Using the EXODUS Storage Manager V3.1

(Last revision: November, 1993)

The Exodus software was developed primarily with funds provided by by the Defense Advanced Research Projects Agency under contracts N00014-85-K-0788, N00014-88-K-0303, and DAABO7-92-C-Q508 and monitored by the US Army Research Laboratory. Additional support was provided by Texas Instruments, Digital Equipment Corporation, and Apple Computer.

1. INTRODUCTION

The EXODUS Storage Manager is a multi-user object storage system supporting versions, indexes, single-site transactions, distributed transactions, concurrency control, and recovery. This document provides information about using version 3.1 of the EXODUS Storage Manager. Information about installing the Storage Manager can be found in the *EXODUS Storage Manager Installation Manual*. Section 2 gives an overview of the system. Section 3 discusses configuration facilities. Section 4 describes, in detail, the Storage Manager's application interface. Section 5 describes how to use the Storage Manager server. Appendices provide more details on certain aspects of the system. A table of contents is located at the end of the document.

2. OVERVIEW OF THE EXODUS STORAGE MANAGER

This section, an executive summary, briefly describes the architecture of the Storage Manager and gives an overview of the facilities provided to applications,

Version 3.1 of the Storage Manager runs on the following architectures: Sun 4 (Sparc) (under SunOS 4.1.[23]), DecStation 3100/5000 (MIPS) (under Ultrix 4.2), and HP 720 (under HP-UX A.08.07). The Storage Manager is written in C++ and had been checked for compilation under the GNU C++ compiler (g++), version 2.3.3 and 2.4.5.

2.1. Architecture

The EXODUS Storage Manager has a client-server architecture. An application program that uses the Storage Manager may reside on a machine different from the machine or machines on which the Storage Manager server or servers run. We use the term *application* to refer to programs that use the Storage Manager through the client programming interface described in Section 4. We use the term *client library*, or *client*, to refer to the Storage Manager code and data structures that are linked into the application program to support the client programming interface. The client allows applications to use the facilities described in the next sub-section. Each client has its own buffer pool for caching data. The client library connects to one or more server processes and communicates with them using a remote-procedure-call-style mechanism that runs over TCP.

The Storage Manager server is a multi-threaded process providing asynchronous I/O, file, transaction, concurrency control, and recovery services to multiple clients. The server stores all data on *volumes*, which are either Unix files or raw disk partitions. The server is more completely described in Section 5 and in the *EXODUS Storage Manager Architecture Overview* [exoArch].

2.2. Facilities

The EXODUS Storage Manager provides *objects* for storing data, *versions* of objects, *files* for grouping related objects, and *indexes* for supporting efficient object access. The Storage Manager also provides *volumes*, *transactions*, *concurrency control*, *recovery*, and *configuration options*. These facilities are presented briefly in this section, and more information can be found in later sections of the document.

2.2.1. Objects

An object is an uninterpreted container of bytes, which can range in size from a few bytes to a little less than the size of a disk. Internally, the Storage Manager distinguishes two types of objects. There are *small objects*, which are objects that fit on a single disk page, and *large* *objects*, which are objects that do not fit on a single disk page. Support is also provided for creating and manipulating versions of both small and large objects. To provide a uniform function call interface, the distinction between small, large, and versioned objects is hidden from applications. Applications are unaware of whether they are dealing with a small or large object, and the same interface functions are called to manipulate either type of object. To simplify the task of manipulating very large objects, the Storage Manager provides flexible buffer management that allows variable-length pieces of large objects to be buffered contiguously in the client buffer pool.

Objects have object identifiers. The object identifier of a small object points directly to the object on disk, while the object identifier of a large object points to a *large object header*. The header of a large object serves as the root of a B^+tree index structure that is used to access the object's data [Care86, Care89]. For space efficiency, a large object header can share a disk page with small objects and other large object headers. The data pages and the pages that make up the index structure of a large object are not shared, however. When a small object grows to the point where it can no longer be stored on a single page, the Storage Manager automatically converts it to a large object, leaving the new header in place of the original object.

The Storage Manager provides functions to read, overwrite, insert, delete, and append to an object. Read requests specify an object identifier and a range of bytes. The desired data is read into a contiguous region in the client buffer pool (even if is distributed over several disk pages), and a pointer to the data is returned to the caller. The overwrite function uses the pointer set up by a read request, and overwrites a subrange of the data. The insert and delete functions allow data to be inserted into and deleted from objects at arbitrary offsets, while the append function allows data to be appended to the end of an object. As mentioned earlier, large objects are represented using a B^+tree index structure. This ensures that each of the above operations can be executed efficiently on large objects.

