using pages or groups of pages.

In a data shipping client-server system each client has a buffer pool, also referred to as *client cache*, where it places the pages fetched from the server. Clients perform most of the database modifications, while the server keeps the stable copy of the database and the log. An important observation is that each client cache can be considered as an extension of the server's cache and the updated pages present in the client's cache can be considered as being shadows of the pages residing on the server. Hence, the two-version two-phase locking protocol [2] could be implemented with no additional overhead. Furthermore, if the pages updated by a transaction running on a client are never written to the database before this transaction commits, then there is no need to generate undo log records and the system is able to offer redo-only recovery. This is because the database will never contain modifications that must be undone when a transaction aborts or when the system restarts after a crash.

EOS is based on a page-server architecture following the data-shipping approach. The implementation of transaction processing in EOS involves two main components. The *concurrency control* subsystem provides correct concurrent execution of transactions accessing the same database. The *recovery* subsystem uses the information stored on a log file to provide database consistency despite transaction and system failures. To reduce the amount of work during crash recovery, EOS checkpoints the database periodically. It is essential that the concurrency control, logging, and checkpointing activities of a transaction manager interfere as little as possible with normal transaction execution. The major characteristics of the EOS transaction processing mechanism are the following:

- Concurrency control is based on the multigranularity two-version two-phase locking protocol. A given database page has a committed version present either in the server's buffer pool or on disk. A second version of the page may temporarily reside in the cache of a client that is in the process of updating it. If the client commits, the modified copy of the page is placed in the server's buffer pool and it becomes the committed version of the page. If the client aborts, the modified copy is discarded. The scheme allows many readers and one writer to concurrently access the same page without incurring extra overhead to the client and server buffer managers.

- Recovery is based on a write-ahead redo-only logging scheme that (a) minimizes the I/O and network transfer costs associated with logging during normal transaction execution because no undo log records are written, (b) reduces the space required by the log since log records contain only after images of updates, and (c) requires only one forward scan of the log in order to re-apply the committed updates, which results in fast system restarts.
- Checkpoints are non-blocking; active transactions are allowed to continue accessing databases while a checkpoint is taken.

The remainder of the paper is organized as follows. Section 2 presents an overview of the EOS client-server architecture and emphasizes the concurrency control and logging protocols employed. Transaction operations, such as commit and abort, are discussed in Section 3. Crash recovery and checkpointing are presented in Section 4. The performance of the logging and recovery implementation is presented in Section 5. We discuss related work in Section 6 and, finally, we state our concluding remarks in Section 7.

2 Architecture Overview

Figure 1 sketches the architecture of the EOS client-server storage manager. The EOS server is the repository of the database and the log. It mediates concurrent accesses to the database and restores the database to a consistent state when a transaction or system failure occurs.

EOS *databases* are collections of EOS *files* in which objects are stored. Databases are created in *storage areas* – UNIX files or raw disk partitions. Each storage area consists of a number of *extents* which are fixed-size sequences of physically adjacent disk *pages*. The allocation policy within storage areas is based on the binary buddy system which imposes minimal I/O and CPU costs and it provides excellent support for very large objects [4].

Objects are stored on slotted pages and they are identified by system generated unique object ids (oids). If an object cannot fit within a single page, EOS stores the object in a number of *segments* – contiguous pages allocated in an extent – and a descriptor that points to these segments is stored on the slotted page [3]. EOS provides transparent access to small and large objects. Both kinds of objects can be accessed either via byte-range operations such as read, write, append, insert, etc. – specially suited for very large, multimedia objects (gigabytes) – or directly in the client's cache, without incurring any in-memory copying cost.

An application program is linked with the EOS client library. The application may consist of many transactions but only one transaction at a time is executed. Each application has a buffer pool for caching the pages requested from the server, a lock cache, some transaction information, and a logging subsystem that generates log records for the updated pages. These log records are sent to the server during transaction execution and at commit time. Inter-transaction caching

