EOS User's Guide

Release 2.2

For UNIX-based Systems

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

Abstract

EOS is a storage manager providing key facilities for the fast development of high-performance database management systems and persistent languages. This document provides a brief overview of the EOS Release 2.2 architecture, facilities, information on using the system including examples, and setting up tuning parameter values to adjust EOS performance.

Contents

1 Introduction

EOS is a storage manager being developed at AT&T Bell Labs for the fast development of high-performance database management systems. The following is a brief summary of the facilities provided by EOS:

- Extensive support for large objects. Objects can be accessed transparently in the client's cache, without incurring any in-memory copying cost or via byte range operations such as read, write, append, insert, delete bytes, etc., specially suited for very large multimedia objects.
- Any object can be *named* for fast retrieval. Referential integrity between named objects and their corresponding names is enforced by EOS.
- *Page objects*, objects that expand over the entire available space in a page, which can be accessed in the same way as any other object.
- *Plain database pages* belonging to a particular storage area. They can be used to build index structures (We used them to build our extensible hashing.)
- *Extensible hashing* supporting variable size keys and user-defined hash and equality functions.
- *Database files* for grouping related objects together. *Databases* as collections of files and objects. Databases are stored in one or more storage areas (UNIX files or disk raw partitions) and each such area may contain many databases. Clustering hints for the physical placement of objects in pages, files, databases and areas.
- A simple and powerful mechanism that allows users to enhance and even modify the EOS functionality, without compromising modularity, by associating hook functions with certain primitive events. EOS traps the events as they occur and causes the corresponding hooks to be executed.
- *Transactions* in a *client-server* environment with the option, for experimentation, to turn on/off the concurrency control, logging and recovery components. Applications can be linked with the single-user version of EOS for accessing private/local databases.
- Concurrency control based on the multigranularity two version two phase locking protocol that allows many readers and one writer to access the same item simultaneously. The option to switch to simple 2PL is also provided.
- Short log files because log records contain only after images of updates.
- Fast recovery from system failures because only one forward scan over the log is required. The restart process can start at any checkpoint taken in the past and stop at any point after that.
- Non-blocking checkpoints that allow active transactions to continue accessing databases while a checkpoint is taken.

- Configuration files that can be edited by the users to customize and tune EOS performance.
- Persistent references that are valid across transactions boundaries, as well as across databases. Objects point to other objects by using persistent references.

EOS is written in C++. It can be accessed by programs compiled with any C or C++ compiler such as the ones distributed by AT&T, Sun, GNU, and CenterLine. Applications that use persistent references must be compiled with a compiler that supports templates. The EOS system works on Sun SPARC architectures running SunOS 4.1.x and SOLARIS 2.x, SGI MIPS architecture running IRIX 4.x and 5.x, and IBM RS6000 running AIX.

1.1 Contact for Further Information

For inquires about EOS and a copy of the EOS system please send e-mail to Alex Biliris at `biliris@research.att.com`, or write to

EOS Project Group
c/o A. Biliris
AT&T Bell Labs, Room 2C-221
600 Mountain Av.,
Murray Hill, NJ 07974

For bug reports please send e-mail to `eos@research.att.com`.

2 EOS Architecture and Facilities

2.1 Basics

2.1.1 Databases and Storage Areas

Databases are collections of files and ordinary objects. A database is created in one of the available *storage areas* – UNIX files or disk raw partitions. The objects a database contains may be stored either in the area where the database was created, or in other areas. Thus, a database may physically extend over many areas. Also, there may be areas in which no database has been created and they contain objects from different databases.

Storage areas are either shared or private. Shared areas are accessed via the EOS server that offers control for multi-user access to the area, as well as recovery. Private areas are created on the local machine of the user creating the area. Access to private areas is in general faster compared to shared areas because all operation are carried out locally with no calls to the server. However, no concurrency control and recovery is offered for private areas.

Disk space allocation in EOS is based on the *binary buddy system.* A storage area is organized as a number of fixed-size *extents* or *buddy spaces* – disk sections of physically adjacent pages. *Segments* are variable-size sequences of physically adjacent disk pages taken from one of the buddy spaces. There is a 1-block allocation map directory associated with each extent indicating the status (free or allocated) and the size of each segment in the extent. With 4K-byte disk blocks, the maximum extent size is approximately 63.5 megabytes [Bil92a]. To maximize performance, the extent size may have to be carefully matched to the physical properties of the disk device.

2.1.2 Object Identifiers (OIDs)

EOS objects are stored on slotted pages and they are identified by system generated object ids (oids). The object id is an 8-byte quantity; it is the physical address in which the object is stored in the database and it consists of the following: the storage area number and the page number within the area the object is stored in, the slot number that gives the offset of the object within the page, and a number to approximate unique ids when space is re-used.

2.1.3 Named Objects

Any EOS object can be given a name. An object may have at most one name, and a name may correspond to at most one object. EOS guarantees that the referential integrity between named objects and their corresponding names is enforced; e.g., when a named object is removed from a database so is the name of the object.

2.1.4 File Objects, File Scans and Clustering

EOS file objects serve as a mechanism to gather "related" objects together; i.e., objects that need to be collocated. They provide facilities for sequencing through the objects they contain. Files may contain other files and every object (including file objects) is a member of exactly one file object. Thus, objects form a tree where internal nodes are file objects and leaves are ordinary (non file) objects. When a new database is created, EOS automatically creates a file object that serves as the root file of this tree. On the physical level, a file consists of a number of single pages and/or segments. Pages are not shared among files.

Forward and backward scanning of objects in a file is supported. In addition, the set of objects on which the scan is performed can be restricted to the objects within a single page of the file being scanned. Such page-oriented file scans are useful for efficiently implementing various kinds of joins that require all objects in two or more files to be compared.

To improve performance, clients may exercise control over the physical placement of objects within a database. Specifically, at object creation, clients may instruct the object manager to place the new object near an existing one. EOS will store the new object on the same page as the existing object if space is available in that page, otherwise, the new object is placed on a new page that is near the page the existing object resides on. Clients may also instruct EOS to place the object in a specific storage area and (a) at the end of a file, (b) in a new page, or (c) not to assign the page on which the object is stored to any other object that is going to be created in the future (i.e., the new object will be the only one in that page). The latter may be useful in reducing contention because of locking for frequently accessed pages – *hot spots*.

2.1.5 Object Representation and Object Handles

Every EOS storage object has an object header attached to it. It contains properties of the object such as the object's length, whether it is small or large, named or unnamed, etc. Two bytes in the header have no meaning to the storage system, and they are available to users to store information about the object.

To operate on an EOS object, a handle to that object must be acquired. A handle is a structure that contains, among other information, the address of the object in the EOS buffer pool. When an application requests a handle on an object, the appropriate lock for the page the object is stored on is acquired, and the page itself is fixed in the buffer pool – that is, the page will not be replaced or moved in another place until it is unfixed. When the object is no longer needed, its handle must be released so that the corresponding page in the buffer pool is unfixed.

The cost of making a handle of a persistent object is at most one disk access which is the cost of fetching the page in which the object is stored; no additional cost is involved in translating an oid because it is a physical database address. After an object handle is acquired, the speed of subsequent accesses to the object is almost the same to that of an in-memory dereference of a pointer to resident data. This is because persistent objects can be manipulated directly on the page on which they reside, without incurring any in-memory copying cost.

2.1.6 Large Objects

EOS has been designed to handle arbitrary large objects provided that physical storage is indeed available and accessible. Technically speaking, an object is *small* if it can fit entirely in a single page, otherwise, it is *large*. Small objects may become large, and vice versa. Large objects can be accessed and updated in exactly the same way as small objects. Thus, the manipulation of large objects is transparent to the applications. For "very" large objects, however, users may want to access portions of the object. EOS provides operations that deal with a specific byte range within the – small or large – object: *read* or *write* a random byte range within the object, *insert* or *delete* bytes at arbitrary positions within the object, and *append* bytes at the end of the object.

Large objects are stored in a sequence of variable-size segments which are allocated as explained in section 2.1.1. These segments are then pointed to by a tree structure in which the keys are the positions of the object's bytes within the segments. When length changing updates (byte range deletes and inserts) are performed on the large object, its segments may have to be broken up into smaller ones. Since small segments have negative performance effect on the read operation, EOS allows the client to specify a segment size threshold T – a constraint on EOS not to store byte chunks in two (logically) adjacent segments, one of which has less than T pages, if they can be stored in one segment. Note that the threshold mechanism does not specify fixed size segments neither a minimum number of pages per segment. For example, with $T = 16$, a large object that is 2 pages and a half long is kept in three pages, not in 16 pages. The tradeoffs that need to be examined in order to set this value are the following. (See also [Bil92b] for performance results.) Larger segments lead to better storage utilization, lower (sequential and random) read costs and higher update cost; i.e., the only aspect of the performance that might be affected negatively by larger segments is the cost of byte inserts and deletes.

2.1.7 Indexes

EOS provides extendible hashing indexes [FNPS79]. Index keys can be variable size strings or any fixed-size structure. The values associated with the keys can be any fixed length structure.

2.1.8 Transactions

EOS provides full support for concurrency control, logging, and recovery. Concurrency control allows multiple users to access a database at the same time in a consistent way. When a transaction is committed, all its updates are permanently posted to the database. When a transaction is aborted by the user, the recovery's goal is to make the database look like as if the aborted transaction was never submitted for execution. The storage manager may also abort a transaction when it is the victim of the deadlock detection algorithm, or when the system malfunctions. In addition, when the system restarts after a crash, the database reflects the updates of all the committed transactions prior to the crash. (See [GR93] for a description on transaction processing).

