The VLDB Journal (1997) 6: 209–223

The VLDB Journal

Synchronization and recovery in a client-server storage system

E. Panagos, A. Biliris

AT&T Research, 600 Mountain Avenue, Murray Hill, NJ 07974, USA

Edited by R. King. Received July 1993 / Accepted May 1996

Abstract. Client-server object-oriented database management systems differ significantly from traditional centralized systems in terms of their architecture and the applications they target. In this paper, we present the client-server architecture of the EOS storage manager and we describe the concurrency control and recovery mechanisms it employs. EOS offers a semi-optimistic locking scheme based on the multigranularity two-version two-phase locking protocol. Under this scheme, multiple concurrent readers are allowed to access a data item while it is being updated by a single writer. Recovery is based on write-ahead redo-only logging. Log records are generated at the clients and they are shipped to the server during normal execution and at transaction commit. Transaction rollback is fast because there are no updates that have to be undone, and recovery from system crashes requires only one scan of the log for installing the changes made by transactions that committed before the crash. We also present a preliminary performance evaluation of the implementation of the above mechanisms.

Key words: Client-server architecture, Object management – Transaction management – Concurrency control – Locking – Recovery – Logging – Checkpoint

1 Introduction

The major components of any database storage system are the ones that provide support for concurrency control and recovery. Concurrency control guarantees correct execution of transactions accessing the same database concurrently. Recovery ensures that database consistency is preserved despite transaction and process failures. To achieve these goals, a storage system controls access to the database and performs some bookkeeping activities such as locking and logging during normal transaction execution. The database is checkpointed periodically in order to reduce the amount of work that has to be done during restart after a crash. It is essential that the concurrency control, logging, and checkpointing activities of a storage manager interfere as little as possible with normal transaction execution.

Today, most commercial and experimental database systems operate in a client-server environment. Providing transactional facilities in a storage manager for client-server object-oriented database management systems (OODBMSs) raises many challenging issues because of the significant architectural differences between client-server and centralized systems and the differences in the applications targeted by them. This paper presents a semi-optimistic two-phase (2PL) locking protocol and a redo-only recovery scheme that are being used in the EOS client-server object storage manager. We discuss implementation as well as performance characteristics of these protocols.

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 the results. In contrast, most of the client-server OODBMSs follow a data-shipping approach, where clients operate on the data items the server sends to them (e.g., Kim et al. 1990; Deux et al. 1991; Lamb et al. 1991; Franklin et al. 1992a). Although there are several alternatives for the granularity of the data items exchanged between the server and the clients, virtually all client-server systems have adopted the page-server model because of its simplicity and the potential performance advantages over the other alternatives (DeWitt et al. 1990). Under the page-server model, the server and the clients interact with each other by exchanging database 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. The 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 seen as an extension of the server's cache and, thus, updated pages present in a client cache can be considered as being shadows of the pages residing on the server. Hence, the two-version 2PL (2V-2PL) protocol (Bernstein et al. 1987) could be implemented easily with no additional overhead. Furthermore, if the pages updated by a transaction running on a client are never written to the database before the 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 rolled back when a transaction aborts or when the system restarts after a crash.

EOS is a data-shipping client-server storage manager based on the page-server architecture with full support for concurrency control and recovery. EOS has been prototyped at AT&T Bell Laboratories as a vehicle for research into distributed storage systems for database systems and specially those that integrate programming languages and databases. EOS is the storage manager of ODE (Biliris et al. 1993), an OODBMS that has been developed at AT&T. The major characteristics of the transactional facilities provided by EOS are the following.

- The concurrency control mechanism combines the multigranularity locking protocol (Gray and Reuter 1993) with the 2V-2PL protocol (MG-2V-2PL). The minimum locking granularity is a database page. For a given database page, there is always one committed version of the page in the server buffer pool or on disk. A second version of the page may temporarily reside in the cache of a client which is executing a transaction that updated the page. If the transaction commits, the modified copy of the page is placed in the server buffer pool and it becomes the committed version of the page. If the transaction aborts, the modified copy is discarded. In other words, a dirty page is placed in the server buffer pool only if the transaction that modified it commits. This scheme allows many readers and one writer to access the same page simultaneously without incurring additional overhead to the client and server buffer managers.
- Recovery is based on write-ahead redo-only logging. Transactions do not generate undo log records, and the updates made by an active transaction are posted to the database only after the transaction commits. Consequently, transaction rollback is very efficient because the updates performed by an aborted transaction do not have to be undone. In addition, system restart recovery requires only one scan of the log file in order to reapply the updates made by committed transactions before the crash. Another important feature of the redo-only protocol employed by EOS is that the number of pages a transaction can update is not bounded by the size of the client buffer pool.
- Checkpoints are non-blocking; active transactions are allowed to continue their processing while a checkpoint is taken. In particular, EOS employs a *fuzzy checkpointing* algorithm (Bernstein et al. 1987; Franklin et al. 1992) that stores only state information on disk and avoids the synchronous writing of updated pages.

The remainder of the paper is organized as follows. Section 2 presents an overview of the EOS architecture. Section 3 analyzes the concurrency control and logging protocols employed by EOS and discusses transaction operations such as commit and abort. Recovery from server crashes and the checkpoint algorithm are presented in Sect. 4. Section 5 contains the results from several performance experiments. Related work is covered in Sect. 6 and, finally, we conclude our presentation in Sect. 7.

2 EOS architecture overview

In this section, we present a brief overview of the facilities provided by EOS and we describe the distributed architecture of EOS.

2.1 EOS facilities

EOS provides facilities for storing and manipulating persistent objects. File objects serve as a mechanism for grouping related objects. Files may contain other files. Every object, including file objects, is a member of exactly one file object and, thus, objects form a tree where internal nodes are file objects and leaves are ordinary (non-file) objects. EOS databases are collections of files and ordinary objects. When a new database is created, a file object is automatically created that serves as the root file of all objects within this database. A database consists of several *storage areas* – UNIX files or disk raw partitions. A storage area is organized as a number of fixed-size *extents* which are disk sections of physically adjacent pages. *Segments* are variable-size sequences of physically adjacent disk pages taken from the same extent. Each extent has a bitmap associated with it. The allocation policy within storage areas is based on the binary buddy system (Biliris 1992a). This scheme imposes minimal I/O and CPU costs and it provides excellent support for very large objects (Lehman and Lindsay 1989; Biliris 1992b).

Objects are stored on pages, one after the other starting at one end of the page, and they are identified by system-generated unique object ids (OIDs). The OID consists of the storage area number and the page number within the area the object is stored in, plus a slot number. The slot number is an index into an array of slots which grows from the other end of the page toward the objects. Slots contain pointers to the objects on the page. In this scheme, objects can be rearranged within a page without affecting OIDs. In addition, the OID contains a number to approximate unique ids when space is reused.

EOS objects are uninterpreted byte strings whose size can range from a few bytes to gigabytes. If an object cannot fit within a single page, EOS stores the object in as many segments as necessary and a descriptor is placed in the slotted page; the descriptor points to the segments in which the actual object is stored (Biliris 1992a). EOS provides transparent access to both small and large objects. Both kinds of objects can be accessed either via byte-range operations such as read, write, append, insert bytes, etc., or directly in the client's cache, without incurring any in-memory copying cost. The byte-range operations are important for very large objects, e.g., digital video and audio, because there may be memory size constraints that would make it impractical to build, retrieve or update the whole object in one big step.