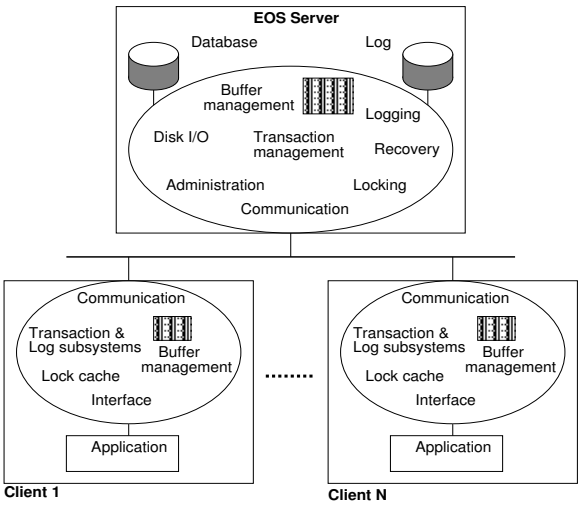

Figure 1: The EOS client-server architecture

is not currently supported and the application buffer pool is empty at the beginning of each transaction. A *least-recently-used* (LRU) buffer replacement policy is employed by the client and server buffer managers.

The EOS server is implemented as a multi-threaded daemon process. At transaction begin, the server assigns a new transaction identifier and creates a new thread when there is no active one serving this client application. At transaction abort, the server sends an *abort* message to the transaction the next time the transaction communicates with the server.

The communication between the server and the clients is done by using reliable TCP/IP connections over UNIX sockets [14]. To avoid blocking I/O operations, the server creates a disk process for each storage area accessed by client applications. These disk processes access directly the server buffer pool, which is stored in shared memory, and communicate with the server threads using semaphores, message queues and UNIX domain sockets [14].

2.1 Concurrency Control

EOS employes the two-version two-phase locking (2V-2PL) locking protocol. When the 2V-2PL protocol is applied in a client-server environment where each application has its own buffer pool, it enables many transactions to read an object while another transaction updates the same object without the need of maintaining different physical versions in the database. In addition, the 2V-2PL is coupled with multigranularity locking (MG-2V-2PL).

Lock acquisition and release is implicit for each read or write operation. Transactions acquire locks on data items before they access them, and they release all locks they hold when they are finished (committed or aborted). EOS supports three locking granularities: page, file, and database.

A page, the smallest lock granule, can be locked by a transaction T in one of the following modes:

Compatibility Table							
Mode	Existing						
Requested	IS	S	IX	X	SIX	IC	C
IS	Y	Y	Y	Y	Y	Y	N
S	Y	Y	Y	Y	Y	N	N
IX	Y	Y	Y	N	Y	Y	N
X	Y	Y	N	N	N	N	N
SIX	Y	Y	Y	N	Y	N	N
IC	Y	N	Y	N	N	Y	N
C	N	N	N	N	N	N	N

Table 1: The lock compatibility table

Lock Upgrade Table							
Mode	Granted						
Requested	IS	S	IX	X	SIX	IC	C
IS	IS	S	IX	X	SIX	-	-
S	S	S	SIX	X	SIX	-	-
IX	IX	SIX	IX	X	SIX	-	-
X	X	X	X	X	X	-	-
SIX	SIX	SIX	SIX	SIX	SIX	-	-
IC	-	-	IC	IC	IC	-	-
C	-	-	C	C	C	-	-

Table 2: The lock upgrade table.

Intention shared (IS): T intends to read an object belonging to a file residing on this page.

Shared (S): T wants to read an object stored on this page.

Intention exclusive (IX): T intends to update an object belonging to a file residing on this page.

Shared intention exclusive (SIX): There is a file object on this page and either T read the file object and intends to update an object in this file, or T updated an object belonging to this file and now wants to read the file.

Exclusive (X): T wants to update an object stored on this page.

Intention commit (IC): T had an IX or SIX lock on this page and it is in the process of committing.

Commit (C): T had an X lock on this page and it is in the process of committing.

Table 1 determines whether a lock request can be granted or not – "Y" means that the lock request can be granted and "N" means that the request has to be blocked. When a transaction locks a page, the file containing this page is also locked in the corresponding intention mode. In addition, when a transaction locks a file in either S or X mode, the pages this file contains are not locked explicitly, unless the transaction locked the file in S mode and updated a page in the file. Transactions that are in the process of committing their updates are blocked when there are active transactions that read the updated pages in order to generate serializable schedules. In addition, transactions attempting to read a page which has been updated by a committing transaction are blocked. In this way, committing update transactions may be blocked only for a finite time period.

Table 2 is used for lock upgrades. A lock upgrade occurs when a transaction holding a lock on a data item wants to lock this item in a different mode. An entry containing '-' corresponds to either a violation of the assumption that no locks will be acquired while the transaction entered its commit phase, or an impossible situation.