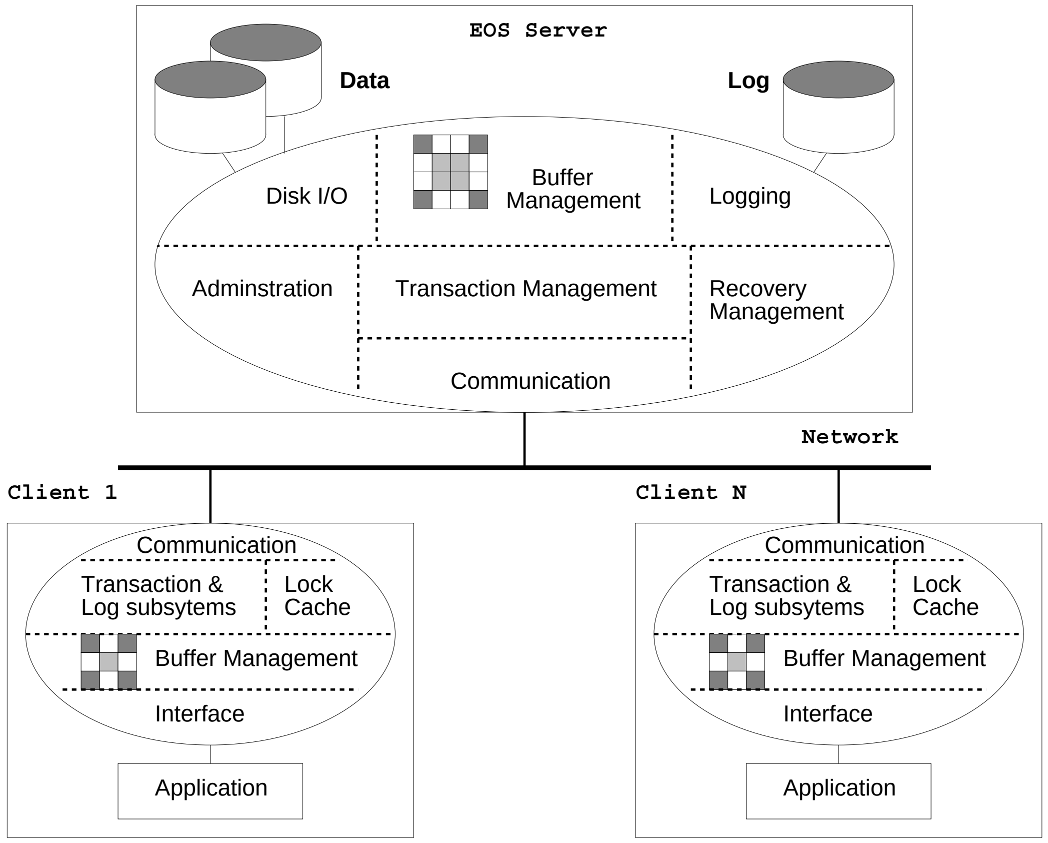

Figure 1: The client-server architecture of EOS

2.2 The EOS Client-Server Architecture

Figure 1 sketches the EOS client-server architecture. The EOS server is a multi-threaded daemon process that mediates all the accesses made to the database. To avoid blocking UNIX disk I/O system calls, the EOS server creates a separate disk process to handle the I/O requests for a storage area the very first time this area is accessed. In this way disk I/O is performed in parallel. Dynamic creation and deletion of storages areas is also hanled by the server (it plays the role of the *area manager* process) and the current release can virtually support up to 6000 different areas. At start up time, the server spawns the checkpoint and the global log processes, and creates a number shared memory regions and semaphores by using the UNIX System V shared memory, memory mapping, and semaphore facilities. The EOS server runs as a separate process on the same or on a different machine than a client application program. The communication between the server and the client workstations is done by using reliable TCP/IP connections over UNIX sockets [Ste90]. The server buffer pool is stored in shared memory and the disk processes access it directly. Semaphores provide controlled access to the structures shared by all threads and disk processes, message queues and UNIX domain sockets [KP84, Ste90] provide the interprocess communication among the threads and the I/O requests directed to the disk processes.

2.2.1 Concurrency Control

Concurrency is controled by a page-level mulrigranularity 2-Version 2-Phase locking (*MG-2V-2PL*) protocol. Transactions acquire locks on data items before they access them, and they release all locks they hold when they are finished (committed or aborted). When a page is locked, the file containing this page is locked too in the corresponding intention mode. When a file is locked in either S or X mode, the pages it contains do not have to be locked explicitly. This minimizes the overhead of the concurrency control module. In this locking scheme, a number of readers and at most one writer can be operating simultaneously on the same granule. The writer has to wait for all the readers to finish before it can commit. Deadlocks are handled by a variation of the *depth first traversal* algorithm applied on the waits-for graph (WFG), constructed from the lock tables, to determine if a cycle exists. Deadlock detection is performed every time a lock request is blocked by another transaction which is blocked too. If a deadlock cycle is found, the requester gets aborted.

2.2.2 Logging

The system maintains two kinds of logs: a *global* log, and a number of *private* logs. Each private log is associated with one transaction only. The log records of a private log are *redo* records, i.e., they contain the results (after images) of the updates generated by the corresponding transactions. The global log contains records that are either commit records or checkpoint records. A commit log record contains the committed transaction's id and other information related to transaction's updates. The checkpoint record contains the ids of all the committed transactions at the time the checkpoint took place, along with the offset in the global log of their commit log record.

2.2.3 Transaction Commit and Abort

EOS recovery is based on the NO-UNDO/REDO protocol. Transaction updates are applied to the transaction's private cache and they are posted to the database only after the transaction is committed. When a transaction commits the transaction updates recorded in its private log are flushed out on stable storage, a *commit* log record is inserted into the global log, and the global log is flushed on stable storage. If all steps are successful, the transaction is declared committed. When a transaction is aborted, its private log is simply discarded and its locks are released.

Recovery for large objects differs from that mentioned above. First, large objects are not buffered in the server's shared pool. Secondly, updates on large objects are applied directly to the database without, however, overwriting the object in the database. When a transaction gets aborted, the changes it made on the extent's directory, which keeps track of free pages, 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 directory.

2.2.4 Checkpoint

To reduce the amount of work the recovery manager has to do during system restart, the EOS server periodically issues checkpoints. During a checkpoint dirty pages buffered in the shared pool are flushed to the stable storage. When the checkpoint procedure completes, a checkpoint record is inserted in the global log file and the location of this checkpoint record is saved on a well known location. The EOS checkpoint is non-blocking: new transactions can begin and active transactions may continue accessing the server and its resources while the checkpoint activity takes place.

2.2.5 Recovery from System Crash (Restart)

If a system crash occurs, the EOS server returns the database to the last consistent state it was in before the failure. This is done by scanning the 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 has been restored, a checkpoint is taken and the system is operational again. If a system failure occurs while the restart is performed, the subsequent restart performs the same work again in an idempotent fashion. The restart procedure of EOS is fast for two reasons: (a) only one forward pass over the log is required, and (b) the log itself is short because only after images of updates are logged.

EOS provides continuous archiving of the private log files. The archiving process is activated by the global log manager every time a checkpoint is taken and the number of the log files generated since the last archive is greater than some threshold value. The private log files created since the last time the archive process was executed are compressed and merged together on a single file.

2.3 Advanced Features

2.3.1 Page Objects and Plain Pages

Page objects are fixed-length object that expand in size over the entire available space of a page. They are useful in building various index structures such as B-trees and hash tables. An object is specified to be a page object when it is created; after that, it is accessed in exactly the same way as ordinary objects.

Plain pages are pages belonging to a particular storage area and they do not contain any control information; i.e., the entire space of the page is available to the application. Plain pages can be used for building index structures.

2.3.2 Extensions and Primitive Events

EOS allows users to extend and even modify the functionality of EOS by associating actions that are executed when certain primitive events occur. Extensions provide a degree of extensibility without compromising modularity.

Primitive events are low-level events that occur at various software layers of EOS, such as page fault, object fault, object update, transaction commit, etc. Applications can associate one or more hook functions with a particular event. This registration process is usually performed at the beginning of a program before any access to the database is initiated. For some events, such as transaction commit, applications will have to specify if the action should be executed just before or just after the occurrence of the event. The EOS event manager traps primitive events as they occur and causes the actions that applications associated with these events to be executed. Actions for a particular event are executed in the order in which they were registered.

This facility can be used to extent the EOS functionality in many ways, e.g, to to collect statistics on object faults or transaction commits, or to set access privileges so that only certain users can access an object (or a file) in the database or the database itself. As another example, a hook may be registered that restarts a transaction after it is being aborted due to a deadlock.

3 Getting Started with EOS

To access the binaries and man pages in the `eos` directory, set your path and manpath environment variables to include the `eos/bin` and `eos/man` directories, respectively. The following shows how to do this, assuming that EOS is installed in `/usr/eos`.

If you use csh or tcsh, place the following in your .cshrc:

```
set path = ($path /usr/eos/bin)
setenv MANPATH $MANPATH:/usr/eos/man
```

If you use Bourne or Korn shell, place the following in your .profile:

```
PATH = $PATH:/usr/eos/bin; export PATH
MANPATH = $MANPATH:/usr/eos/man; export MANPATH
```

The rest of this section describes the simplest way of configuring the EOS storage manager, and running the three example application programs `obj_create, obj_scan,` and `obj_update` included in the `eos/example` directory of the EOS distibution. The `obj_create` creates a number of objects and puts them in a named file belonging to a database. The `obj_scan` reads all the objects created and prints them on the screen. Finally, the `obj_update` randomly updates several of the created objects. Go to the `eos/example` directory and compile these programs to be ready for execution.

These are the steps that have to be followed:

1. Install the configuration files by running the following three programs:

   ```
   eosformatenv
   eosclientenv
   eosserverenv
   ```

   These programs create the files `formatrc`, `clientrc`, and `serverrc`, respectively, in the directory `.eos` under your home directory.

2. Start the server. Open a new window and go in the `bin` directory where the `eosserver` executable is. To start up the server, type

   ```
   eosserver
   ```

   When the server is ready to accept requests the following message appears on the screen:

   EOS server. We are open for business.

3. Create and format a storage area. Let us assume that you want to create an EOS storage area in the UNIX file `/usr/sue/area`. Run the command

```
eosareaformat /usr/sue/area
```

You are now ready to create databases in this area and populate it with objects.

4. Run the `obj_create` to create a number of objects in a database belonging to the area you have just created. To create 1000 objects type the following:

   ```
   obj_create /usr/sue/area/myfirstdb 1000
   ```

   To read the objects you have just create, type

   ```
   obj_scan /usr/sue/area/myfirstdb
   ```

   To update few of the objects you have just create, type

   ```
   obj_update /usr/sue/area/myfirstdb
   ```

   If you want to run the `obj_scan` and `obj_update` programs concurrently, open up two new windows and run the `obj_scan` on one and the `obj_update` on the other. Running them concurrently may result to periodic deadlocks because both programs access the same file in the same database. When a deadlock occurs, the offending program is terminated.

5. If you want to shut down the server, go to the window where the server runs and type the following command:

   ```
   shutdown
   ```

   The server will print a number of messages on the screen and when it finishes processing the shutdown command the following message appears on the screen:

   EOS server: SHUTTING DOWN. Closing the shop.

3.1 Compiling and Linking your Application

Application programs written in C++ or C must include the `eos.h` file. Application programs that use persistent references must include the `eos_Ref.h` file in addition to `eos.h`. For a C++ application, use the `-I` option of the compiler to add the `eos/include` directory to the list of directories the pre-processor searches for `#include` files. For a C application, add the `eos/include_c` path to the list.

EOS offers two basic libraries: `client.o`, and `private.o`. An application program that uses the EOS server must be linked with the `client.o` library. If the application program manipulates private, local databases, the `private.o` library is linked with it.

In addition, programs compiled with the C compiler must also be linked with the `cfuncs.o` library.