EOS offers extensible hashing indexing facilities which support variable size keys and user-defined hash and comparison functions. Other index structures can be built on top of EOS by using page objects – objects that expand over the entire available space of a page. For example, B-trees have been built in this way and they are used in ODE (Biliris et al. 1993). The EOS architecture has been designed

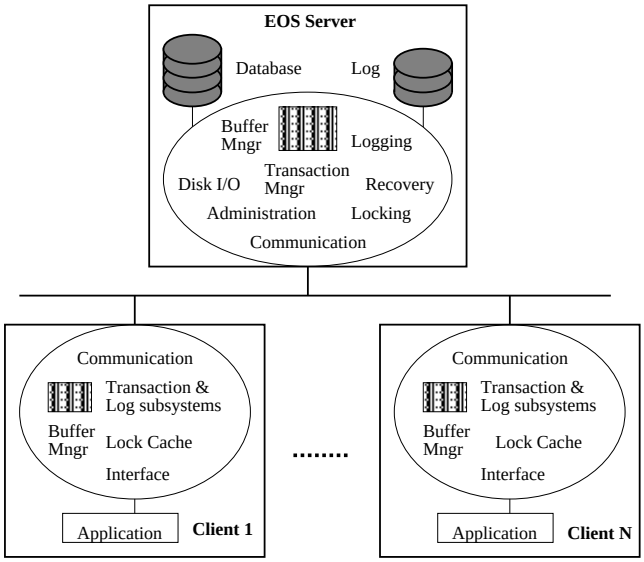

Fig. 1. The EOS client-server architecture

to be extensible. Users may define hook functions to be executed when certain primitive events occur. This allows controlled access to several entry points in the system without compromising modularity. Finally, configuration files, which can be edited by the users to tune the performance and customize EOS, are provided.

2.2 The client-server architecture of EOS

Figure 1 shows 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 is aborted or when a client or server failure occurs. The server is implemented as a multi-threaded demon process that waits passively for a client application to start communication. For each client application there is a thread running on the server which acquires all the locks needed, sends the requested pages to the application, receives log records and updated pages, installs the updates made by a committed transaction, and frees all the resources used by an aborted transaction. The communication between the server and the client workstations is done by using TCP/IP connections over UNIX sockets (Stevens 1990). To avoid blocking I/O operations, the server creates a disk process for each storage area accessed by client applications. The 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 (Kernighan and Pike 1984; Stevens 1990).

Client applications are linked with the EOS client library and perform all data and index manipulation during normal transaction execution. Each application may consist of many transactions but only one transaction at a time is executed. Each application has its own buffer pool for caching the pages requested from the server. For transaction management, each application has 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. A least recently used (LRU) buffer replacement policy is employed for both the client and server buffer pools.

An application starts a transaction by sending a *start transaction* message to the server. When the server receives the message, it assigns a new transaction identifier and creates a new thread if there is no active one serving this application. The application can commit or abort an active transaction by sending a *commit transaction* or *abort transaction* message to the server, respectively. The server can unilaterally abort a transaction when the transaction is involved in a deadlock cycle, or when the transaction exceeds the resources allocated to it, or when an internal error is detected. In this case, an abort message is sent to the transaction the next time the transaction communicates with the server.

The current implementation of EOS does not support inter-transaction caching (Wilkinson and Neimat 1990; Carey et al. 1991; Wang and Rowe 1991; Franklin et al. 1992b) of either pages or locks. Consequently, when a transaction terminates (whether committing or rolling back), all pages present in the application's buffer pool are purged, and the locks acquired by the transaction are released. In addition, no support for distributed transactions is provided. Although applications may connect to many EOS servers and access the databases stored with them, EOS does not provide support for the two-phase commit (2PC) protocol yet.

3 Transaction management in EOS

EOS provides facilities to preserve the atomicity, isolation, and durability database properties while allowing the interleaved execution of multiple concurrent transactions. Transactions in EOS are *serializable* (Bernstein et al. 1987), and recovery is based on logging. In the following sections, we describe in detail the transactional facilities offered by EOS.

3.1 Concurrency control

Like most commercial database management systems and research prototypes, EOS uses locking for serializability. However, EOS takes a semi-optimistic approach to locking by employing the 2V-2PL protocol. The 2V-2PL is also coupled with multi-granularity locking (MG-2V-2PL) to reduce the number of locks a transaction has to acquire during its execution.

3.1.1 Lock modes

EOS supports three locking granularities: page-level, file-level, and database-level. A file is locked by locking the page containing the file object. A database is locked by locking the page containing the root file object mentioned in Sect. 2.1. A page, the smallest lock granule, can be locked by a transaction $\mathcal{T}$ in the following modes.

Intention shared (`IS`): There is a file object on the page and $\mathcal{T}$ intends to read an object belonging to the file.
Shared (`S`): $\mathcal{T}$ wants to read an object stored on the page.

Table 1. The lock compatibility table

Lock compatibility table							
Mode requested	Existing mode						
	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 2. The lock upgrade table

Lock upgrade table							
Mode requested	Existing mode						
	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	X	SIX	-	-
IC	-	-	IC	-	IC	-	-
C	-	-	-	C	-	-	-

Intention exclusive (IX): There is a file object on the page and $\mathscr{T}$ intends to update an object belonging to the file.

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

Exclusive (X): $\mathscr{T}$ wants to update an object stored on the page.

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

Commit (C): $\mathscr{T}$ had an X lock on the page and it is in the process of committing.

When a transaction accesses an object, the page containing the object is locked in the appropriate mode. All locks acquired by a transaction are released when the transaction terminates (by committing or aborting). Lock acquisition is implicit when objects are accessed via the byte-range operations mentioned in Sect. 2.1. However, when a transaction obtains a direct pointer to an object in the page where the object resides, the transaction has to call the storage manager to obtain an exclusive lock on the page before it updates the object for the first time.

When a page is locked, the file containing this page is locked, too, in the corresponding intention mode. In addition, when a file is locked in either S or X mode, the pages the file contains are not locked explicitly, unless the file is locked in S mode and a page in the file is updated. When a transaction wants to commit, it converts its locks to commit locks according to the following rules.

CR1: IS and S locks remain unchanged.
CR2: IX and SIX locks are converted to IC locks.
CR3: X locks are converted to C locks.

Table 1 determines whether a lock request can be granted or not. Each column corresponds to a lock that some transaction can acquire and each row corresponds to a lock requested by some other transaction. A “Y” table entry indicates that the lock request can be granted and a “N” table entry indicates that the request has to be blocked.

Table 2 is used for lock upgrades. A lock upgrade occurs when a transaction holds a lock in some mode and then executes an operation that requires a different mode for the same lock. Each column indicates a lock mode that the transaction holds and each row indicates the lock mode requested by the new operation. A table entry containing “-” corresponds to an erroneous case; either violation of the assumption that no more locks are acquired when the transaction enters its commit phase and starts acquiring IC and C locks, or violation of the three convert rules mentioned above.

Transactions that are in the process of committing their updates are blocked when there are active transactions that read some of the updated pages. EOS enforces this constraint in order to generate serializable schedules. In addition, EOS blocks transactions attempting to read a page that has been updated by a transaction that is in the process of converting its locks. In this way, committing update transactions are not blocked indefinitely.

3.1.2 Deadlock detection

In every lock-based concurrency control algorithm, deadlocks may occur. The 2V-2PL protocol is more susceptible to deadlocks than the strict 2PL locking protocol, for it does not prevent a transaction from reading a page that was updated by another transaction, nor does it prevent a transaction from updating a page that was read by another transaction. Consequently, conflicts that may develop during the conversion of locks to commit locks may result in deadlocks.[1]

A deadlock cycle can be discovered either by using timeouts or by running a deadlock detection algorithm. Although timeouts offer a simple and inexpensive solution, they are very pessimistic and cannot distinguish between deadlocks and long waits. On the other hand, deadlock detection algorithms require extra CPU cycles during the construction of the waits-for graph (WFG). However, this additional CPU demand can be kept fairly low when an incremental approach is used for building the WFG.

In a data-shipping client-server architecture, arbitrary delays may be introduced because of the communication network and the fact that most of the computation is performed at the clients. Consequently, deadlock detection is better suited for this environment than timeouts, and this is the approach EOS follows. EOS performs deadlock detection when a lock request has to be blocked.[2] In particular, EOS performs deadlock detection every time a lock request by some transaction is blocked by another transaction which is waiting for a lock request to be granted.

When a lock request is blocked, EOS visits the list of granted lock requests for the same lock entry and checks whether one of these requests belongs to a transaction that

[1] EOS supports the 2PL protocol and applications may choose between 2V-2PL and 2PL depending on the expected workload. The choice of the locking protocol is done at server start up time and cannot be changed while the server is running

[2] A periodic deadlock detection scheme could also be used

is waiting for a lock to be granted. If this is the case, the deadlock detection algorithm is invoked. Because each lock request structure contains a direct pointer to the transaction control block of the owner transaction, the above check is very efficient.