In every lock-based concurrency control algorithm, deadlocks may occur. Since in a client-server environment delays are inherent due to the communication network and the fact that most of the computation is performed by the client, mistaking a long wait for a deadlock would affect the performance of the system dramatically. For this reason, EOS uses a deadlock detection algorithm as opposed to a timeout approach.

Deadlock detection is performed when a lock request has to be blocked. The deadlock detection algorithm consists of two steps. The first step attempts to discover a cycle involving transactions that tried to upgrade their locks on the same locked page. If no cycle is discovered during the first step and the blocked request is waiting for a transaction that has a blocked lock request, then the second step dynamically constructs the waits-for graph (WFG) and searches for a cycle by traversing the constructed graph in a depth first fashion. EOS evicts the transaction whose lock request resulted in the formation of a deadlock cycle. An alternative approach that we may adopt in the future is to never evict a committing transaction, unless it is the only choice[1].

2.2 Recovery

EOS maintains a write-ahead redo-only log on disk. The generated log records contain the after image of the modified page or the byte ranges within this page that have been updated. The log contains three kinds of records: *checkpoint* records indicating that a checkpoint has been taken; *commit* records indicating that a transaction has successfully committed; *redo* records containing the results of the updates generated by committed transactions.

EOS partitions the log into two kinds of logs: a *global* log and a number of *private* logs, as shown in Figure 2. Each private log is associated with one transaction only and contains the physical after images of

[1]EOS becomes aware of the fact that a transaction is about to commit because a transaction that wants to commit requests from the lock manager to convert its locks to commit locks (IC and C).

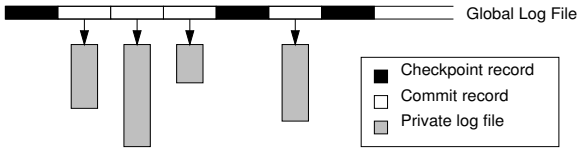

Figure 2: The physical structure of the log file

the updates generated by the corresponding transaction. The global log contains records that are either commit or checkpoint records. A commit record contains the committed transaction's id and the address of its private log. The checkpoint record contains the location of the commit record belonging to the first transaction that placed a commit record and which had not finished posting its updates to the database while checkpointing was in progress. When a transaction aborts its private log is simply discarded.

Each log record has a small header that contains the page id, the byte range updated and the actual update. Currently, EOS performs full page logging and the byte range value of each log record is equal to the database page size. Log records may be sent to the server asynchronously while the transaction is running. When a transaction commits, however, all of its remaining log records are sent to the server. When a log record is written in the private log, the log manager returns a key for the record – called *log sequence number* (LSN) – that can be used later to access that record.

3 Transaction Execution

A committed transaction T goes through the following phases: *active, committing,* and *write*. T is in the active phase from the time it starts up to the time it finishes normal execution and it is ready to commit. At this point T must convert all exclusive locks it acquired to commit locks and send the remaining of the log records to its private log. During this process, T is in the *committing phase*. After the log records are written to the stable storage and a commit record is placed in the global log, T is said to be committed. The last phase is the one where T establishes its updates in the server buffer pool; this phase is called *write phase*. A transaction can be aborted at any time during the active and the committing phases. Once the transaction reached the write phase, its updates are guaranteed to persist.

3.1 Active Phase

During normal transaction processing locks are acquired from the server and database pages are cached in the application's private buffer pool. When the private pool is full, pages are replaced to make room for new ones. In the redo-only logging approach, a dirty page being replaced should never be written to its disk location in the database before the transaction commits because no undo information is kept in the log. Avoiding to write an uncommitted dirty page to the database is achieved by either prohibiting the replacement of modified pages or by employing a shadow-like approach.

Prohibiting the replacement of modified pages is not an option because it severely restricts the number of pages a transaction can update. On the other hand, shadowing can be implemented by storing the replaced pages either in the database (as in O_2 [7]) or in a *swap* area. We abandoned following O_2's approach because of the extra overhead it adds to the server buffer manager and recovery subsystem.

When a dirty page is replaced from the client's private pool, it is placed in the swap space and the location of the page is kept by the client buffer manager. The next time the page needs to be fetched in the application's cache, the page is taken from the swap space. The swap area in EOS can be either on the client or the server machine and it is specified in the configuration files offered by EOS. As a special case, the private log of a transaction could be used as the swap space for uncommitted updates – this is the default in our current implementation [5]. When a dirty page is replaced from the application's private pool, EOS generates a log record whose data part contains the entire page and the LSN of the log record is kept by the client's buffer manager.