Since the EOS libraries have been generated by a C++ compiler, the linking must be done with a C++ compiler.

4 Formatting And Deleting Storage Areas

Storage areas are created, formatted, and removed by invoking EOS commands at the UNIX prompt. An area can be either a UNIX file or a disk raw partition, the latter must already exists in order for EOS to format it.

As it was mentioned in Section 2.1.1, an area cam be either shared and private. Shared areas are accessed via the EOS server. Private areas are created on the local machine of the user creating the area. The file `~/.eos/areas` of the server account keeps information about all areas formatted by the server. Information about a private area is kept in the `~/.eos/areas` file of the user formatting the area.

4.1 Formatting Areas (`eosareaformat`)

The command `eosareaformat` formats an EOS storage area. If the area is a UNIX file and the file does not exist, `eosareaformat` creates it first. The rooted path name of the area being formatted must be specified when the command is invoked. For example,

```
eosareaformat /usr/thimios/eos_area
```

formats the UNIX file `/usr/thimios/eos_area`.

In general, an area name has the following form:

`[` *host_name*`:]` *rooted_area_path*

where *host_name* is the machine name the server is running on. If it is not specified, the name of the area manager's host machine is taken from the the EOS configuration file `~/.eos/formatrc`, see section 7.1.

The complete set of command line arguments of the `eosareaformat` command is the following:

`eosareaformat` *aname* [`-l`] [`-a` *num*] [`-s` *num*] [`-n` *num*] [`-d` *num*] [`-e`] [`-o`]

-l	It formats a private area on the local machine. The EOS server is not contacted. Information about this area is kept in the `~/.eos/areas` file of the user executing the command.
-a *num*	The area number to be assigned to the area. If no number is specified, the system will pick a unique one.
-s *num*	The size of each extent (in pages of 4K-bytes).
-n *num*	The number of extents in the area.
-d *num*	The maximum number of databases that can be created in this area.

-e	It makes the area expandable; the area will grow dynamically as more space is needed, by appending one extent at a time to the existing ones. The area must be a UNIX file to be specified expandable.
-o	If the area exists, it is re-formatted and all its contents are purged.

The values of the command line arguments that are not specified by the user are taken from the `~/.eos/formatrc` configuration file, see section 7.1.

4.2 Deleting Areas (`eosareadelete`)

The `eosareadelete` command removes an area given either its name or its number:

```
eosareadelete aname [-l]
eosareadelete -a num [-l]
```

-l	It removes a private area on the local machine. The EOS server is not contacted. The record of this area is removed from the `~/.eos/areas` file of the user executing the command.
aname	The rooted path name of the area to be deleted.
-a *num*	The area number of the area to be deleted.

When an area is deleted by the server, the following actions take place:

1. Information about the deleted area is erased from the `~/.eos/areas` file of the server account.
2. The disk process servicing the deleted area exits.
3. The area is removed (if it is a UNIX file).

Removing or reformatting an area dynamically – while the server is running – is highly discouraged because it is inherently unsafe and it may result in a system failure. We suggest to perform such operations when the server is not active because then they can be performed in a safe way. Go to the server's account and use the command `eosareaformat` with the `-o -l` options to reformat an existing area, and `eosareadelete` command with the `-l` to delete an area. However, if you want to delete or re-format an area while the server is running make sure that there is no active transaction using this area,

5 The EOS Client C++ Interface

The following sections describe the EOS C++ classes available to programmers.

The function `eos_begin()`, which initializes the internal structures of the EOS storage manager, must be invoked before calling any other EOS function.

5.1 Object ID (`class eosoid`)

EOS objects are identified by object ids of type `eosoid`. The data members of the `eosoid` class are the following:

```
    eosspid pno;                // the page number the object resides on
    unsigned ano :  14;         // the area number the page resides in
    unsigned uno :  8;          // number that approximates unique oids
    unsigned sno :  10;         // slot number in the page that points to the object
```

`static const eosoid::null`
is the null object id.

`int is_valid(void) const`
returns true if the oid is a valid object id (i.e., there exists an object with such id), otherwise it returns false.

```
int operator==(const eosoid& oid) const
int operator!=(const eosoid& oid) const
int operator>=(const eosoid& oid) const
int operator<=(const eosoid& oid) const
int operator>(const eosoid& oid) const
int operator<(const eosoid& oid) const
```
return true if `this` and `oid` satisfy the corresponding relational operator.

5.2 Persistent New (`class eos_new`)

Persistent object are created by either using the `eosobj` class described in Section 5.6, or by using the overloaded operator `new` provided by EOS. EOS's persistent new takes one argument, which specifies where the new persistent object is going to be placed. Persistent objects can be placed in an EOS database, or in an EOS file, or they can be placed close to another persistent object. The interface provided is the following:

```
void* operator new(size_t size, const eosdatabase* db)
void* operator new(size_t size, const eosfile* pfile)
void* operator new(size_t size, const eosobj* obj)
void* operator new(size_t size, const eos_Ref_Any & refObj)
```

The EOS persistent new returns a pointer to the data part of the newly created persistent object.

CAUTION: The SPARCompiler C++4.0 does not invoke EOS-2's overloaded `new` operators for objects that are of a nonclass type and arrays of class objects. In this case, the `eosobj::create()` function should be used.

5.3 Persistent References (`class eos_Ref`)

Objects may refer to other objects through a persistent reference called an `eos_Ref`. Persistent references are valid across transaction boundaries, as well as across databases. The class `eos_Ref` is parameterized, with a parameter for indicating the type of the object being referenced by the persistent reference. This means that `eos_Ref` must be used by enclosing in angle brackets the name of the referent type. For example, we can define a persistent reference to an object of type `employee` in the following way:

```
    eos_Ref<employee> emp;
```

EOS also provides an `eos_Ref_Any` class that provides a generic reference to an object of any type. The `eos_Ref` class provides an operator for performing the conversion from a reference to an object of type `T` to an `eos_Ref_Any`, according to the ODMG-93 [Cat93] standards.

`eos_Ref(void)`
the default constructor for this class. The persistent reference is initialized to NULL.

`eos_Ref(T *fromObj)`
constructs the persistent reference to an object of type `T` given a virtual memory pointer to that object.

`eos_Ref(const eos_Ref<T> &)`
constructs the persistent reference to an object of type `T` given a persistent reference to that object.

`eos_Ref(const eos_Ref_Any &)`
constructs the persistent reference to an object from a generic reference.

`~ eos_Ref()`
the destructor for this class.

`operator eos_Ref_Any(void) const`
converts an `eos_Ref<T>` to an `eos_Ref_Any`.

`void clear(void)`
sets the persistent reference to NULL.

`int is_null(void) const)`
return true if the persistent reference is NULL.

`int delete_object(void)`
deletes the object that is referenced by this persistent reference.

```
T * ptr(void) const
```
returns a memory pointer to the object referenced by the persistent reference. The pointer is only valid until the end of the transaction or until the object it points to is deleted.

```
T * operator -> (void) const
```
dereferences the persistent reference and returns the valid `T *` for which the specified reference is a substitute.

```
T & operator * (void) const
```
dereferences the persistent reference and returns the valid `T *` for which the specified reference is a substitute.

```
eos_Ref & operator = (T *)
eos_Ref & operator = (const eos_Ref<T> &)
eos_Ref & operator = (const eos_ref_Any &)
```
these operator are used for copy assignment.

```
friend int operator==(const eos_Ref<T>& refL, const eos_Ref<T>& refR)
friend int operator==(const eos_Ref<T>& refL, const T *ptrR)
friend int operator==(const T *ptrL, eos_Ref<T>& refR)
friend int operator==(const eos_Ref<T>& refL, const eos_Ref_Any& anyR)
friend int operator==(const eos_Ref_Any& anyL, eos_Ref<T>& refR)
friend int operator!=(const eos_Ref<T>& refL, const eos_Ref<T>& refR)
friend int operator!=(const eos_Ref<T>& refL, const T *ptrR)
friend int operator!=(const T *ptrL, eos_Ref<T>& refR)
friend int operator!=(const eos_Ref<T>& refL, const eos_Ref_Any& anyR)
friend int operator!=(const eos_Ref_Any& anyL, eos_Ref<T>& refR)
```
these operators are used for testing for equality and inequality.

5.4 Databases (`class eosdatabase`)

The following functions create, open, remove, and truncate a database. A database may be opened many times; it will be closed when an equal number of database close function invocations are performed. In addition, a number of databases, belonging to the same or different areas, can be open at any point in time.

```
static eosdatabase* open(const char *name,
                            int rdonly=0, int create=0,int trunc=0)
```

opens the database specified in `name`. The `name` argument must have the following format:

[*hostname*:] *areaname* / *dbname*

If *hostname* is omitted, it's value is taken from the `clientrc` configuration file. Note that the *areaname* must be a rooted path relative to the server, not the client. The remaining of the arguments are used in the following way:

`rdonly`	if true the database is opened for reading only.
`create`	if true the database will be created if it does not exist.
`trunc`	if true the contents of the database will be purged.

For example the following piece of code opens the database `phones` in area `/usr/alex/area1` on machine `allegra`, if it exists.

```
eosdatabase *db = eosdatabase::open(allegra:/usr/alex/area1/phones);
```

`static eosdatabase* of(const eosobj* obj)`
returns the database descriptor of the database that contains the object `obj`, or null on failure.

`int open(int rdonly=0)`
opens the database whose descriptor that has been retrieved by using one of the static functions `eosdatabase::open()` or `eosdatabase::of()`. A non-zero value argument opens the database with read-only access. It returns zero on success, non-zero on failure.

`int close(void)`
closes the database. It returns zero on success, non-zero on failure.

`int destroy()`
removes this database. It returns zero on success, non-zero on failure.

`int rename(const char* new_name)`
changes the name of the database to `new_name` provided no other database in the same area has name `new_name`. It returns zero on success, non-zero on failure.

`int is_readonly(void) const`
returns true if the database has been opened for read-only access.

`const char* name(void)`
returns the name of the database.

`const eosoid& rootoid(void) const`
returns the object id of the root file.

`eosoid oid_of(const char *name)`
returns the oid of the object whose name is `name`. If no such object exists in `this` database, the function returns `eosoid::null`.

`int set_object_name(const eos_Ref_Any& objRef, const char* name)`
sets the name of the object referenced by `objRef` to `name`. Zero is returned on success and non-zero of failure.

`int remove_object_name(const eos_Ref_Any& objRef)`
removes the name associated with the objected referenced by `objRef`. Zero is returned on success and non-zero of failure.

`int rename_object(const char* old_name, const char* new_name)`
replaces the name of the object whose name is `old_name` with `new_name`. Zero is returned on success and non-zero of failure.

`const char* get_object_name(const eos_Ref_Any& objRef) const`
returns the name on the object referenced by `objRef`.