The EOS deadlock detection algorithm consists of two steps. The first step tries to find deadlock cycles involving transactions that tried to upgrade their lock modes on the same locked page. If no cycle is discovered during the first step, the second step dynamically constructs the WFG and searches for cycles by following a variation of the depth first traversal algorithm (Beeri and Obermarck 1981).

In the current implementation, EOS evicts the transaction whose lock request resulted in the formation of a deadlock cycle. An alternative approach, which we may adopt in the future, is to avoid evicting a transaction that is in the process of committing, unless it is the only choice. A transaction enters its committing phase at the time it starts converting its exclusive locks to commit locks (see Sect. 3.3.3).

3.2 Logging and recovery

One factor that affected the design of the EOS recovery scheme is the way in which applications perform updates. As we mentioned in Sect. 2.1, applications can update an object by either calling a function provided by the EOS client library or by updating the object directly after obtaining a direct pointer to the object in the page where this object resides. The latter approach allows applications to update an object at memory speeds with no extra overhead. However, this approach makes the detection of the portions of the object that have been updated much more difficult than the former approach.

A second challenge we faced is the fact that EOS is designed to handle both traditional and non-traditional database applications, including CAD, CASE, and GIS. While relational database systems usually update individual tuples only once during an update operation and they generate a log record for each individual update, non-traditional database applications typically work on several objects by repeatedly traversing relationships between these objects and updating some of them as well. Consequently, generating a log record for each update would not be efficient, since an object may be updated multiple times by the same application. Grouping multiple updates in one log record is a better solution.

Finally, since EOS is based on a client-server architecture in which updates are performed at each client, recovery is different than the approach followed in centralized database systems, where both the updates and the generation of log records take place at the server. In EOS, log records are generated at client workstations and they are shipped to the server, which maintains the log, during normal transaction execution and at transaction commit. However, clients do not send to the server the updated pages at transaction commit, as done in the Exodus client-server storage manager (Franklin et al. 1992a). Instead, EOS employs the *redo-at-server* approach; the server reads the log records written by committed transactions and applies their updates to the database.

3.2.1 Whole-page redo-only logging

EOS writes to the log entire modified pages instead of individual log records for updated regions of the page. This approach has several advantages. It allows the allocation of larger client buffer pools since no separate space is required for generating log records. The overhead of performing updates is low since the only required action at update time is to mark the page dirty. Whole-page logging may also reduce log space when several objects residing on the same page are updated by the same transaction. This is in part due to the fact that no undo log records are written and in part due to the fact that only one log header is required. Transaction rollback is very efficient since there are no undo log records. Finally, when the redo-at-server approach is used, whole-page logging avoids I/O related to posting committed updates to the database because the server does not have to read a page from disk to update it.

Nevertheless, a main disadvantage of whole-page logging is that it wastes log space when a small part of a page is modified. A better approach may be the page diffing scheme presented by White and DeWitt (1995). However, the page diffing approach may require disk I/O to apply the log records to the database under the redo-at-server scheme employed by EOS; a page that is not cached in the server buffer pool has to be read from secondary storage before the updates present in a log record are applied to it.

EOS allows logging to be done asynchronously during transaction execution so that at commit only a small (tunable) number of log records have to be forced to the log. In particular, when an object is accessed by an application for the first time, EOS allocates a handle for it and returns the handle to the application. The application may release an object handle when the object is not needed anymore. Every time the application communicates with the server, EOS checks whether there is an updated page in the client buffer that is not being accessed by the application – in which case all handles for the objects residing on this page are released. In this case, a log record is generated for this page and it is sent to the server together with the request.[3] Any remaining log records are sent to the server at transaction commit.

A conceptual view of a redo-only log is shown in Fig. 2. The log contains three kinds of records: *checkpoint* records indicating that a checkpoint has been taken; *commit* records indicating that a transaction has committed; *redo* records containing the results of the updates performed by committed transactions. EOS partitions the log into two kinds of logs: a *global* log and a number of *private* logs, as shown in Fig. 3. Each private log is associated with one transaction only and contains all the log records generated by this 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. A 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.

[3] Actually, log records are being generated after a number of dirty pages that are not being accessed by the application is reached. This number corresponds to an EOS configuration parameter

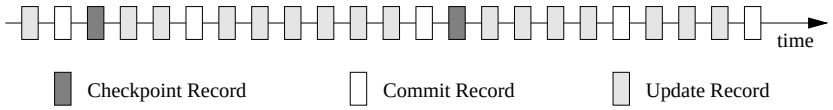

Fig. 2. Conceptual view of a redo-only log

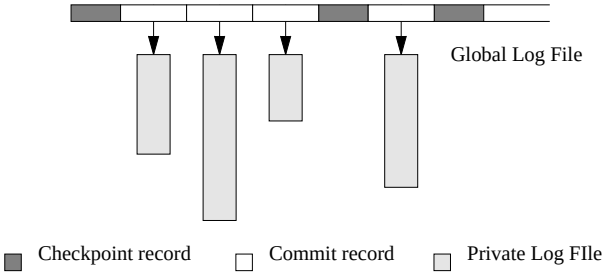

Fig. 3. The physical structure of the EOS log file

The private log could be kept in either the client or the server machine. In the first approach, we would have to move the log records of committed transactions from the client to the server. Allowing those records to reside on the client machine is undesirable because client machines can connect to and disconnect from the network at any time. Thus, to guarantee that the restart procedure will not skip any committed updates, the update log records must be moved to the server machine as part of the transaction commit protocol. We abandoned this approach because of the double copying involved (first to client local disk and then to server's log), and all private logs are stored on the server machine.

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. The LSN of a log record corresponds to the address of the record in the private log file.

3.2.2 Large object recovery

Similar to small objects, large objects must be recoverable. However, the recovery mechanism must not impose a high overhead on large object operations. The large object recovery mechanism depends on the way large object data segments operations are carried out. We examined two methods for handling large object data segment operations; namely, buffering and direct disk I/O. We chose the latter technique because buffering complicates buffer management and requires substantial amounts of contiguous buffer space for objects whose size may be up to hundreds of megabytes and even gigabytes (e.g., an MPEG-compressed 100-min movie requires about 1.125 GB of storage).

We examined two main alternatives for recovery of large objects: logging and shadowing. Redo-only logging is employed by EOS for small object recovery. However, the redo-only logging approach cannot be used for large object recovery since large objects are written directly to disk. Consequently, undo log records must be written to recover the state of a large object that is updated by an aborted transaction. These undo log records require extra disk I/O for reading the previous image of the affected byte-range. Shadowing is a recovery technique in which updates are never performed on the current page contents. Instead, new pages are allocated and written, while the pages whose contents are being updated are retained as shadow copies until the transaction that performed the updates commits. In case of transaction rollback, the shadow pages are used to restore the updates performed by the aborted transaction.

We chose the shadow technique for recovery of large objects in EOS because it is simple and efficient. Shadowing does not require the generation of undo log records and, consequently, the redo-only logging scheme employed by EOS for small object recovery does not require any modifications. In addition, shadowing causes fewer I/O operations during transaction rollback and system restart. During transaction rollback, no I/O is required for restoring the state of a large object since the original state of the object is present in the database. During system restart, no I/O is required for redoing committed large object updates because all large object updates are applied directly to the database during normal transaction execution.

However, shadowing does not provide protection against media failures and some additional measures are necessary. One solution is to store the database in two disks and propagate each update to both disks. Another solution is to use logging and log all large object updates. Currently, EOS follows the second approach because duplexed disks are costly. Before the EOS server writes to disk a large object, it generates a redo-only log record that is inserted in the in-memory log buffer. Unlike the traditional WAL protocol, shadowing does not require the forcing of the in-memory log buffer to disk. The log buffer, however, is forced to disk when it overflows and at transaction commit.

It is worth mentioning that logging of large objects for protection against media failures may degrade the performance of the system. This is especially true when large objects are updated frequently and the sizes of the updated byte-ranges are large. For this reason, EOS allows logging for large objects to be turned off when media recovery is not an issue.