For every page P in the private buffer pool of a transaction T there is a control block (CB) containing locking information related to P and a pointer to a buffer frame where P is stored. When a transaction wants to read/write an object in a page, it calls the buffer manager and passes along the lock mode L that needs to be acquired on P. The buffer manager executes the following algorithm.

1. Scan the buffer pool to locate the CB of P.
2. If the CB of P is not found then make room in the buffer pool by evicting the least recently used page P_{LRU}. If P_{LRU} is dirty then send it to the private log file and save the returned LSN in the CB of P_{LRU}.
3. If the CB of P is found and the page is present in the buffer pool then continue with step 5 if the lock mode needs to be upgraded, else return.
4. If the CB of P is found and the page is swapped out, then make room in the buffer pool by replacing a page as in step 2 above and request P from the private log using the LSN stored in P's CB.
5. Request from the server L-lock on P and IL-lock on P's file.

3.2 Transaction Abort

When a transaction T aborts, no undo action needs to be carried out besides cleaning up possible object copies in the transaction private space and removing the private log. In other words, T sends an abort message to the server, frees various control structures used, and purges the local cache.

When the server receives an *abort transaction* message, it frees all resources used by T. In particular, all locks held by T are released, the private log associated with T is discarded, and T is added to the list of the aborted transactions.

3.3 Transaction Commit

When the transaction T finishes its active phase and it is ready to commit, it follows the steps presented below:

1. Without waiting for a response send a *convert locks* message to the server.
2. Send asynchronously to the private log all remaining log records.
3. Send a *commit transaction* message to the server and wait for the acknowledgment.

While T is executing step 2 or waiting on step 3, the server may reply with an *abort* message. The reason for the abort may be: (a) T was involved in a deadlock that materialized when the server was acquiring the commit locks on behalf of T, or (b) an internal error occurred while writing the log records or flushing the private log.

When the server receives a *convert locks* message from a transaction, it upgrades all `IX` and `SIX` locks to `IC` and all `X` locks to `C`. Next, the server releases all `IS` and `S` locks and replies with a *success* message.

All the log records generated by a committing transaction have been sent to server when the transaction sends the *commit* message to the server. When the server receives the *commit* message it follows the steps described below.

1. Flush the private log to stable storage.
2. Insert a *commit* record in the global log and flush the global log.
3. Send a *success* message to the application.
4. Install the updates performed by the committed transaction.
5. Release all remaining locks (i.e., `C` and `IC` locks).

If an error occurs while the server executes the first two steps of the above algorithm, it aborts the transaction and replies with an *abort* message.

3.4 Write Phase

The log records generated by a committed transaction are used to apply the updates to the database. Thus, the fourth step of the commit algorithm presented in the above section is done as follows.

1. For each log record in the private log of the committed transaction do the following:
   (a) If the log record contains the after image of an entire page, then overwrite the page if it is present in the server buffer pool. Otherwise, place the data part of the log record in the pool.
   (b) If the log record contains an update performed on some byte range of a page, then overwrite this particular byte range. If the page is not present in the server buffer pool, bring it in first and then perform the overwriting.

3.5 Recovery for Large Objects

Recovery for large objects differs from that mentioned above. First, large objects are not buffered in the server buffer pool. Secondly, updates on large objects are applied directly to the database without, however, overwriting the object in the database [3]. When a transaction gets aborted, the updates it performed on the allocation bitmaps are thrown away. In addition, no other transaction can see these changes because of the write lock held on the page containing the large object descriptor.

4 Recovery from Server Crashes

To reduce the amount of work the recovery manager has to do during recovery from a system crash, the EOS server periodically issues checkpoints. In the current implementation, the checkpoint algorithm executes the following steps.

1. Let *RestartLoc* be the earliest *commit* record inserted in the global log by transactions that have not finished their write phase yet.
2. For each dirty page in the server buffer pool do:
   (a) Latch the page and write it to disk.
   (b) Mark the page clean and unlatch it.
3. Place a checkpoint record in the global log containing the *RestartLoc* computed at the first step.
4. Save the location of the checkpoint record in a place well known to the restart procedure.