`eos_Ref_Any lookup_object(const char* name) const`
returns a reference to the object whose name is `name`. The validity of the returned reference must be checked using the `eos_Ref_Any::is_null()` member function.

5.5 Transactions (`class eostrans`)

All accesses to EOS objects (except database opening and closing) must be done within a transaction block.

`static int begin(int rdonly = 0)`
begins a transaction block. If `rdonly` is true, the transaction is a read-only transaction and no updates within the transaction area allowed. It returns zero on success, non-zero on failure.

`static int commit(void)`
commits an active transaction. It returns zero on success, non-zero on failure.

`static int abort(int normal = 1)`
aborts an active transaction. If `normal` is true, it returns 0 on successful completion and non-zero on failure. If `normal` is false, the application program exits after the completion of abort.

`static int is_active(void)`
returns true if the application program has already started a transaction. Otherwise, it returns false.

`static int is_readonly(void)`
returns true if a transaction has begun and it is read-only.

5.6 EOS Objects (`class eosobj`)

EOS objects are manipulated via object handles. There is one handle for each object fetched from the database. Having a handle to an object implies that the page the object resides on is fixed in the buffer pool – i.e., the page cannot be replaced or moved until it is explicitly unfixed.

5.6.1 Creating and Removing Objects

```
static eosobj* create(int size, eosdatabase *db, const void* data=0,
                      int flags=0, int hint=0, int ano=0)
```

creates an EOS object of size `size` in the root file of the database `db`. The new object is physically created in the EOS area whose number is `ano` unless `ano` is zero in which case it is created in the same area the database was created.

```
static eosobj* create(int size, eosfile *pfile, const void* data=0,
                      int flags=0, int hint=0, int ano=0)
```

creates an EOS object of size `size` in the file `pfile`. The new object is physically created in the EOS area whose number is `ano` unless `ano` is zero in which case it is created in the same area the file was created.

```
static eosobj* create(int size, eosobj *obj, const void* data=0,
                      int flags=0, int hint=0)
```

creates an EOS object of size `size` in the file in which the object `obj` belongs. The new object is created on the same page the object `obj` is stored if space is available, or in a new page close to it and always in the same area `obj` is stored.

On success, these three functions return the address of the new object handle. On failure, they return NULL.

If `data` is NULL the object remains uninitialized; otherwise, it is initialized with the first `size` bytes pointed by `data`.

The `hint` value is a hint about the potential size of the object being created and it is taken into consideration only if `hint` > `size`. EOS will place the object in a page that can accommodate MAX(`hint`, `size`) bytes in anticipation of the object growth. If no page can accommodate this amount of space, EOS switches to a different representation suitable for large objects in way that is invisible to the client.

The `flags` value is used for further control of the placement of the new object. The `flags` value is constructed by ORing constants from the following list:

`eosobj::NEAR_LAST`	The new object is appended to the file, i.e., it is placed after the last object in the file. This is also the default action when the value of flags is 0.
`eosobj::NEW_PAGE`	The new object is placed in a brand new page even if some other page can accommodate this object.
`eosobj::NO_FILL`	The page in which the new object is stored will not be assigned to any other object that might be created in the future.
`eosobj::HDR_ONLY`	Only the object header remains in the client's pool after the object's creation. This is useful, when the object is large and the user intents to create the object in pieces by successive appends. If this option is specified the value returned by `mptr()` is garbage and the object can be accessed only via the byte-range operations described in section 5.6.6.

`eosobj::VAR_LENGTH`	It creates a variable length object. Length changing updates can be applied on this object by using the byte-range function calls described in section 5.6.6.

`int destroy(void)`
removes this object. It returns zero on success, non-zero on failure.

5.6.2 Accessing Objects

`static eosobj* get(const eosoid& oid, int flags = 0)`
returns a handle to the object with id `oid`. On failure, it returns NULL. The `flags` value is constructed by ORing constants from the following list:

`eosobj::DIRTY`	The object is marked dirty.
`eosobj::HDR_ONLY`	If the object is large, the header of the object only is fetched. If this option is specified, the value returned by `mptr()` is garbage and byte range operations must be used to access the object.

`static eosobj* get(const eosdatabase* db, const char* name, int flags = 0)`
returns a handle to the object with name `name` in the database `db`. The argument `flags` is used as explained above.

`int release(void)`
releases the handle for an object. It returns zero on success, non-zero on failure.

`void* mptr(void) const`
returns the memory address of the object in the local buffer pool. If the application intents to update the object, then the EOS storage manager must be informed *before* the update materializes, so that the right locks are requested. This can be done either when `get()` is called, by passing the `eosobj::DIRTY` flag, or by using `touch()`.

`int touch(void)`
marks dirty the objects referenced by `this`.

`fetch_all(void)`
fetches the entire object in the application's space. It has no effect if the object is small or the object is large and it is already fetched in. This is useful when the handle to this object was obtained by using the `eosobj::HDR_ONLY` flag. Thus, applications may get an object with eosobj::HDR_ONLY set, examine the header and then decide whether they want the entire object.

5.6.3 Object Properties

`int size(void) const`
returns the size of the object in bytes.

`const eosoid& oid(void) const`
returns the object id of the object.

`const eosoid& parentoid(void) const`
returns the object id of the file containing the object.

`const eosoid& rootoid(void) const`
returns the object id of the database's root file containing the object.

`int is_root(void) const`
returns true if the object is the root file.

`int is_file(void) const`
returns true if the object is a file object.

`int is_large(void) const`
returns true if the object is large.

`int is_named(void) const`
returns true if the object has a name.

`int is_var_length(void) const`
returns true if the object is variable length – i.e., it was created with the `eosobj::VAR_LENGTH` flag set.

`int is_page_object(void) const`
returns true if the object is a page object.

`int is_index(void) const`
returns true if the object is a the directory of an extendible hash index.

`int is_dirty(void) const`
returns true if the object has been modified by the current transaction.

5.6.4 Naming Objects

`int name_set(const char *name)`
sets the name `name` to the persistent object. It returns zero on success, non-zero on failure.

`const char* name(void)`
returns the name of the object, or 0 if the object has no name.

`int name_remove(void)`
removes the name of the object. It returns zero on success, non-zero on failure.

5.6.5 Tagging Objects

`int utag_set(void* tag)`
sets the value of the 2-byte tag associated with the object to the value of the first two bytes pointed by `tag`.

`const void* utag_get()`
returns a pointer to the object's 2-byte tag.

5.6.6 Accessing Portions of Objects – Byte Range Operations

Byte range operations can be applied to both small and large objects but they are specially useful for very large objects that cannot be accessed in one step. A byte range (`offset, n`) defines the start byte `offset` and the number of bytes `n`. The first byte of the object is at offset 0. Thus, for an object of size s, a byte range is valid if $0 \leq$ `offset` $< s$, and `n` ≥ 0, and `offset` $+$ `n` $\leq s$. These functions return 0 on success, non-zero on failure.

`int read(void *buf, int offset, int n) const`
reads `n` bytes of the object starting at offset `offset` into the buffer pointed by `buf`.

`int write(const void *buf, int offset, int n)`
replaces (overwrites) `n` bytes of the object starting at offset `offset` with the first `n` bytes pointed to by `buf`.

`int append(const void *buf, int n, int hint)`
appends the first `n` bytes pointed to by `buf` at the end of the (possibly 0 size) object. The `hint` can be used when the object is written with several append operations and it indicates an estimate of the total object size. If the precise size of the object is not known, it is always good to overestimate the size. The `hint` is taken into consideration only if its value is greater than the sum of `n` and the current object size.

`int insert(const void *buf, int offset, int n)`
inserts the first `n` bytes pointed to by `buf` into the object starting at offset `offset`. None of the existing bytes is overwritten.

`int delete_range(int offset, int n)`
deletes `n` bytes of the object starting at offset `offset`. Deleting all bytes of an object does not delete the object itself.

`int truncate(int offset)`
deletes all bytes on the right of and including byte `offset` of the object. The size of the object becomes `offset`.

5.7 File Objects (`class eosfile`)

`class eosfile : private eosobj`

EOS treats files in the same way as it treats objects. A file can be created within another file, and a file can have a name as any other object.

```
static eosfile* create(eosfile* pfile, const char* name=0,
                       int flags=0, int ano=0)
```

creates a new file. The new file is a child of the file `pfile` and it is created in area `ano` if the value passed is different than 0. Otherwise, it is created in the same area as the `pfile`. The new file is unnamed, unless `name` points to a string in which case the file gets this name. Applications that want to associate names with files may first create an unnamed file and then give a name to it. The parameter flags is used as in the `eosobj::create` function.

```
static eosfile* create(eosdatabase* db, const char* name=0,
                     int flags=0, int ano=0)
```

creates a new file as a child of the root file of the database `db`.

`static eosfile* open(const eosoid& oid)`
opens the file with id `oid`. It returns the file descriptor or zero on failure.

`static eosfile* open(const eosdatabase* db, const char* name)`
opens the file with name `name` in the database `db`. It returns the file descriptor or zero on failure.

`static eosfile* of(const eosobj *obj)`
returns the file descriptor of the file containing the object `obj`.

`int destroy(void)`
removes the file and all the objects it contains. An attempt to remove a root file will result in error; the root file of a database is removed when the database itself is removed. It returns zero on success, non-zero on failure.

`int clear(void)`
removes all the objects belonging to the file without removing the file itself. It returns zero on success, non-zero on failure.

`int npages(void) const`
returns the number of pages the file has.

`int close(void)`
closes an open file. It returns zero on success, non-zero on failure.

The following functions of `eosobj` are also public members of the `eosfile` class:

```
    oid
    parentoid
    rootoid
    utag_get
    utag_set
    name
    name_set
    name_remove
    is_named
    is_root
```

5.8 File Scan (`class eosfilescan`)

Visiting objects belonging to a file is done by opening a scan for this file. The state of a scan is recorded by a cursor which points to the "current" object in the file being scanned.

```
static eosfilescan* open(eosfile *fh, int order=AUTO_FWRD,
                         const eosoid& oid = eosoid::null)
```

returns a scan for the file `fh` or zero on failure. The value of `order` specifies the order in which the object of the file will be scanned. It can take be one of the following values:

`eosfilescan::AUTO_FWRD`	automatic scan of the file in forward order starting at object with oid `oid` or at the very first object if `oid` is the null eosoid.
`eosfilescan::AUTO_BWRD`	automatic scan of the file in backward order starting at object with oid `oid` or at the very last object if `oid` is the null eosoid.
`eosfilescan::MANUAL`	explicit manual movement of the cursor. If `oid` is not the null oid, the cursor is position at the object with oid `oid`.