3.3 Transaction execution

A committed transaction $\mathcal{T}$ goes through three phases, active, committing, and write, which are shown in Fig. 4. $\mathcal{T}$ is in the *active phase* from the time it starts up until the time it finishes normal execution and it is ready to commit. At this point, $\mathcal{T}$ 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, $\mathcal{T}$ is in the *committing phase*. $\mathcal{T}$ is said to be committed when the log records are written to stable storage and a commit log record is placed in the global log. The last phase is the one where $\mathcal{T}$ establishes its updates in the server buffer pool; this phase is called *write phase*. A transaction can be aborted at any time

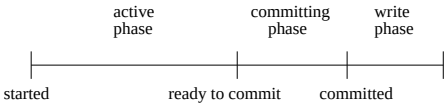

Fig. 4. The states a transaction can be in during its execution

during the active and the committing phases. Once the transaction has reached the write phase, its updates are guaranteed to persist.

3.3.1 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 used in EOS, a dirty page that has to be replaced should never be written to its disk location in the database before the transaction commits. This is because no undo information is kept in the log and, thus, the updates could not be undone. Avoiding writing an uncommitted dirty page to the database can be achieved in several ways.

A first alternative is to send uncommitted dirty pages to the server. The server can place these pages either in a shadow disk location or in its buffer pool. In the latter case, shadow pages are written when the uncommitted modified pages are replaced from the buffer pool. The shadow pages replace the originals when the transaction commits and they are discarded when the transaction aborts – this approach is used in O_2 (Deux et al. 1991).

A second alternative is to store uncommitted dirty pages in a swap space. When a dirty page is replaced from the application's private pool, it is placed in the swap space and the buffer manager stores the location of the page in the buffer control block (BMCB) associated with the page. If the page is accessed again by the same transaction, the buffer manager fetches the page in the buffer pool by reading it from the swap space. A similar approach was proposed by Franklin et al. (1993) to increase the effectiveness of inter-transaction caching.

We have chosen the second approach because under 2V-2PL an S lock does not conflict with an X lock and, consequently, a dirty page present in the server buffer pool could be read by any transaction. Although we could have prevented this by devising special lock modes, we abandoned this solution in favor of a simpler lock manager. Hence, EOS stores the swapped-out dirty pages in a swap storage area.

The swap storage area can be on the client or the server machine, and it is specified in the EOS configuration files. 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 (Biliris and Panagos 1993). When a dirty page is replaced from the application's private pool, EOS generates a log record containing the page and the LSN of the log record is stored in the BMCB of this page. The following paragraph describes the data structures and algorithms used.

Every page $\mathscr{P}$ in the private cache of a transaction $\mathscr{T}$ has a BMCB associated with it, as well as a buffer frame where it is stored. The BMCB keeps various information related to the page, e.g., the lock held by $\mathscr{T}$, the relative order of $\mathscr{P}$ in the least recently used page list, whether $\mathscr{P}$ has been updated, etc. When a transaction wants to access an object on the page, it calls the buffer manager and passes along the lock mode $\mathscr{L}$ that needs to be acquired on $\mathscr{P}$. The buffer manager executes the following algorithm.

1. Scan the buffer pool to locate the BMCB of $\mathscr{P}$. Depending on whether the BMCB is found, execute one of the following steps.
2. The BMCB of $\mathscr{P}$ is not found.
   a) Create a new BMCB for $\mathscr{P}$.
   b) Allocate a frame in the buffer pool to place $\mathscr{P}$. If the buffer pool is full, replace a page from the pool as follows:
      i. Find the least recently used page $\mathscr{P}_{LRU}$.
      ii. If $\mathscr{P}_{LRU}$ has not been updated, free the frame and the BMCB associated with it.
      iii. If $\mathscr{P}_{LRU}$ is dirty, send $\mathscr{P}_{LRU}$ to the private log file, save the returned LSN in the BMCB of $\mathscr{P}_{LRU}$, and free its frame.
   c) Request from the server page $\mathscr{P}$ with lock $\mathscr{L}$ on $\mathscr{P}$ and $I\mathscr{L}$ on $\mathscr{P}$'s file. Return.
3. The BMCB of $\mathscr{P}$ is found.
   a) If $\mathscr{P}$ is present in the buffer pool and no lock upgrade is needed. Return.
   b) If $\mathscr{P}$ is present in the buffer pool and the lock mode needs to be upgraded, request from the server $\mathscr{L}$-lock on $\mathscr{P}$ and $I\mathscr{L}$-lock on $\mathscr{P}$'s file. Return.
   c) If $\mathscr{P}$ was swapped out, make room in the buffer pool by replacing a page as in step 2b above and request $\mathscr{P}$ from the private log using the LSN stored in $\mathscr{P}$'s BMCB. Return.

3.3.2 Transaction abort

When a transaction $\mathscr{T}$ 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 particular, $\mathscr{T}$ sends an *abort transaction* message to the server, frees various control structures used, and purges the local cache.

When the server receives an *abort transaction* message, it releases all locks held by $\mathscr{T}$, purges the private log file associated with $\mathscr{T}$, and adds $\mathscr{T}$ to the list of aborted transactions.

3.3.3 Transaction commit

When a transaction $\mathscr{T}$ 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 $\mathscr{T}$ is executing step 2 or waiting on step 3, the server may reply with an *abort* message. The reason for the abort may be: (1) $\mathscr{T}$ was involved in a deadlock that

materialized when the server was acquiring the commit locks for $\mathcal{T}$, or (2) an internal error occurred while writing the log records or flushing the private log.

On receiving the *convert locks* message the server executes the following steps.

1. For each page locked by $\mathcal{T}$ do one of the following:
   a) If the lock mode is IX or SIX, upgrade it to IC.
   b) If the lock mode is X, upgrade it to C.
2. Release all IS and S locks.
3. Send a *success* message to the application.

An application sends a *commit transaction* message to the server after it sends all log records generated by the committing transaction to the server. When the server receives the *commit transaction* 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. **Write Phase:** Install the updates performed by the committed transaction by scanning the transaction's private log file. For each log record corresponding to the after image of a page do the following:
   a) If the page is present in the server buffer pool, overwrite its contents with the data part of the log record.
   b) If the page is not present in the server buffer pool, make room in the buffer pool and place the data part of the log record there.
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.

4 Recovery from process crashes

When a client application crashes, the server aborts the transaction associated with this application, if any, and the server thread bound to the application is terminated. When the server process crashes, the server restart procedure returns the database to a state that includes all the updates made by committed transactions before the failure. To reduce the amount of work the recovery manager has to do during system restart, the EOS server periodically takes checkpoints.

4.1 Checkpoints

During normal transaction execution, the server buffer pool contains two kinds of pages: *clean* and *dirty*. A page is clean when the disk version of the page is the same as the version of the page that is present in the server buffer pool. A page is considered dirty when its contents are not the same as the disk version of the page. Since EOS employs a redo-only recovery protocol, dirty pages contain only committed updates. The server's buffer manager tracks in the BMCB of each dirty page the location of the commit log record, referred to as *RedoLSN*, that belongs to the last committed transaction that updated this page.

EOS employs a *fuzzy checkpoint* algorithm that takes checkpoints asynchronously, while other processing is going on. The EOS checkpoint algorithm does not force any dirty pages to disk. Instead, EOS has a background process that forces dirty pages to disk on a continuous basis. Formally, the steps followed by the EOS checkpoint algorithm are the following.

1. Compute the address, referred to as $CommitLSN$, of the earliest commit record inserted in the global log by transactions that have started their write phase. If there are no transactions that have started their write phase, set $CommitLSN$ to be the current end of the global log.
2. Compute the minimum, referred to as $DirtyLSN$, of all $RedoLSN$ values present in the BMCBs of the dirty pages that are present in the server buffer pool. If the server buffer pool does not contain any dirty page, then set $DirtyLSN$ to be the current end of the global log.
3. Set $RestartLoc$ to be the minimum of the $DirtyLSN$ and $CommitLSN$ values computed above.
4. Write in the global log a checkpoint record that contains the $RestartLoc$ computed in the previous step.
5. Save the location of the checkpoint record in a place well known to the restart procedure.