Even though the second step of the checkpoint algorithm does not block active transactions while the server buffer pool is flushed to stable storage, flushing of dirty pages makes the checkpoint algorithm expensive. Currently, we are implementing a new checkpoint algorithm that is based on the *fuzzy checkpoint* method described in [2]. This algorithm flushes to stable storage only the frequently updated pages that were not flushed out since the last checkpoint.

System restart is done by scanning the log file and re-doing all the updates made by committed transactions in exactly the same order as they were originally performed. The redo processing starts at the *RestartLoc* value present in the last complete checkpoint record. After the database state has been restored a checkpoint is taken and the system is operational again. If the server crashes while restart is performed, the subsequent restart performs the same work again in an idempotent fashion.

5 Performance Results

In this section we present an initial study of the performance of the logging and the recovery components of EOS. In this study we measure the logging overhead, the time required to abort a transaction, and the time spent when restarting the system after a crash. Some comparison with the client-server Exodus storage manager (ESM-CS) [9] is also presented. This comparison is necessarily rough, because ESM-CS uses a totally different recovery protocol and a different hardware configuration.

DB Name	Objects in DB	Object Size	Objects per page	Pages in DB
FewObj	6000	500	6	1000
MediumObj	30000	100	30	1000
ManyObj	100000	20	100	1000

Table 3: The configuration of the databases used

DB Name	Logging (sec) ON	OFF	Logging Overhead
FewObj	17.2	15.7	1.5 (9.55%)
MediumObj	18.9	17.3	1.6 (9.24%)
ManyObj	25.6	23.8	1.8 (7.56%)

Table 4: Writing to the log disk overhead

All the experiments presented were run on two SPARCstation 10's running SunOS version 4.1.3 and having 32M bytes of main memory and 142M bytes of swap space. The client and server processes were run on separate machines and they were connected by an Ethernet network. The database was stored in a raw disk partition to minimize the operating system's buffering overhead and the database page size was 4K bytes. The log was stored in a regular UNIX file and `fsync()` was used for flushing any internal operating system buffers to disk at transaction commit time. All the times reported are in seconds and they were obtained using `gettimeofday()`.

5.1 Database and System Model

Due to the limited number of published performance results for the logging and recovery components of client-server systems, we adopted a variation of the model presented in [9]. Table 3 describes the three databases used for the experiments we ran. Each database consists of 1000 pages and the key difference among them is the number of objects they contain. The first database *FewObj* contains 6000 objects of size 500 bytes each. The second database *MediumObj* contains 30000 objects of size 100 bytes each. The third database *ManyObj* contains 100000 objects of size 20 bytes each.

We used only one kind of transaction for the experiments performed: *Update*, which sequentially scans the entire database and overwrites part of each encountered object. The server buffer pool was set to 2000 pages so that the entire database was cached in main memory and the writing of log records was the only I/O related activity. The application's buffer pool was also set to 2000 pages. In this way the entire database fits in the private pool during transaction processing. However, as mentioned in 2, the local pool is empty at the beginning of each run. All the numbers presented in the forthcoming sections were obtained by running each transaction five times and taking the average of the last four runs.

5.2 Logging Results

In the first set of experiments we measured the overhead of writing the log records to the log disk, as it was observed by the application process. In order to compute this overhead we altered the EOS server to allow the writing of the log records to be selectively turned on and off. Table 4 shows the average times computed and Figure 3 illustrates them pictorially. The execution time for a transaction includes the time to initialize all EOS internal structures, execute and commit the transaction as well as the generation, shipping, and writing of log records and the forcing the log to stable storage.

Table 4 indicates that the overhead of shipping log records to the server and forcing them to disk decreases as the number of objects accessed by the application program increases. As mentioned in Section 2.2, EOS logs entire pages. Because the number of pages that are updated is the same in all three experiments, the number of log records generated is also the same. Thus, the logging overhead is reduced when the processing time of the application program increases.

While in EOS the logging overhead decreases when the processing time is increased, in ESM-CS the logging overhead increases. Because EOS logs pages whose after image can found in the application's buffer pool, there is no need to write log records in a separate log page for each operation that updates an object, as done by ESM-CS. Thus, the construction of the 32-byte log header is the only processing overhead related to the generation of a log record. Figure 4 illustrates the logging overhead of the *ManyObj* experiment for both EOS and ESM-CS under the assumption that regular transaction execution (including logging) takes one time unit. In this experiment, the logging overhead of ESM-CS is almost 5 times higher than the logging overhead of EOS. ESM-CS generated 100 log records per page while EOS generated only one. Secondly, for each individual page that was modified ESM-CS generated 6400 bytes of log headers (each log header takes 64 bytes) while EOS generated only 32 bytes.