`eosoid cursor(void)`
returns the id of the object pointed by the "cursor". The `eosoid::null` is returned when the cursor does not point to a valid object, or there are no more objects in the file to be visited. If the `eosfilescan::AUTO_FWRD` order was specified, the cursor will be positioned to the next object. If the `eosfilescan::AUTO_BWRD` order was specified, the cursor will be positioned on the previous object. By specifying the `eosfilescan::MANUAL`, the cursor will remain unchanged.

`int eosfilescan::close()`
closes a file scan.

When the `eosfilescan::MANUAL` order is specified, the cursor must be moved explicitly by using the following functions. They all return zero on success, non-zero on failure. Note that it is NOT an error to seek to the first, next, previous, or last object when no such object exists in the file or the page depending on the kind of scan. In this case, a call to `cursor()` will simply return `eosoid::null`.

`int first(int inpage=0)`
positions the cursor to the first object within the current page, if the value of `inpage` is true, or within the file if its value is false.

`int last(int inpage=0)`
positions the cursor to the last object within the current page, if the value of `inpage` is true, or within the file if its value is false.

`int next(int inpage=0)`
positions the cursor to the next object within the current page, if the value of `inpage` is true, or within the file if its value is false.

`int prev(int inpage=0)`
positions the cursor to the previous object within the current page, if the value of `inpage` is true, or within the file if its value is false.

`int seek_at(const eosoid& oid)`
positions the cursor on the object with id `oid`. It is an error to attempt to position the cursor on an object that does not exist or it is not an immediate member of the file being scanned.

5.9 Page Objects (`class eospageobj`)

```
class eospageobj :  public eosobj
```

```
static eospageobj* create(eosfile* pf, int flags = 0, int ano = 0,
                                   const eosoid& near = eosoid::null)
```

creates a page object in file `pf` and returns the handle to the new object; it return NULL on failure. The rest of the arguments are the same with the ones of `eosobj::create()`.

5.10 Plain Database Pages (`class eospage`)

EOS provides direct access to plain pages. An application program can create, destroy, pin in the local buffer pool, and unpin a plain database page belonging to any storage area. A plain page is associated only with the storage area it belongs to.

`int create(int ano)`
allocates and pins down in the local buffer pool a new plain page in the storage area `ano`. It returns zero on success, non-zero on failure.

`int destroy(void)`
unpins the plain page associated with `this` and it returns the page to the storage area it belongs. It returns zero on success, non-zero on failure.

`static int destroy(eosspid pno, int ano)`
frees the page `pno` belonging to the storage area `ano`. If the page appears in the local buffer pool it is invalidated. It returns zero on success, non-zero on failure.

`int pin(eosspid pno, int ano, int flags = 0)`
pins down in the local buffer pool the page `pno` belonging to the storage area `ano`. If the value of `flags` is `eospage::DIRTY` then an exclusive lock is acquired on this page; otherwise a shared lock is obtained. It returns zero on success, non-zero on failure.

`int unpin(int flags = 0)`
unpins the page pointed by `this`. If the value of `flags` is `eospage::UNLOCK` then the lock held on this page is released. It returns zero on success, non-zero on failure.

`int touch(int flags)`
marks the page pointed by `this` dirty when `flags` have the value `eospage::DIRTY`. This means that an exclusive lock is acquired on the page. It returns zero on success, non-zero on failure.

`void* mptr(void)`
returns a memory pointer to the page.

`const eospid& pid(void)`
returns the `eospid` id of the page pointed by `this`.

`int size(void)`
returns the size of the page.

5.11 Extendible Hashing (`eosehash`)

EOS provides extendible hashing indexing facility that associates keys with fixed-size values. More information about the extendible hashing can be found in [FNPS79]. The keys can be either fixed-length structures or variable length strings. In addition, user-defined key-hash and key-equality functions may be provided.

5.11.1 Creating and Destroying Indexes

```
static eosehash* create(eosfile* pfile, unsigned key_size, unsigned val_size,
                        int unique, int string, int init_size,
                        HASH_FUNC hashf = 0, EQ_FUNC eqf = 0, int ano = 0)
```

creates a new extendible hash index and returns the address of the handle created on success. On failure it returns NULL.

The directory of the new index is an EOS object created in the file `pfile`. The `ano` specifies the area number in which buckets od this index are allocated. If `ano` is zero, the buckets are allocated in the same area in which the directory is stored. `key_size` gives the maximum length of a key. `val_size` gives the size of each value associated with a key and it must be a multiple of four. If `unique` is 0 then multiple values can be associated with the same key. Zero-length values are allowed only when the index is unique. If `string` is true then the keys are variable size strings. `init_size` specifies the initial number of buckets in the hash.

`hashf` and `eqf` are the user defined hash and key-equality functions, respectively. Their prototypes are:

```
typedef unsigned (* HASH_FUNC)(const void *key)
typedef int      (* EQ_FUNC )(const void *, const void *)
```

`int destroy(void)`
destroys an existing extendible hash structure. All the entries in the index are destroyed and the directory itself is deleted.

5.11.2 Accessing Indexes

`static eosehash* open(const eosoid& oid, HASH_FUNC hashf=0, EQ_FUNC eqf=0)`
opens an extendible hash index whose oid is `oid`. The `hashf` and `eqf` are the functions to be used for hashing and comparing for equality index keys. Their prototypes are those described in the previous section. On success the address of the handle created is returned. On failure NULL is returned.

`int close(void)`
releases the handle for the index. It returns zero on success, non-zero on failure.

5.11.3 Inserting and Removing Index Elements

`int insert(const void *key, const void *value)`
inserts the <`key, value`> pair into the index. If the index is a unique-key index and the key already exists then a non-zero value is returned as indication for the error. On success zero is returned.

`int remove(const void *key, const void *value=0, EQ_FUNC eqf=0)`
removes the <`key, value`> pair from the index when when `value` is not NULL. If `value` is NULL, all pairs whose key part is `key` are removed. The `eqf` is used to find the `value` that has to be removed, when it is provided; otherwise, `memcmp(3)` is used. On success zero is returned. If an error occurs non-zero is returned.

5.11.4 Accessing a Particular Key

`int count(const void* key)`
returns the number of values associated with the `key`. Zero is returned when there are no values associated with the key, and a negative number is returned if the key does not exist.

`int lookup(const void *key, void *value)`
copies to the space pointed by `value` the very first value associated with `key`. A positive number is returned when the key cannot be located, and a negative number is returned when a failure occurs. Zero is returned when the `key` is found.

5.11.5 Accessing Index Properties

`int nbuckets(void)`
returns the number of buckets belonging to `this` extendible hash.

`int depth(void) const`
returns the depth of the extendible hash directory. The value returned indicates the number of bits used to distinguish the keys stored in the index.

`const char* name(void) const`
returns the name, if any, associated with the index pointed by `this`, or 0 if the index has no name.

`int name_set(const char* name)`
sets the name of the index to `name`. If the name already exists or an error occurs, non-zero is returned.

`int name_remove(void)`
removes the name of the index. It returns zero on success, non-zero on failure.

`const eosoid& oid(void) const`
returns the object id of the index directory.

`const eosoid& parentoid(void) const`
returns the object id of the file containing the index directory.

`const eosoid& rootoid(void) const`
returns the object id of the database's root file containing the index directory.

5.12 Index Scan (`class eosehashscan`)

EOS provides facilities for accessing all the keys and their accosiated values in an index or all the values associated with a particular key only. The state of an index scan is recorded by a cursor which points to the "current" <key, value> pair being scanned.

`static eosehashscan* open(eosehash* eh, int order=AUTO_FWRD, const void* key=0)`
opens a scan for the index pointed by `eh`. It returns 0 on failure. If the value of `key` is not NULL then the scan is opened for this particular key. `order` specifies the order in which the <key, value> pairs of the index will be scanned. It can take one of the following values:

`eosehashscan::AUTO_FWRD`	Automatic scan of the index in forward order. Initialy, the cursor is positioned at the first pair of the index if `key` is NULL; otherwise, it is positioned at the first pair whose key is `key`.
`eosehashscan::AUTO_BWRD`	Automatic scan of the index in backward order. Initialy, the cursor is positioned at the last pair of the index if `key` is NULL; otherwise, it is positioned at the last pair whose key is `key`.
`eosehashscan::MANUAL`	Explicit movement of the cursor.

`int cursor(int same_key=0)`
returns true if the cursor currently points to a valid <key, value> pair. If the scan order is `eosehashscan::AUTO_FWRD` or `eosehashscan::AUTO_BWRD` the cursor moves to the next or previous pair, respectively. If the value of `same_key` is true, the cursor is restricted to visit only those pairs whose key is the same as the key of the currently pointed pair. In other words, if `same_key` is always true, the cursor will visit all pairs with a particular key. The cursor will remain unchanged if the scan order is `eosehashscan::MANUAL`.

`const void* key(void)`
returns a pointer to the key pointed by the cursor. Zero is returned when the cursor does not point to a valid <key, value> pair.

`const void* value(void)`
returns a pointer to the values pointed by the cursor. Zero is returned when the cursor does not point to a valid <key, value> pair.

`int remove(void)`
removes the value curently pointed by the cursor. If this is the only one value associated with the key then the key is removed too. It returns zero on success, non-zero on failure.

`int replace_value(const void* new_value)`
replaces the value that is currently being scanned with `new_value`. Zero is returned on success and non-zero if no value is being scanned or an internal error occurs.

`int close(void)`
closes an open scan. It returns zero on success, non-zero on failure.

When the `eosehashscan::MANUAL` order is specified, the cursor must be moved explicitly by using the following functions. They all return zero on success, non-zero on failure. Note that it is NOT an error to seek to the first, next, previous, or last <key, value> pair when no such pair exists in the index. In this case, a call to `cursor()` will simply return false.

`int first(int same_key=0)`
positions the cursor to the first value of the current key, if the value of `same_key` is true, or to the first value of the first key in the index, if its value is false.

`int last(int same_key=0)`
positions the cursor to the last value of the current key, if the value of `same_key` is true, or to the last value of the last key in the index, it its value is false.

`int next(int same_key=0)`
positions the cursor to the next value of the current key. If the value of `same_key` is false and there are no other values for the same key, then it positions the cursor to the first value of the next key in the index.

`int prev(int same_key=0)`
positions the cursor to the previous value of the current key. If the value of `same_key` is false and there are no other values for the same key, then it positions the cursor to the last value of the previous key in the index.

`int seek_at(const void* key)`
positions the cursor on either the first or the last value associated with `key` when the order specified in `eosehash::open()` was `eosehashscan::AUTO_FWRD` or `eosehashscan::AUTO_BWRD`, respectively. If the order was `eosehashscan::MANUAL` the cursor is not positioned at any value. It is an error to attempt to position the cursor on an <key, value> that is not valid, i.e. the `key` does not exist.