Note that the checkpoint algorithm visits the list of the transactions that are in the write phase first and then it computes the minimum of the $RedoLSN$ values corresponding to dirty pages. This is necessary in order to guarantee correct restart recovery in the case where the server buffer pool does not contain any dirty pages when the checkpointing process starts and a transaction T finishes its write phase before the list of transactions that are in the write phase is examined. In this case, if the server were to crash after the checkpoint was taken, the updates made by T would not have been redone during restart.

4.2 Restart

System restart is done by scanning the global log file and redoing all the updates made by committed transactions in exactly the same order as they were originally performed. After the database state is restored, a checkpoint is taken and the system is operational again. If the server process crashes in the middle of the restart procedure, the next restart repeats the same steps again in an idempotent fashion. Formally, the steps followed during restart recovery are the following.

1. Get the $RestartLoc$ value stored in the last checkpoint record of the global log. For each log record inserted in the global log after the $RestartLoc$ do the following.
   a) If it is a checkpoint record, skip it.
   b) If it is a commit record, redo all updates present in the private log corresponding to that record by executing the write phase of the server's transaction commit algorithm presented in Sect. 3.3.3.
2. Take a checkpoint.

5 Performance results

In this section we present the results collected from three sets of experiments. The first set of experiments measures the

Table 3. Database (DB) configuration

DB name	Objects in DB	Object size (B)	Objects per page	Pages in DB
FewObj	6000	500	6	1000
MediumObj	30 000	100	30	1000
ManyObj	100 000	20	100	1000

overhead of the logging and recovery components of EOS when only one transaction is active in the system. The second set of experiments concentrates on a multi-client setting and measures the overhead of logging during normal transaction processing. The third set of experiments compares the performance of the 2V-2PL and 2PL locking algorithms.

All experiments presented in this section were run on SPARCstation 10s running SunOS 4.1.3 and having 32 MB of main memory and 142 MB of swap space. The clients 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 and the database page size was 4 KB. The log was stored in a regular UNIX file and `fsync()` was used at transaction commit for flushing any internal operating system buffers to disk. All times reported were obtained by using the `gettimeofday()` and `getrusage()` UNIX system calls.

5.1 Logging, abort, and system restart

In this section we present an initial study which measures the logging overhead, the time required to abort a transaction, and the time spent when restarting the system after a crash.

5.1.1 Database and system model

Table 3 describes the three databases used for the experiments we ran; this is a variation of the model presented by Franklin et al. (1992a) for measuring the performance of the recovery components of client-server systems. Each database consists of 1000 pages and the key difference among them is the number of objects they contain. The first database, called *FewObj*, consists of 6000 objects of size 500 B each. The second database, called *MediumObj*, consists of 30,000 objects of size 100 B each. The third database, called *ManyObj*, contains 100 000 objects of size 20 B each. Space on pages was purposely left unused so that the total number of pages is the same for all three databases.

We used only one kind of transaction for the experiments performed, referred to as *Update*, that sequentially scans the entire database and overwrites part of each encountered object. The server's cache was set to 16 MB 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 set to 8 MB. In this way the entire database fits in the private pool during transaction processing. However, 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.

Table 4. Performance of logging, transaction abort, and system restart

Database name	Execution time (s)		Logging overhead	Abort time (s)	Restart time (s)
	Logging on	Logging off			
FewObj	17.2	15.7	9%	1.0	1.3
MediumObj	18.9	17.3	9%	1.1	1.3
ManyObj	25.6	23.8	7%	1.7	1.3

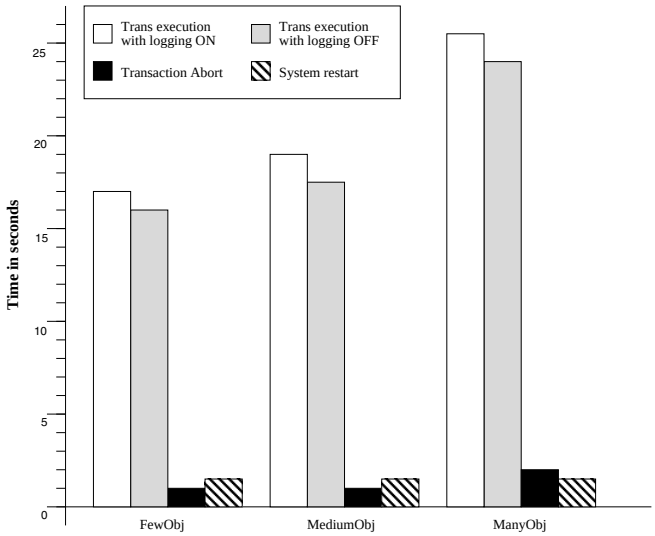

Fig. 5. EOS performance results

5.1.2 Results

In the first set of experiments we measured the overhead of writing the log records to the log, 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. The execution time for a transaction includes the time to initialize all EOS internal structures, to execute and to commit the transaction. If logging is on, the execution time also includes the generation, shipping and writing of log records, and the forcing of 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 Sect. 3.2.1, EOS employs whole-page logging. Since 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 because the processing time of the application program increases.

The time to abort a transaction was measured by the same set of experiments that were run to measure the effect 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. The abort tests shown in Fig. 5 indicate a slight increase in the time needed to abort a transaction as the number of updated objects increases. The time to abort a transaction corresponds to the time needed to release all the locks held by the transaction plus the time needed to clean all data structures used by the transaction. The release of locks takes the same time for all three databases since page-level locking

Table 5. Database configuration

Database name	Number of modules	Objects per module	Module size MB	Total size MB
Small	5	10 000	1.24	11.15
Large	5	100 000	6.20	55.75

is used by EOS and the same number of pages is accessed. Thus, the increase in the abort time is due to the the cleaning of the object-related data structures - one handle per accessed object.

To measure the time needed to redo the updates performed by a committed transaction, we turned off the checkpoint activity and the server process was crashed immediately after the transaction committed. During restart, the entire log was scanned and the updates made by the committed transaction were re-done. The restart tests showed that the time needed to redo the committed updates is independent of 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 4 indicate that EOS offers fast system restart compared to normal processing time.

5.2 The performance of the whole-page redo-only logging

In this section, we present an initial study of the performance of the whole-page redo-only logging algorithm employed by EOS. In this study, we used two databases of different sizes and two different operation sets in order to measure transaction response time and system throughput when several clients are active in the system.

5.2.1 Database and system model

A modified version of the OO1 benchmark (Cattell and Skeen 1992) was used as the basis for measuring the performance of the EOS logging component. We used two different database sizes in the study, referred to as *small* and *large*. Table 5 shows the size characteristics of the two databases. Each database consists of five modules. Each module contains several part objects, each having size equal to 100 B. Each part object is connected to exactly three other objects. In order to be able to traverse all objects, one connection is initially added to each object to connect the objects in a ring; the other two connections are added at random. Furthermore, one of the part objects serves as the root of the object hierarchy and it is given a name so that it can be retrieved later on from the database.

The experiments were performed using two different traversal operations, referred to as *update one* and *update all*. Both traversals retrieve the root part object of a module and they visit all part objects in the module. While the update all traversal updates all part objects it scans, the update one traversal updates only the very first part object. In order to avoid performance degradation due to lock conflicts and deadlocks, each client accesses a different module in the database. Thus, the number of clients in all experiments was varied from 1 to 5. During each experiment, each traversal was run as a separate transaction, which was run repeatedly so that the steady-state performance of the system could be observed. While the server pool was not flushed between transactions, the client pool is empty at the beginning of each transaction because EOS does not support inter-transaction caching in the current implementation. Each client was given 8 MB of buffer space and the server's cache was set to 16 MB.

Table 6. Small database performance results

Active clients	Update one		Update all	
	Response (s)	Throughput (trans/min)	Response (s)	Throughput (trans/min)
1	3.32	18.07	11.56	05.19
2	3.83	31.33	11.89	10.09
3	4.71	38.22	12.86	14.00
4	5.83	41.17	14.53	16.52
5	7.28	41.21	17.03	17.62

The following two sections present the collected results. First we present the results for the small database and then we analyze the results for the large database.