5.3 Transaction Abort and Recovery Results

The time to abort a transaction was measured by the same set of experiments that were run to measure the impact of the logging subsystem during normal transaction processing. This time, each transaction is aborted after it finishes normal execution and the shipping of all log records to the server. Table 5 shows the results of these experiments together with the normal processing time (taken from Table 4) for comparison. Figure 3 illustrates these results pictorially.

The abort tests showed a slight increase in the time needed to abort a transaction as the number of updated objects increased. The results presented in Table 5 correspond to the time needed to release all the locks held by the aborted transaction plus the time needed to clean all data structures used on the client. The release of locks takes the same time for all three

DB name	Execution time (sec)	Abort time	Restart time
FewObj	17.2	1.0	1.3
MediumObj	18.9	1.1	1.3
ManyObj	25.6	1.7	1.3

Table 5: Transaction abort and Restart time

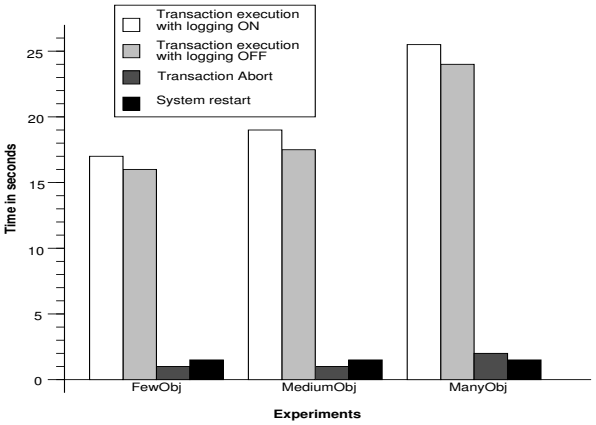

Figure 3: EOS performance results

databases since page level locking is used by EOS and the same number of pages is accessed. Thus, the cleaning of the data structures used is responsible for the increase in the abort time. This is reasonable since the number of structures used corresponds to the number of objects accessed. The fact that no transaction rollback and writing of compensation log records are needed for aborting a transaction in EOS, keeps the abort time very small compared to normal processing time. This is in contrast to the time needed to abort a transaction in ESM-CS that uses a redo/undo recovery protocol. ESM-CS has to read all pages updated by the transaction from the database and generate compensation log records.

For measuring the time needed to redo the updates performed by a committed transaction, we turned off the checkpoint activity and the server was crashed immediately after the transaction committed. During restart, the entire log had to be scanned and the updates made by the committed transaction were redone. The restart tests showed that the time needed to redo the committed updates is independent from the number of objects updated. This is so because the number of log records processed during restart was the same for all three databases used.

The times showed in Table 5 indicate that EOS offers a fast restart procedure compared to normal processing time. Because only after images of updates are logged there is not need to analyze the transaction's log. Thus, only one scan over the log is performed. The ESM-CS performs considerably worse that EOS since it has to analyze all log records written first and

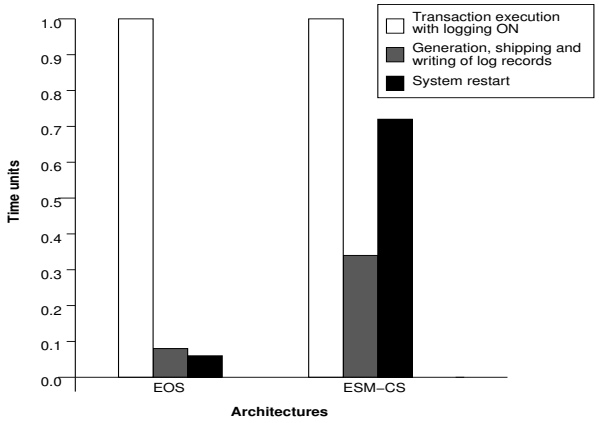

Figure 4: Normalized logging and system restart overhead

then redo the committed updates by re-scanning the log. Figure 4 illustrates the restart overhead of the *ManyObj* experiment for both EOS and ESM-CS.