5.13 Extensions and Primitive Events (`class eosexten`)

5.13.1 Defining Hook Functions

`static int insert(int type,int when, int (*func)(eosstat *stat))`
registers the hook function `func` to be executed when the event type `type` occurs. The value of the `when` argument determines if the hook is called before or after the occurrence of this event and it can take one of two values;
`eosexten::BEFORE`: the hook is invoked just before the event occurs;
`eosexten::AFTER`: the hook is invoked just after the occurrence of the event.
On success, a non negative number is returned, called the *extension number*. On failure -1 is returned.

5.13.2 Return Values of User Functions

The user function `func` must return one of the following values:

`eosexten::FAIL`	The user function `func` failed. Then, the EOS component that called `func` fails too, and it returns either an integer $\neq 0$, or NULL depending on its interface.
`eosexten::CONTINUE`	The user function `func` successfully completed its operations. The flow of the EOS function that called `func` remains unchanged.
`eosexten::RETURN`	The user function `func` successfully completed its operations. The EOS function that called `func` skips the remaining steps and returns success immediately. This is translated to a returned value of 0 when the EOS function returns an integer, and the value of that member of the `eosstat` structure, used by the registered function, that has the same type as the EOS function prototype.

5.13.3 Argument Passed to User Functions

The `eosstat` structure has the following members:

```
typedef struct eosstat {
    eosdatabase* db;
    unsigned     create : 1;
    unsigned     trunc  : 1;
    unsigned     rdonly : 1;
    unsigned     normal : 1;
    char*        name;
    eosobj*      obj;
    eosfile*     file;
    eosfile*     pfile;
    eosoid       oid;
    int          flags;
    int          hint;
    int          size;
} eosstat;
```

5.13.4 Primitive Events

When a primitive event is trapped, members of the `eosstat` structure are initialized. The events that EOS traps and the members of the `eosstat` set for each event are described below. The members of the `eosstat` structure are set from the values of the corresponding arguments of the EOS function that called the registered function.

`TR_BEGIN`	Captured by the `eostrans::begin()`. Members set: `rdonly`.
`TR_COMMIT`	Captured by the `eostrans::commit()`. Members set: none.
`TR_ABORT`	Captured by the `eostrans::abort()`. Members set: `normal`.
`TR_DEADLOCK`	Captured in various levels of the EOS storage manager. Members set: none.
`DB_OPEN`	Captured by the `eosdatabase::open()`. Members set `BEFORE`: `db, rdonly, create, trunc` Members set `AFTER`: `db, rdonly, create, trunc` The `db` member of the `eosstat` structure is returned by the `eosdatabase::open()` when the user registered function returns `eosexten::RETURN`.
`DB_REMOVE`	Captured by the `eosdatabase::destroy()`. Members set `BEFORE`: `db` Members set `AFTER`: none
`FILE_CREATE`	Captured by the `eosfile::create()`. Members set `BEFORE`: `pfile, name, flags`. Members set `AFTER`: `file, pfile, name flags`. The `file` member of the `eosstat` structure is returned by the `eosfile::create()` when the user registered function returns `eosexten::RETURN`.
`FILE_OPEN`	Captured by the `eosfile::open()`. Members set `BEFORE`: `oid`. Members set `AFTER`: `file, oid`. The `file` member of the `eosstat` structure is returned by the `eosfile::open()` when the user registered function returns `eosexten::RETURN`.
`FILE_REMOVE`	Captured by the `eosfile::destroy()`. Members set `BEFORE`: `file` Members set `AFTER`: none.
`OBJECT_FAULT`	Captured by the `eosobj::get()`. Members set: `obj` Only the `eosexten::AFTER` can be specified for this event. The `obj` member of the `eosstat` structure is returned when by the `eosobj::get()` when the user registered function returns `eosexten::RETURN`.

`OBJECT_CREATE`	Captured by the `eosobj::create()`. Members set `BEFORE`: `pfile, oid, flags, size, hint` Members set `AFTER`: `obj, pfile, oid, flags, size, hint` The `obj` member of the `eosstat` structure is returned when by the `eosobj::create()` when the user registered function returns `eosexten::RETURN`.
`OBJECT_REMOVE`	Captured by the `eosobj::destroy()`. Members set `BEFORE`: `obj`. Members set `AFTER`: none.
`OBJECT_UPDATE`	Captured by a number of EOS functions when an object is going to be updated, or it is updated for the very first time. Member Set: `obj`.
`SLOTTED_PAGE_FAULT`	Captured by a number of function in the lower level of the EOS storage manager. Only the `eosexten::AFTER` can be specified. Member Set: `oid` The user registered function is called for every object in the page that just fetched in the local buffer.

5.13.5 Extension Activation Status

`static int alter(int type, int when, int extno, int on_off)`
activates and de-activates a registered function. The extension with number `extno` registered to be called `when` the event `type` occurs, is de-activated if the value of `on_off` is 0, and it is activated when `on_off` has the value 1. A registered action is by default active.

`static int is_active(int type, int when, int extno)`
checks whether the extension `extno` for the event `type`, `when` is active or not.

5.13.6 Example 1 – Access Control

This example demonstrates the use of EOS extensions to provide access control such as when certain databases should not be accessible to all users. Let us assume that the application keeps a table of databases and the users that have access to them and that the function `access_authorized(char* dbname, char* uname)` returns true if the user with name `uname` has access to database with name `dbname`.

We want to define an extension that is triggered just before a database is going to be opened. That is, the event that triggers this extension is `BEFORE DB_OPEN`. The first step is to write the hook we want to be executed when the above event occurs; its code may look as follows:

```
int check_db_access(eosstat *s) {
    if (! access_authorized(s->name, getlogin())) {
        printf("%s: permission denied for this database.\n", s->name);
        return eosexten::RETURN;
    }
    return eosexten::CONTINUE;
}
```

After the action is written, the second and final step is to register this extension with EOS as follows:

```
    eosexten::insert(eosexten::DB_OPEN, eosexten::BEFORE, check_db_access);
```

Before any attempt to open a database, EOS sets the `name` field of the `eosstat` structure to the name of the database and invokes the function `check_db_access`. In turn, the function checks whether the user is authorized to access the database with the given name. If the user has access to this database, the function returns `eosexten::CONTINUE`. This return value signals EOS to follow its normal flow of control, which is to go ahead and open the database. On the other hand, if the user does not have access to this database, the function prints an error message and then returns `eosexten::RETURN`. This return value instructs EOS to bypass its normal control flow, and return immediately with error, NULL in this case. Thus, the call to `eosdatabase::open` that triggered this action returns NULL – it is as if EOS could not open this database.

5.13.7 Example 2 – Fixing C++ Pointers

This example is taken from the implementation of Ode [BGL+93], a C++ based database system, that uses EOS as its storage manager. The problem encountered during the implementation was that C++ objects of types that have virtual functions or virtual base classes contain *hidden* pointers [BDG93] – pointers that they were not specified by the user. In the case of virtual functions, the hidden pointer points to a virtual function table that is used to determine which function is to be called. In the case of virtual base classes, the hidden pointers are used for sharing base classes [Str87]. Hidden pointers are invalid across program invocations and they need to be fixed. We register the hook function `fix_hidden_ptrs` to be execute right after an object fault occurs as follows:

```
    eosexten::insert(eosexten::OBJECT_FAULT, eosexten::AFTER, fix_hidden_ptrs);
```

The hook function `fix_hidden_ptrs` looks as follows:

```
int fix_hidden_ptrs(eosstat *s)
{
    if ( has_hidden_ptrs(s->obj->utag_get()) ) {
        s->obj->fetch_all();
        ...; // fix the pointers
    }
    return eosexten::CONTINUE;
}
```

6 The EOS Server

6.1 Server Startup

The EOS server is started up by executing the following command

```
eosserver [-h] [-v] [-c] [-l] [-r] [-b] [-t num] [-f secs] [-s service] [-B npages] [-E Mbytes]
          [-p port] [-J name] [-C name] [-P name] [-O num] [-R num] [-T num]
          [-L lock_protocol] [-e name] [-a name]
```

The meaning of each one of the available command line arguments is presented below.

`-h`	A brief explanation for each command line argument is displayed on the screen.
`-v`	Verbose mode. It displays messages regarding the progress of the server. In this way the behavior of the server can be studied.
`-c`	It turns off the lock manager. No locks will be acquired for any kind of access to the database.
`-l`	It turns off the log manager. No log records are generated.
`-r`	Starts the server without performing recovery.
`-b`	The server can run in the background if the `&` shell flag is used. The `-v` option is ignored in this case.
`-t` *num*	The maximum number of client programs that can be connected to the server at the same time.
`-f` *secs*	The frequency in seconds of the checkpoint process.
`-s` *service*	The name of the service offered as it is in the `/etc/services` file (if such service exists).
`-B` *npages*	The size of the buffer pool used by the server in terms of pages (4 Kbytes each).
`-E` *Kbytes*	The size of the extra space used by the server for large object manipulation because large objects do not get stored in the buffer pool.
`-p` *port*	The port where the server daemon listens for connection requests.
`-J` *name*	The rooted path of the global log file to be used.
`-C` *name*	The rooted path of the checkpoint file.
`-P` *name*	The directory that will contain the private log files. The directory must exists otherwise an error will be generated.
`-O` *num*	The maximum number of pages that can be locked at the same time by all active transactions communicating with the server.

`-R` *num*	The maximum number of lock requests, both granted and blocked, that can be outstanding at any point in time.
`-T` *num*	The maximum number of active transactions that can hold locks on database pages at the same time.
`-L` *lock_protocol*	The concurrency control protocol to be used. The available protocols are **2PL** and **2V2PL**. 2PL is the standard two-phase locking protocol. 2V2PL is the two-version two-phase locking protocol used by default in EOS.
`-e` *name*	The name of the file to be used for reporting errors. If not given the `stderr` is used.
`-a` *name*	The directory that will contain the archived private log files. If the directory does not exist an error will be reported.

6.2 Normal Operation

Once the server has started up it monitors the standard input (`stdin`) for user entered commands. The commands that are recognized are given below.

`help`	A brief description of all the available commands is given.
`version`	The current EOS version number is printed on the screen.
`checkpoint`	Take a checkpoint now.
`debug`	The values of all the parameters used by the server are printed on the standard output. In addition, information about the active processes spawned by the server is given.
`stats`	A number of statistics about the server buffer and transaction modules of EOS are printed on the screen.
`shutdown`	The server will be shut down after aborting all active transactions. A checkpoint is taken so that no recovery is needed when the server starts up again.
`exit`	Exit immediately.

6.3 Server Shutdown

To shutdown the server when it runs in the background or from a remote machine, run `eosserverkill`. The invocation of the above program is done in the following way:

`eosserverkill` [`-c`] [`-a`] [`-w`] [`-f`] [*host*]

The meaning of the available options is explained below.

host The name of the machine where the server runs.

-c Take a checkpoint and then exit immediately.

-a Abort all active transactions present in the system and exit.