5.2.2 Small database

This section presents the performance results collected for the two traversal operations using the small database. Both traversal operations do not experience any paging because each individual module is smaller than the client buffer pool and all five modules are smaller than the server's cache. Table 6 contains the collected results. Figure 6 shows the response time and throughput versus the number of active clients for both traversal operations.

Each update one traversal produces only one log page since only one part object is updated and each part object is much smaller than the database page size. On the other hand, each update all traversal produces the same number of log pages as the number of pages updated, namely 278. However, Table 6 shows that the difference in response time between the two traversal operations varies from 2.3-fold to 3.4-fold. This is because EOS writes log records asynchronously, during normal transaction execution. As a result, at transaction commit time only a small number of log pages have to be sent to the server and written to the log.

Interestingly, the response time of the update one traversal increases faster than the response time of the update all traversal as the number of active clients increases. For example, when the number of clients increases from one to two the response time of the update one traversal increases by 15%, while the response time of the update all traversal increases by 2%. This is because whole-page logging is not very effective when only a small region of a page is updated. Consequently, the overhead of writing log records to private log files and the overhead of reading these records during the transaction write phase affect more the update one traversal. Log writes (reads) are more expensive because the amount of data written to (read from) each private log file is small, and the cost of disk seeks is not amortized over the amount of data written to (read from) disk.

However, when a large number of log records is generated, the buffering of log records in main memory reduces the cost of log writes because disk seeks are amortized over

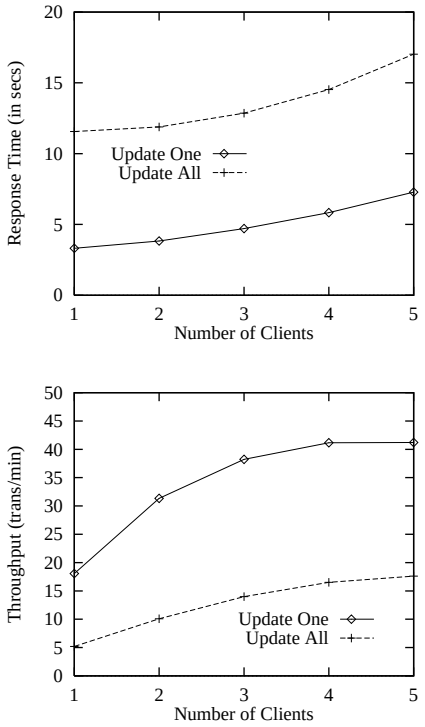

Fig. 6. Small database results, response (**a**) and throughput (**b**)

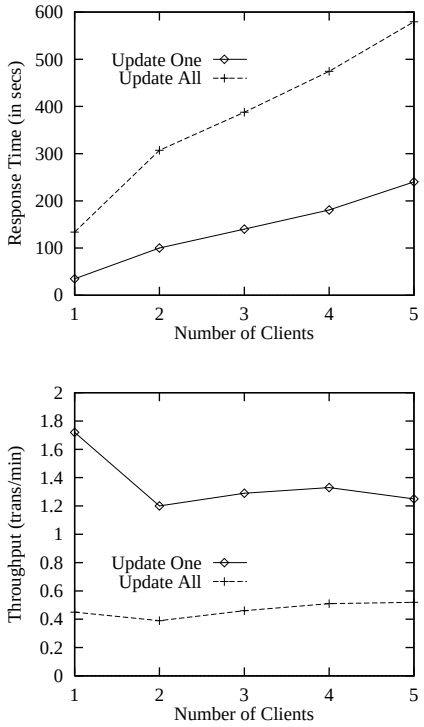

Fig. 7. Large database results, response (**a**) and throughput (**b**)

Table 7. Large database performance results

Active clients	Update one		Update all	
	Response (s)	Throughput trans/min	Response (s)	Throughput trans/min
1	034.92	1.72	133.79	0.45
2	100.03	1.20	307.24	0.39
3	140.05	1.29	387.67	0.46
4	180.68	1.33	474.50	0.51
5	240.17	1.25	579.34	0.52

time. Furthermore, because EOS prefetches log records during the transaction write phase, log reads are also less expensive.

The throughput results shown in Fig. 6 are calculated from the response time results. While the throughput of the system almost doubles when the number of active clients increases from one to two, when more than two clients are active in the system the throughput increases with a much slower rate. In particular, when the number of clients increases from four to five, the throughput increases by 6% for the update all traversal. The increase for the update one traversal is marginal.

5.2.3 Large database

This section presents the performance results collected for experiments using the large database. Table 7 shows the collected results numerically and Fig. 7 illustrates the response time and throughput versus the number of active clients for both traversal operations.

As in the small database case, the asynchronous sending and writing of the log records during normal transaction processing keeps the difference in response time between the two traversal operations between 2.4 and 3.8 times, despite the fact that the update all traversal generates 2778 log pages. However, transaction response time increases dramatically when two clients are active in the system. In particular, the update one traversal becomes 2.8 times slower and the update all traversal becomes 2.3 times slower. This is because there is a significant amount of paging going on in the server's cache. Since each module consists of 2778 pages and the server's cache is set to 4000 pages, when two or more clients are active simultaneously, the server needs to perform cache replacement. After a number of transactions are committed, the server's cache consists of pages that contain committed updates and each cache replacement operation requires a disk I/O.

Although the throughput of the system drops when two clients are active, for three and four clients the throughput increases for both traversals. However, after four clients the server becomes a bottleneck and the update one traversal does not scale anymore. The server bottleneck is due to the overhead of the random seeks to the log disk during the writing and reading of log records. This is because each transaction private log file corresponds to a UNIX file in the current EOS implementation.

5.3 The performance of the 2V-2PL algorithm

This section presents a comparative performance evaluation of the 2PL and the 2V-2PL protocols in the context of a client-server environment such as EOS. For each protocol, we measure the response times of transactions that read a number of database objects which are updated concurrently by another transaction, as well as the response time of the updater transaction.

Table 8. Performance results for the reader and writer transactions

Number of clients			Response time (s)			
			2PL		2V-2PL	
Total	*Readers*	*Writers*	*Reader*	*Writer*	*Reader*	*Writer*
1	1	0	2.86	n/a	2.86	n/a
1	0	1	n/a	11.56	n/a	11.79
2	1	1	14.01	14.00	3.31	14.07
3	2	1	15.28	15.26	4.68	16.53
4	3	1	17.11	17.20	6.81	19.89
5	4	1	19.18	19.06	9.25	23.22

The small database described in Sect. 5.2 was used as the basis for comparing the performance of the 2V-2PL algorithm against the performance of the 2PL algorithm. The experiments were performed using two different transactions, referred to as *writer* and *reader*, that access exactly the same database module. The writer corresponds to the *update all* traversal operation studied in Sect. 5.2. The reader, on the other hand, just visits all part objects in a module and looks up two fields of each part object, without performing any updates at all.

Because EOS supports both locking algorithms, no change had to be made to the system for collecting the performance results. To avoid costs related to cache replacement, each transaction was given 8 MB of buffer space and the server's cache was set to 16 MB. During each experiment, the reader and writer transactions were run repeatedly at each client so that the steady-state performance of the system could be observed.

5.3.1 Results

We first present results obtained when only one client is interacting with the server. The goalof this experiment was to observe the response time of a transaction (including the overhead of the locking protocol) when there are no lock conflicts in the system. The first row of Table 8 shows the response time of the reader transaction when no other client is active in the system. As we can see, the response time of the reader transaction does not depend on the particular locking protocol employed.

The second row of Table 8 shows the response time of the writer transaction when no other client interacts with the server. For the writer transaction, 2V-2PL is somewhat slower compared to 2PL. This is because the 2V-2PL protocol requires the client to send one extra message to the server to convert its exclusive locks to commit locks (this message corresponds to the first step of the algorithm presented in Sect. 3.3.3). In addition, while log records are being sent to the server, the client checks whether the server replied with an abort message so that to avoid sending unnecessarily the remaining log records – recall that a transaction may be aborted while it is in the process of converting its locks.