6 Related Work

Although client-server systems have received a great deal of attention in recent years, information on recovery problems and solutions of existing client-server architectures is limited.

The Exodus client-server storage manager (ESM-CS) [9] employs an ARIES-based [12] recovery approach. Unlike ESM-CS that requires three passes over the log during restart recovery, EOS needs only one pass. In addition, ESM-CS requires each database page to contain an LSN-like field and handles large objects in a page-at-a-time fashion by stripping the headers present on them. EOS, on the other hand, does not store log related information on database pages and it does not alter the data of the segments where large objects are stored.

Objectstore [11], an object-oriented database management system based on a memory mapped architecture, uses a strict two-phase page-level locking algorithm and the write-ahead log protocol for recovery. In addition, Objectstore forces to disk all the updated pages at transaction commit.

O_2 [7] employs an ARIES-based recovery approach using shadowing to provide a Redo-only scheme. Modified pages that are replaced from a client's cache are sent to the server's and they are written to a shadow area when they are replaced from the server's buffer pool. EOS cannot place dirty pages in the server's buffer pool because in MG-2V-2PL an `S` is compatible with an `X` lock. Alternatively, EOS logs the entire page and it does not have to keeps track of the shadow pages as O_2 does.

ORION-1SX [10] uses both logical and physical locking. The logical locking is applied on the class hierarchy, whereas the physical locking is used for transferring objects atomically. ORION-1SX employs an undo-only recovery protocol and pages updated by a committed transaction are flushed to stable storage as

part of the commit process.

7 Conclusions

EOS is a storage manager that has been prototyped at AT&T Bell Laboratories as a vehicle for research into distributed storage architectures for database systems and specially those that integrate programming languages and databases. EOS is the storage manager of ODE [1], an object oriented database management system also being developed at Bell Laboratories.

In this paper we have described the client-server architecture of EOS and the concurrency control and recovery mechanisms it provides. The MG-2V-2PL concurrency control protocol was chosen with the goal of increasing the concurrency level of the system in the client-server environment. The no-undo/redo recovery method was designed with the goal of minimizing the impact of the recovery related activities during normal transaction processing, while providing fast transaction abort and system restart times. We also presented measurements of the recovery implementation in EOS. From the results computed, and from the limited number of published performance results for logging and recovery systems, we concluded that the recovery overhead in EOS is minimal, despite the write-intensive nature of the tests we ran.

We are currently working on issues related to inter-transaction caching within the same application and across applications that share the same cache, providing support for multiple servers, distributed transactions, and client-side logging [6, 13].

References

[1] R. Agrawal and N. Gehani. ODE (Object Database and Environment): The language and the data model. In *Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data,* Portland, Oregon, June 1989.

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

[3] A. Biliris. An efficient database storage structure for large dynamic objects. In *Proceedings of the Eighth International Conference on Data Engineering,* Tempe, Arizona, pages 301–308, February 1992.

[4] A. Biliris. The performance of three database storage structures for managing large objects. In *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data,* San Diego, California, pages 276–285, May 1992.

[5] A. Biliris and E. Panagos. EOS User's Guide, Release 2.0. Technical report, AT&T Bell Laboratories, May 1993.

[6] A. Biliris and E. Panagos. A high performance configurable storage manager. In *Proceedings of the Ninth International Conference on Data Engineering,* Taipei, Taiwan, March 1995. To appear.

[7] O. Deux et al. The O_2 system. *Communications of the ACM,* 34(10):51–63, October 1991.

[8] D. J. DeWitt, D. Maier, P. Futtersack, and F. Velez. A study of three alternative workstation-server architectures for object-oriented database systems. In *Proceedings of the Sixteenth International Conference on Very Large Databases,* Brisbane, pages 107–121, August 1990.

[9] M. Franklin, M. Zwilling, C. Tan, M. Carey, and D. DeWitt. Crash recovery in client-server EXODUS. In *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data,* San Diego, California, June 1992.

[10] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering,* 2(1), March 1990.

[11] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM,* 34(10):51–63, October 1991.

[12] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems,* 17(1):94–162, March 1992.

[13] E. Panagos, A. Biliris, H.V. Jagadish, and R. Rastogi. Exploiting client disks for high performance client-server architectures. Submitted for publication.

[14] R. Stevens. *UNIX Network Programming.* Prentice-Hall Software Series, 1990.