-w Wait until all active transactions finish (either commit or abort) and then exit. No new transaction will be started in the interim.

-f Flush the shared buffer pool before exiting.

The `-a` and `-w` options are conflicting and the `-a` will be used when both are present. If the `eosserverkill` is invoked with no options, the server will exit after aborting all active transactions. The current release does not verify the permissions of the person invoking the `eosserverkill` program.

The server removes all shared memory segments and semaphores used when it exits. However, there might be cases where there are leftover shared memory segments and semaphores; e.g., if the server is killed by the `kill -9` command. Because the server will not operate properly if these leftovers are not removed, use the `ipcrm` system command to remove them. The `ipcs` command is used to check if there are any shared memory segments and semaphores left.

6.4 Checkpoint

The frequency of the checkpoint procedure is set by the `-f` command line option of the `eosserver` command. In addition, the `eoscpfrequency` utility program can be used to change the checkpoint frequency while the server is running. Its usage is as follows:

`eoscpfrequency` [*host*] *secs*

The time period between two successive checkpoint requests is given by the *secs* argument passed to the above program. If the *host* is not given, then the host given by the EOS_SERVER_HOST_NAME variable will be contacted. Currently, there is no limit on the upper bound of the checkpoint period. This might be changed.

6.5 Is Alive

The command `eosserveralive` checks if the EOS server is alive:

`eosserveralive` [*host*]

If the *host* is given, then this particular host will be contacted. Otherwise, the host name will be taken from the EOS_SERVER_HOST_NAME environment variable, see section 7.3.

If the EOS server is active then the

EOS server is alive and healthy.

message will appear on your screen. Otherwise, the

No EOS server is running on 'host'.

will appear.

7 EOS Customization

The client and the server modules of EOS, as well as the one that formats an area use a number of environment variables to perform their task. EOS provides three installation programs that set up default values for all these variables so users that do not want to customize EOS or they are not concerned with performance need only change few things to set up their environment. The default values of these variables are stored in three configuration files named `formatrc`, `clientrc` and `serverrc` under the `$HOME/.eos` directory of the user making the installation.

The following are the three programs that install the default values of the environment variables used by EOS.

- `eosformatenv` creates and initializes the configuration file `formatrc`; this file is used whenever a new area is formatted with the `eosareaformat` command.
- `eosclientenv` creates and initializes the configuration file `clientrc`; this file is used by the applications linked with the `private.o` or `client.o` object module of EOS.
- `eosserverenv` creates and initializes the configuration file `serverrc`; this file is used by the EOS server.

The configuration files contain *name, value* pairs, one per line, with the equal sign (`=`) separating the value from the name. There are two ways to change these default values:

1. Use an editor to open the configuration file and locate the parameter whose value is going to be updated; update the value making sure that you do not delete the `=` sign or modify the name of the parameter.
2. Set environment variables whose names are identical to the names found in the configuration files. For csh or tcsh users, this can be done as follows:

   `setenv` *name value*

   For korn shell users, this can be done as follows:

   name `=` *value*; `export` *name*

For a given name, environment variables are checked first and if no such variable is set, the value of the name is taken from the appropriate configuration file. For the server, there is a third way of initializing the parameters by using command line arguments when the server is invoked. Section 6.1 elaborates on this.

7.1 Customizing the Area Formatting Procedure (`formatrc`)

The formation of an area is based on the following environment variables.

EOS_AREA_EXTENT_SIZE: The default extent size of the area being formatted. If possible, the size of the extent should be set to the number of physically contiguous pages available in the disk device the area resides on. In this ways, the number of disk seeks required for accessing large objects is minimized.

EOS_AREA_NEXTENTS: The default number of extents of an area being formatted.

EOS_MAX_DBS_PER_AREA: The maximum number of databases that can be created in a given EOS area.

EOS_AREA_EXPANDABLE: This variable specifies whether the area being formatted is expandable; i.e., whether a new extent should be appended to the existing area if more space is requested. This variable applies only to storage areas that are UNIX files.

EOS_AREA_HOST_NAME: The default host machine name the area manager is running on.

EOS_AREA_HOST_PORT: The port number used for communication with the area manager. It must be different than the EOS_SERVER_HOST_PORT number in the `clientrc` file.

7.2 Customizing the Client (`clientrc`)

The following are the environment variables for the client program.

EOS_POOL_SIZE: The maximum size of the transaction's buffer pool measured in pages (4K-bytes each). The cache is created incrementally in chunks of EOS_POOL_INCREMENTS pages. The maximum size of the client cache affects the performance of the client program. A small cache will force the EOS client cache manager to force pages out of the cache sooner than a larger cache. On the other hand, a very large cache will increase the operating system's swapping activity due to memory size limitations and activities from other users running on the same client machine. It's hard to come up with a right size for all applications a user may run. As a general suggestion, keep this number large – e.g., 16 or 32 megabytes – and then experiment with the EOS_POOL_INCREMENTS value.

EOS_POOL_INCREMENTS: The number of frames allocated to the transaction's cache each time more frames are needed – until the cache has reached its final size.

EOS_LO_SEG_THRESHOLD: It affects large objects only. It specifies the default segment size threshold used when updates are performed on large objects, such as when inserting or deleting a number of bytes starting at an arbitrary offset within the large object. See section 2.1.6 for the effects of this value.

EOS_OH_INCREMENTS: The number of object handles that will be allocated when new ones are needed. There is no explicit upper bound on the number of objects fetched. However, such limit is implicitly imposed by the number of object present in the pages the cache can fit, as specified by the EOS_POOL_SIZE variable.

EOS_EM_MAX_ACTIONS: The maximum number of hook functions that can be registered for execution for a particular event. For example, if this value is 4, you can register up to 4 actions to be executed before the event occurs and up to 4 actions to be executed after the event occurs.

EOS_SERVER_HOST_NAME: The host machine name the server is running on. This name is used when no host name is provided while opening, creating, or deleting a database.

EOS_SERVER_HOST_PORT: The port name through which communication with the server will take place. This value must match the corresponding value set at the server's environment or the value in the `/etc/services` entry when the service provided by the server is known.

7.3 Customizing the Server (`serverrc`)

The following are the environment variables used by the EOS server.

EOS_SERVER_HOST_PORT: The port number where the EOS server listens for connection requests. This number must be well known to all the clients interested in establishing a connection. One way of achieving this is by having an entry in the `/etc/services` file. If this is not possible, then try to use a port number between 1024 and 5000 (for the Internet).

EOS_REPORT_PROGRESS: If its value is not 0 then most of the steps followed in each operation carried out by the server will be displayed on the screen. If the value is 0 only error conditions are reported.

EOS_LOCK_IS_ENABLED: If its value is not 0 then the concurrency control module will be active and locks are acquired while accessing the database. Otherwise, no locks are placed of the pages accessed. The default value is 1.

NOTE: no guarantee is given that the system will operate in a consistent and correct way when the lock manager is turned off.

EOS_LOG_IS_ENABLED: If its value is not 0 then all the updates performed by committed transactions will be logged on durable storage. If the value is 0, no logging is performed.

NOTE: no guarantee is given that the system will be in a consistent state should a failure occur and the log manager is off.

EOS_AUTO_RESTART: If its value is not 0, when the server starts it makes sure that all updates performed by all committed transactions in the past are in the database. If the value is 0, the servers does not check wether the database is consistent.

NOTE: no guarantee is given that the system will behave correctly in the case where a failure occurred and the automatic restart is turned off.

EOS_MAX_CONNECTIONS: The maximum number of open connections at any point in time.

EOS_CHECKPOINT_FREQUENCY: The frequency (in seconds) of the checkpoint request issued by the server. The time interval between two checkpoints is of importance to the recovery procedure during system restart. The default value is set to 600 seconds.

EOS_SHARED_POOL_SIZE: The number of 4K-byte pages of the server's buffer pool.

EOS_LARGE_OBJ_SIZE: The size in Kbytes of the extra space the server allocates for large object disk I/Os.

EOS_SHARED_POOL_FILE_PATH: The rooted path name of the file that will be used to store the server's buffer pool as well as the extra space allocated for large object I/Os. This file is memory mapped (`mmap()`) by the server and the disk daemon during start up. The default path is `/tmp/_eos_srv_pool`. If the file does not exist it will be created automatically.

EOS_CONTROL_FILE_PATH: The rooted path name of the file that will be used to store all control structures required by the server's buffer manager. This file is memory mapped (`mmap()`) by the server during start up. If the file does not exist it will be created automatically. The default path is `/dev/zero`.

EOS_CHECKPOINT_NAME: The rooted path name of the checkpoint file.

EOS_GLOBAL_LOG_NAME: The rooted path name of the global log file.

EOS_PRIVATE_LOG_DIR: The rooted path name of the directory used to store private log files. All the log records generated by a transaction are stored in a file, whose name has the form `eos.priv_x_y_z`, under the above directory.

EOS_ARCHIVE_DIR_PATH: The rooted path name of the directory used to store the archived private log files. The name of an archive file has the form `eos_arch_N.Z`, where `N` refers to the order of creation and it is a monotonically increasing number.

EOS_LOG_THRESHOLD: The minimum number of private log files that have to be created before the archive procedure is activated. The value of this variable depends on the expected size of the log files created. If the log files are small then a large number (hundreds) is adequate. On the other hand, if the log files created are big, a small value (tens) is recommended.

EOS_MAX_LOCK_UNITS: The maximum number of database pages that can be locked at the same time.

EOS_MAX_LOCK_ENTRIES: The maximum number of locks held by all transactions present in the system on different database pages.

EOS_MAX_TRANSACTIONS: The maximum number of active transactions that have at least one lock entry at the same time.

EOS_SRV_SND_BUFFER: The size in kilobytes of the operating system's buffer used to send data over a TCP/IP connection. This value should not exceed the upper bound imposed by the kernel. The default value is 17K.

EOS_SRV_RCV_BUFFER: The size in kilobytes of the operating system's buffer used to receive data over a TCP/IP connection. This value should not exceed the upper bound imposed by the kernel. The default value is 17K.

EOS_TR_FAIL_TIMEOUT: The maximum allowable "idle period" for an application process expressed in minutes. If the application does not have any interaction with the server during the above time period, then the server unconditionally aborts the transaction.

8 EOS File System Viewer (eosfsview)

`eosfsview` is a primitive – and not yet complete – interactive browser that shows information about an EOS storage area, databases, files and objects. The program accepts one argument, the name of the area to be viewed.