We now turn our attention to the case where many clients interact with the server and lock conflicts materialize. Under this experiment, there is always one writer in the system and the number of reader transactions varies from one to four – so the total number of clients varies from two to five. Table 8 presents the results and Fig. 8 illustrates them.

For the 2PL protocol, the average response time of the readers and the response time of the writer are almost the same, regardless of the number of readers that are present in the system. This behavior was expected since under 2PL shared and exclusive locks are not compatible. Thus, a reader is made to wait when it needs to access a page that is locked by the writer and, similarly, the writer has to wait until all the readers that are accessing a page finish execution before it can update any objects residing on that page. Consequently, the aggregate execution and blocking times for both the readers and the writer are similar.

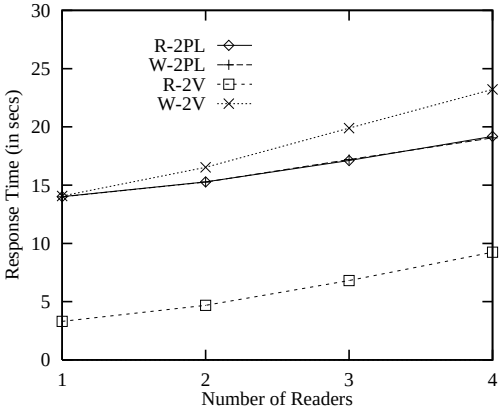

Fig. 8. Reader and writer response times: 2V-2PL vs 2PL

In contrast, when the 2V-2PL protocol is in use, the average response time of the readers is much lower than the response time of both the writer and the reader transactions under the 2PL protocol. Since the 2V-2PL protocol allows readers to access objects that are being updated by the writer, readers are not blocked. However, the average response time of the reader transactions increases as the number of readers in the system increases. For example, the average reader under 2V-2PL is about twice as fast as the reader under 2PL when four readers are interacting with the server; when only one reader is active, the 2V-2PL reader is 4.2 times faster that the 2PL reader. This is because a reader may be blocked when the writer is in the process of converting its locks to commit locks. As the number of readers in the system increases, the probability of a reader being blocked during the writer's lock conversion process also increases.

Interestingly, the 2V-2PL writer is always slower than the 2PL writer, due to a number of reasons. First, an extra message and a check are required by the 2V-2PL protocol, as mentioned in the beginning of this section. Second, since each transaction is executed repeatedly by each client, there is always at least one active reader when the writer starts converting its locks and, thus, the writer is blocked. However, the server thread that is responsible for this transaction does not wait until the lock conversion is over. Instead, it checks whether the client sent any log records, since log records may be sent while the lock conversion is in progress. Finally, the conversion of locks to commit locks involves a traversal of all lock entries belonging to the writer transaction. For each lock entry, a number of expensive UNIX semaphore operations are required for converting the lock.

6 Related work

There has been a considerable amount of research and experimental work on client-server object stores. There are also several commercial systems that are based on the client-server model but few details about the specifics have been published.

O_2 (Deux et al. 1991) employs an ARIES-based (Mohan et al. 1992) recovery protocol using shadowing to offer a redo-only logging scheme. The log is maintained by the server and only after-images of updates are logged. When a page containing uncommitted updates is swapped out of the client cache, a log record is generated first and then the page is sent to the server. The page is placed in the server buffer pool unless no space is available, in which case a shadow page is created. Placing a dirty page in the server buffer pool is not a problem in O_2 because a S lock conflicts with an X lock (standard 2PL). Unlike O_2, EOS cannot place dirty pages in the server buffer pool because under 2V-2PL a S lock is compatible with an X lock. Alternatively, EOS logs the entire page and it does not have to keep track of the shadow pages as O_2 does.

The Exodus client-server storage system ESM-CS (Franklin et al. 1992a) employs an ARIES-based recovery approach, modified to work in a client-server environment, and uses the steal and no-force buffer management policies. Concurrency control is based on the strict 2PL algorithm, and the minimum locking granularity is a database page. In contrast to EOS, ESM-CS applications send all pages modified by a transaction to the server at transaction commit. Aborting a transaction in ESM-CS requires the following steps: (1) scanning of the log to locate the log records written by the transaction, (2) undoing the updates present in each log record, and (3) generating compensation log records. On the other hand, EOS does not have to scan the log, nor does it have to undo any updates.

ESM-CS requires three passes over the log during start-up time; log records are written during undo, and dummy log records are written in order for the conditional undo to work correctly. EOS requires only one pass over the log and it does not generate any log records during restart. Unlike EOS, ESM-CS buffers large objects and uses logging for their recovery. In addition, large object recovery is handled a page at a time, and the pages used to store a large object contain an LSN-like field. When a byte-range crossing page boundaries is requested, pages have to be fetched individually and stripped from their headers before being presented to the application.

ARIES/CSA (Mohan and Narang 1994) is similar to the ESM-CS architecture. ARIES/CSA follows the traditional client-server recovery paradigm where clients send all their log records to the server as part of the commit processing. Unlike EOS and ESM-CS, transaction rollback is performed by the clients in ARIES/CSA. Similar to EOS, ARIES/CSA clients do not send modified pages to the server at transaction commit. Although ARIES/CSA employs a fine-granularity locking protocol, clients are not allowed to update the same page simultaneously. Instead, a client has to obtain an update token before updating a page as well as a copy of the page from the previous owner of the update token, as described in the algorithms presented by Mohan and Narang (1991). This results in an increased volume of messages and it is expensive. ARIES/CSA allow clients to take checkpoints. Client checkpoints are stored in the global log file maintained by the server. In contrast to EOS and ESM-CS, server checkpointing in ARIES/CSA requires synchronous communication with all the connected clients.

ObjectStore (Lamb et al. 1991) is a commercial OODBMS based on a memory-mapped architecture. The strict two-phase page-level locking is used together with multi-version concurrency control. Similar to EOS, ObjectStore employs a whole-page logging scheme and it uses the log for storing dirty pages belonging to active transactions. However, due to the memory-mapped architecture, ObjectStore tends to generate all log records during transaction commit. In contrast to EOS, ObjectStore offers inter-transaction caching, nested transactions, full dumps, and continuous log archiving.

ORION-1SX (Garza and Kim 1988; Kim et al. 1990) 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 uses an undo-only recovery protocol. Pages updated by active transactions can be written in place on disk during transaction execution. However, all pages updated by a transaction are forced to disk at transaction commit. As a consequence, the performance of the system degrades, since pages that are updated frequently are forced to disk very often.

POSTGRES (Stonebraker and Kemnitz 1991) follows a different recovery scheme than most existing systems assuming that stable memory is available. POSTGRES does not use the write-ahead logging approach, and updates always create new versions. This approach offers a fast recovery with no logging overhead and supports time travel. However, POSTGRES requires special hardware and a separate process to store old versions in the historical database maintained by the system. The same recovery technique is followed by MNEME (Moss 1990) but without making use of stable memory.

QuickStore (White and DeWitt 1994) is a memory-mapped storage system for persistent C++ implemented on top of ESM-CS. Concurrency control uses the page-level strict 2PL protocol, and recovery uses the ESM-CS ARIES redo-undo protocol. White and DeWitt presented a study of several recovery protocols (1995), including a redo-at-server scheme and whole-page logging. Similar to EOS, the redo-at-server studied by White and DeWitt (1995) sends only log records to the server and not dirty pages. Unlike EOS, these log records contain redo and undo information and they are generated using the page diffing scheme. The drawback of this approach is that the server may have to read a page from disk before applying a log record. Consequently, performance degrades when the number of clients increases and the volume of log records generated per client is high.

Although the EOS whole-page logging scheme and the whole-page logging scheme presented by White and DeWitt (1995) are similar, they differ in several ways. Unlike EOS, White and Dewitt's system places uncommitted dirty pages in the server buffer pool. In contrast to O_2, these pages are discarded when they have to be replaced before the transaction that updated them commits. Placing uncommitted dirty pages in the server buffer pool may affect system perfor-

mance, especially when the server buffer pool experiences paging, because the buffer hit ratio becomes lower. Another difference between the two schemes is that EOS installs the updates of a committed transaction in the server buffer pool immediately after the transaction is declared committed. On the other hand, White and DeWitt (1995) use a background thread which periodically reads committed updates from the log and installs them in the server buffer pool.

Carey et al. (1991) studied the optimistic 2PL (O2PL) protocol was studied as part of a performance analysis of several inter-transaction caching protocols for client-server systems. Although 2V-2PL is similar to O2PL, the two protocols differ in several ways. First, transactions running under O2PL do not request any locks from the server until they are ready to commit. Consequently, several transactions may be updating different copies of the same database page concurrently and, hence, a high number of deadlocks is possible. On the other hand, 2V-2PL prevents concurrent updates on the same database page by allowing only one transaction to acquire an exclusive lock on the page at a time. Second, all deadlock cycles under O2PL are discovered only at transaction commit. 2V-2PL discovers deadlock cycles when they are formed, and it resolves them at that time. Finally, a family of O2PL protocols supporting inter-transaction caching was presented in the work of Carey et al. (1991). EOS does not support inter-transaction caching at its current implementation. However, inter-transaction caching of data can be supported in 2V-2PL by using the "check on access" scheme employed by the 2PL scheme presented by Carey et al. (1991).

Ephemeral logging is a new logging approach that has been described and evaluated by Keen and Dally (1993). The description presented by Keen and Dally (1993) is based on a redo-only recovery protocol which does not require checkpointing of the database and does not abort lengthy transactions that use a lot of log space. However, ephemeral logging relies on large quantities of main memory capable of storing both the original and the updated values for all objects that have updated by an active transaction. Consequently, transactions that perform a large number of updates may not be able to run because the available memory is not big enough. EOS does not have this drawback. In addition, ephemeral logging requires higher bandwidth for logging because it uses a number of disk log files, referred to as generations, and a given log record may be written to a number of them. Finally, the authors do not discuss how this new logging scheme would be used in a client-server environment.

7 Conclusions

In this paper, we have described the client-server architecture of EOS and have presented the transaction management facilities provided by EOS. The concurrency control protocol was chosen with the goal of increasing the concurrency level of the system when a given database page is being accessed by several transactions, one of which is going to update some objects residing on the page. The recovery protocol was designed with the goal of minimizing the overhead of the recovery-related activities during normal transaction execution and the goal of offering fast transaction abort and system restart times.

In addition, we have presented several performance studies of the EOS implementation of the concurrency control and recovery protocols that we have described in the paper. From the results collected from these studies and from the limited number of published performance results for logging and recovery systems, we concluded that the overhead for many cases was reasonable, despite the write-intensive nature of the tests we ran. In addition, the performance study of the 2V-2PL concurrency control method showed that the concurrency level of the system increases considerably compared with the performance of the strict 2PL. Finally, the studies raised several issues that have to be addressed in order to improve the performance of the system, including: substituting the expensive UNIX semaphores with a faster test-and-set utility, eliminating the overhead during the write phase by batching reads from the transaction private log files so that the number of random seeks is reduced, and batching log writes to transaction private log files to avoid I/O overhead due to random seeks.

EOS is used as the storage engine of the ODE OODBMS (Biliris et al. 1993). The EOS facilities are also being used in a major AT&T project that provides interactive TV capabilities. This project requires efficient manipulation of multimedia objects in a client-server environment. EOS runs under UNIX on SPARCstation, Solaris, IBM RS/6000, and SGI architectures. EOS is written in C++ but it can also be used by programs written in C. Release 2.0 (Biliris and Panagos 1993) is available free of charge to universities.

We are currently working on issues related to inter-transaction caching, providing support for multiple servers and distributed transaction as well as media recovery.

Acknowledgements. We would like to thank the reviewers for helping us improve both the presentation and the technical depth of the paper. We are also grateful to the EOS users for their helpful feedback.

References

Beeri C, Obermarck R (1981) A resource class independent deadlock detection algorithm. In: Proceedings of the Seventh International Conference on Very Large Databases, Cannes, France, pp 166–178

Bernstein PA, Hadzilacos V, Goodman N (1987) Concurrency control and recovery in database systems. Addison-Wesley, Reading, Mass

Biliris A (1992a) An efficient database storage structure for large dynamic objects. In: Proceedings of the Eighth International Conference on Data Engineering, Tempe, Ariz, February pp 301–308

Biliris A (1992b) 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, Calif, May, pp 276–285

Biliris A, Panagos E (1993) EOS user's guide, rel 2.0. Technical report. AT&T Bell Laboratories, Murray Hill, NJ, USA

Biliris A, Gehani N, Lieuwen D, Panagos E, Roycraft T (1993) Ode 2.0 user's manual. Technical report. AT&T Bell Laboratories, Murray HIll, NJ, USA

Carey MJ, Franklin M, Livny M, Shekita E (1991) Data caching tradeoffs in client-server DBMS architectures. In: Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data, Denver, Colo, May, pp 357–366

Cattell RGG, Skeen J (1992) Object operations benchmark. ACM Trans Database Syst 17: 1–31

DeWitt DJ, Maier D, Futtersack P, Velez F (1990) 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, Australia, August, pp 107–121

Deux O (1991) The O_2 system. Commun ACM 34: 51–63

Franklin M, Zwilling M, Tan C, Carey MJ, DeWitt D (1992a) Crash recovery in client-server EXODUS. In: Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data, San Diego, Calif, June, pp 165–174

Franklin M, Carey MJ, Livny M (1992b) Global memory management in client-server DBMS architectures. In: Proceedings of the Eighteenth International Conference on Very Large Databases, Vancouver, BC, August, pp 596–609

Franklin M, Carey MJ, Livny M (1993) Local disk caching for client-server database systems. In: Proceedings of the Nineteenth International Conference on Very Large Databases, Dublin, Ireland, August, pp 641–654

Garza J, Kim W (1988) Transaction management in an object-oriented database system. In: Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data, Chicago, Ill, USA, June

Gray J, Reuter A (1993) Transaction processing: concepts and techniques. Morgan Kaufmann, San Mateo, Calif

Keen J, Dally W (1993) Performance evaluation of ephemeral logging. In: Proceedings of ACM-SIGMOD 1993 International Conference on Management of Data, Washington, DC, May

Kernighan B, Pike, R (1984) The UNIX programming environment. Prentice-Hall Software Series, Englewood Cliffs, NJ

Kim W, Garza J, Ballou N, Woelk D (1990) Architecture of the ORION next-generation database system. IEEE Trans Knowl Data Eng 2: 109–124

Lamb C, Landis G, Orenstein J, Weinreb D (1991) The ObjectStore database system. Commun ACM 34: 51–63

Lehman TJ, Lindsay BG (1989) The Starburst long field manager. In: Proceedings of the Fifteenth International Conference on Very Large Databases, Amsterdam, Netherlands, August, pp 375–383

Mohan C, Narang I (1991) Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In: Proceedings of the Seventeenth International Conference on Very Large Databases, Barcelona, Spain, September, pp 193–207

Mohan C, Narang I (1994) ARIES/CSA: a method for database recovery in client-server architectures. In: Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data, Minneapolis, Minn, May, pp 55–66

Mohan C, Haderle D, Lindsay BG, Pirahesh H, Schwarz P (1992) ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans Database Syst 17: 94–162

Moss JEB (1990) Design of the Mneme persistent object store. ACM Trans Inf Syst 8: 103–139

Stevens R (1990) UNIX network programming. Prentice-Hall Software Series, Englewood Cliffs, NJ

Stonebraker M, Kemnitz G (1991) The Postgres next-generation database management system. Commun ACM 34: 51–63

Wang Y, Rowe LA (1991) Cache consistency and concurrency control in client/server DBMS architecture. In: Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data, Denver, Colo, USA, May, pp 367–376

White SJ, DeWitt DJ (1995) Implementing crash recovery in quickstore: a performance study. In: Proceedings of ACM-SIGMOD 1995 International Conference on Management of Data, San Jose, Calif, May, pp 187–198

White SJ, DeWitt DJ (1994) QuickStore: a high performance mapped object store. In: Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data, Minneapolis, Minn, USA, May, pp 395–406

Wilkinson K, Neimat MA (1990) Maintaining consistency of client-cached data. In: Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane, Australia, August, pp 122–133