When you run `eosfsview` you see this menu:

```
                        EOS Release x.y.z

Options: 0 - quit
         1 - area info, 2 - list databases
         4 - make db,   5 - open db,   6 - rm db, 7 - rename db
Command:
```

Option 2 displays all databases in the area. After a database is opened with the option 5, the following menu is displayed:

```
Options: 0) quit, 1) go to main menu
         2) ls,   3) cd
         20) show names, 21) unix od
Command [2]:
```

You may then list all objects in the current file (option 2), change current file (option 3), show all names in the database (option 20), and apply the Unix `od` command to see the contents of a byte range of an object. When option 2 is chosen, the properties of objects within the current file are displayed as follows:

```
----v-h       20  00008001-0002-ce-001
----v-h       20  00008002-0002-ce-001
--ln---     5000  00007ffe-0002-73-001 my-large-obj
---n---       40  000070fe-0002-a2-001 my-small-obj
-------       40  000070fe-0002-4c-002
-f-nv--       20  00008802-0002-02-001 my-file
```

The set of dashes and characters refers to the object properties interpreted as follows:

```
r        root file object
 f       file object
  l      large object
   n     named object
    v    variable length object
     p   page object
      h  hash object
```

The object size (in bytes) is displayed next, followed by its oid (page no, area no, unique no, slot no), followed by the object's name, if any.

9 An Example of Using EOS

This section presents two simple programs included in the EOS distibution – the source code of these programs is in `eos/example` directory. The first program creates a number of objects that are linked together by using a number of persistent references. The second programs, traverses the object hierarchy down to the level which is specified in the command line argument. A number of other demo programs can be found in the smae directory.

9.1 File `Part.h`

```
#ifndef __Part_h__
#define __Part_h__

#include "eos_Ref.h"
#include "eos.h"

#define NPARTS    200
#define NAME_SIZE       128

struct Part {

   int part_no;                // The part number
   eos_Ref<char> name;         // A name given to this part
   eos_Ref<Part> x_part;       // Persistent reference to another part
   eos_Ref<Part> y_part;       // Persistent reference to another part
   eos_Ref<Part> z_part;       // Persistent reference to another part

};

#endif
```

9.2 File `part_create.c`

```
// Usage: create <database-name>
// This programs creates a hierarchy of objects

#include <stdio.h>
#include <stdlib.h>
#include "Part.h"

#define ErrR(msg)       { printf msg; return -1; }

int create_parts(eosdatabase *db)
{
  eos_Ref< eos_Ref<Part> > pers_array;
  Part* a_part;
  // Create the array of pointers to parts and give it a name

  pers_array = new(db) eos_Ref<Part>[NPARTS];

  if ( pers_array.is_null() )
     ErrR(("Failed to create the persistent array of references to Parts."));

  if ( db->set_object_name(pers_array, "part_array") )
     ErrR(("Cannot set the name to %s", "part_array"));

  // Create the part objects and store references to them in the pers_array

  for ( int i=0; i<NPARTS; i++) {

     pers_array[i] = new(db) Part;

     if ( pers_array[i].is_null() )
        ErrR(("Failed to create a persistent Part."));

     a_part    = pers_array[i];
     a_part->name = new(db) char[NAME_SIZE];

     if ( a_part->name.is_null() )
        ErrR(("Failed to create the persistent name of a Part."));


     sprintf( (char *) a_part->name, "Part Object With Number %d", i);
     a_part->part_no = i;
  }

  // Link the object together

  for ( int j = 0; j < NPARTS; j++ ) {
      a_part         = pers_array[j];
      a_part->x_part = pers_array[ rand() % NPARTS];
      a_part->y_part = pers_array[ rand() % NPARTS];
      a_part->z_part = pers_array[ rand() % NPARTS];
  }

  return 0;
}

///////////////////////////////////

main(int argc, char** argv)
{
  eosdatabase *db;

  if ( argc != 2 )
     printf("Usage: %s database_name \n", argv[0]), exit (-1);
  if ((db = eosdatabase::open(argv[1], 0, 1)) == NULL )
     printf("Cannot create database %s\n", argv[1]), exit (-2);

  if ( eostrans::begin(0) )
     printf("Cannot start a transaction\n"), exit (-3);

  if ( create_parts(db) )
     printf("Cannot create the objects.\n"), eostrans::abort(), exit(-1);

  if ( eostrans::commit() )
     printf("Transaction commit failed.\n"), eostrans::abort(), exit (-5);

  if ( db->close() )
     printf("Cannot close the opened database\n"), exit (-6);

  exit(0);
}
```

9.3 File part_traverse.c

```
// Usage:  traverse <database-name> <level of traversal>
// This program starts from a random object and traverses the object
// hierarchy down to the level given in the input.

#include <stdio.h>
#include <stdlib.h>
#include "Part.h"

// The depth-first traversal of the object hierarchy

void traverse(Part* p, int level)
{
  if ( level == 0 ) { printf(" ----- \n"); return; }

  printf("Part: %d with name %s\n", p->part_no, (char *) p->name);
  printf("\tVisiting x_part: "); traverse(p->x_part, level-1);
  printf("\tVisiting y_part: "); traverse(p->y_part, level-1);
  printf("\tVisiting z_part: "); traverse(p->z_part, level-1);

  return;
}

///////////////////////////////////

main(int argc, char **argv)
{
  eosdatabase *db;
  eos_Ref< eos_Ref<Part> > p_array;
  if ( argc != 3 )
     printf("Usage: %s database level \n", argv[0]), exit ( -1);

  if ((db = eosdatabase::open(argv[1])) == NULL )
     printf("Cannot open database %s\n", argv[1]), exit (-2);

  if ( eostrans::begin(0) )
     printf("Cannot start a transaction\n"), exit (-3);

  p_array = (eos_Ref< eos_Ref<Part> >) db->lookup_object("part_array");

  if ( p_array.is_null() )
     printf("Failed to locate the object\n"), eostrans::abort(), exit (-4);

  Part* a_part = p_array[ rand() % NPARTS ];
  traverse(a_part, (int) atoi(argv[2]) + 1);

  if ( eostrans::commit() )
     printf("Transaction commit failed.\n"), eostrans::abort(), exit (-5);

  if ( db->close() )
     printf("Cannot close the opened database\n"), exit (-6);

  exit (0);
}
```

10 Troubleshutting

`Trying to connect .........` The server was not able to establish the communication link because the port number it uses is busy. If this happens, the server will re-try. If the server does not succeed after 5 tries it gives up.
Solution: change the port number the server uses by altering in both the `clientrc` and `serverrc` configuration files the value of the EOS_SERVER_HOST_PORT. The values must be the same in both files and it must be different than the EOS_AREA_HOST_PORT value in the `formatrc` file.
Note: if the above message appears after the message *EOS server. We are open for business.* then the problem is with the disk daemon. In this case, shutdown the server and try again. If the problem persists, then change the EOS_AREA_HOST_PORT value.

`Cannot create shared memory` This may happen when:

1. The machine where the server started on does not support shared memory.

   Solution: ask the system administrator to install shared memory on this machine.

2. The number of the available shared memory segments is not enough to satisfy the server's requirements.

   Solution: decrement the EOS_SHARED_POOL_SIZE value.

3. A conflict was created because the key used to create a shared memory segment has been already used by someone else.

   Solution: remove any shared memory segments used by other processes because some of the keys used for creating shared memory are fixed.

4. The server was shut down and was not able to cleanup the shared memory segments it used.

   Solution: check if there are any shared memory segments left by using the `ipcs -m` command. If there are some, remove them by using the `ipcrm` command.

`Cannot create semaphore` This may happen when:

1. The machine where the server started on does not support semaphores.

   Solution: ask the system administrator to install semaphores on this machine.

2. The number of the available semaphores is not enough to satisfy the server's requirements.

   Solution: ask the system administrator to increase the number of the semaphores that can be created system-wise.

3. A conflict was created because the key used to create a semaphore has been already used by someone else.

   Solution: remove any semaphores used by other processes because some of the keys used for creating semaphores are fixed.

4. The server was shut down and was not able to cleanup the semaphores it used.

   Solution: Check if there are any semaphores left by using the `ipcs -s` command. If there are some, remove them by using the `ipcrm` command.

`Error -1302 occurred on the EOS server.` This may happen when the application tries to access a storage area that is not known to the server. This can be a result of:

1. Name misspelling.
2. Creation of the area with the `-l` flag while the server was running.
3. The application was connected to the wrong server.

`Error -1701 occurred on the EOS server` This may happen when the maximum value for either of `EOS_MAX_LOCK_UNITS`, `EOS_MAX_LOCK_ENTRIES` and `EOS_MAX_TRANSACTIONS` is exceeded. Either increase their values in the `serverc` configuration file, or use larger numbers when you use the command line arguments when starting up the server.

11 Release Notes

EOS 2.2.0 versus EOS 2.1.0:

- A parametarized class to be used as a persistent reference to an object has been added. This class obeys the ODMG-93 [Cat93] standards.
- Persistent object can also be created by using the overloaded operator `new` which takes one argument that specifies where the new persistent object is going to be placed.

EOS 2.1.0 versus EOS 2.0.x:

- Reported bugs have been fixed and a number of changes have been made to increase performance.
- The system now runs on SGI, Suns 4.1.x, and Solaris 2.x architectures. C++ compilers that can be used include the ones distributed by AT&T, Sun, GNU, and CenterLine.
- The server can now run in the background. Use `-b` flag and the `&` shell flag. See Section 6.1.
- Error messages of the server can be stored in a user specified file. Use the `-e` flag. See Section 6.1.
- The archived private log files can now be stored in a user specified directory by using the `-a` flag at start up time. See Section 6.1.
- The `eosareadelete` invalidates all the pages belonging to the deleted area that are present in the buffer pool of the server.

- The `eosareaformat` invalidates all the pages present in the server's buffer pool that belong to the area being formatted when the area exists already.
- Abnormal client failures are detected by the server by using a timeout period as given by the user-specified value of EOS_TR_FAIL_TIMEOUT.
- `eofsview` is a primitive – and not yet complete – interactive browser that shows information about an EOS storage area, databases, files and objects. See section 8.

References

[BDG93] A. Biliris, S. Dar, and N. Gehani. Making C++ objects persistent: The hidden pointers. *Software Practice and Experience*, 23(12):1285 – 1303, December 1993.

[BGL$^+$93] A. Biliris, N. Gehani, D. Lieuwen, E. Panagos, and T. Roycraft. Ode 2.0 User's Manual. Technical report, AT&T Bell Laboratories, 1993.

[Bil92a] 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.

[Bil92b] 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.

[Cat93] R.G.G. Cattell. *Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, California, 1993. Contributions by T. Atwood, J. Dubl, G. Ferran, M. Loomis, and Wade, D.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, California, 1993.

[KP84] B. Kernighan and R. Pike. *The UNIX Programming Environment.* Prentice-Hall Software Series, 1984.

[Ste90] R. Stevens. *UNIX Network Programming.* Prentice-Hall Software Series, 1990.

[Str87] B. Stroustrup. *C++ Programming Language.* Addison-Wesley, Reading, MA, 1987. 2nd ed.