2.2.2. Versions

A version of an object is another object that appears to be a copy of the original object. A version of a small object is a copy of the original object. A version of a large object is an object header with a pointer into the original object's data, until either the version or the original object is updated. When the large object version is updated, the affected portions of the original object are copied to prevent the original object from being affected by the update [Care89]. Although the version support described here is primitive, essentially providing "copy-on-write" objects, it has been purposefully designed that way so that a variety of application-specific versioning schemes can be implemented on top of the Storage Manager.

2.2.3. Files

Objects are allocated in *files*, which are collections of related objects. Files have three uses.

First, files are used for clustering objects. The objects in a file are stored on disk pages allocated solely to that file, so files provide a way to physically co-locate related objects on the disk.

Second, the Storage Manager provides an efficient way to *scan* the objects in a file, visiting each object exactly once.

Third, the Storage Manager offers an efficient mechanism for loading the objects into a file in bulk.

2.2.4. Indexes

The Storage Manager provides B^+tree indexes and linear hashing indexes. Index keys can be any basic C language data type or strings. Values can be any type of fixed length.

2.2.5. Volumes

User data and Storage Manager meta-data (objects, files, indexes, and logs) are stored on volumes. A volume represents a disk, although in fact it may be a Unix raw disk partition or a Unix file.

Volumes can be *temporary*, which means that data stored on them are not logged, and they do not persist from one transaction to the next. Temporary volumes are meant to provide fast storage for temporary data.

2.2.6. Transactions

A transaction is a set of operations on objects, files, and indexes. Transactions are either committed or aborted. Updates made by committed transactions are guaranteed to be reflected on stable storage, even in the event of software or processor failure. Updates made by aborted transactions are not reflected on stable storage.

Transactions that use data on more than one server are committed using a distributed two-phase commit protocol [Moha83].

2.2.7. Concurrency Control

Concurrency control allow multiple client applications safely to use data simultaneously. Concurrency control is based on the standard hierarchical two-phase locking protocol providing degree-three consistency (see [Gray78, Gray88]). The lock hierarchy contains two granularities: file-level, and page-level. Locking for index operations is performed with a non-two-phase protocol, which allows multiple clients to read and update the same index.

Deadlocks involving more than one server are resolved through timeouts.

2.2.8. Recovery

The Storage Manager recovers from software, operating system, and CPU failure by restoring data to a state in which all transactions have been committed or aborted. After an application fails, the transaction it is running is aborted by the servers that cooperated in the transaction. After a server fails and is restarted, updates made by committed transactions are restored, and updates by transactions in progress at the time of failure are undone. Recovery from media (disk) failure is not supported.

2.2.9. Configuration Options

The Storage Manager client library and servers have *configuration options*, which can be set by users. These options control such things as parameters that affect performance and memory use, formats of volumes and logs, the choice of servers to be contacted by clients, and path names of installed executable files.

2.3. Illustration of Using the Storage Manager

The purpose of this section is to give the reader a context in which to read the rest of this document. This section illustrates a way to get started using the Storage Manager. There are many ways to install, configure, and use the Storage Manager; only the simplest way is illustrated here.

This section uses an example application, “producer-consumer”. The source code for the application programs is included in the Storage Manager software release, along with other example applications.

The producer program generates a series of transactions, each of which creates an object. The consumer program generates a series of transactions, each of which reads an object and destroys it. These programs were selected because they are relatively small, demonstrate the use of transactions, and show how to respond to server-initiated transaction failures and server failures.

The remainder of this section gives specific directions for starting a server and running the example program. Detailed explanations of the steps are not given here; all the details are given elsewhere in this document.

Installing the storage manager is akin to installing an operating system or a remote file system (but it’s much simpler). You need to:

(1) install the system’s executable code, libraries, and include files;

(2) prepare your disks for use;

(3) configure your server so that it will use your disks, and so that it is otherwise tailored for your use;

(4) compile and link your application programs to use the installed system;

(5) configure your application programs’ environment, run the programs, and

(6) when you are finished, shut the system down.

2.3.1. Files Needed

The following files are needed to use the Storage Manager:

(1) `libsm_client.a`, the Storage Manager client library,

(2) `sm_client.h`, the include file containing declarations of key data structures and constants,

(3) `sm_server`, the executable file for the server portion of the Storage Manager,

(4) `diskrw`, the executable file for the disk I/O processes used by the server process,

(5) `formatvol`, and a utility program for formatting volumes,

(6) `.sm_config`, configuration files for a server, the formatter, and the application programs. One configuration file can be used for all programs, but it is sometimes easier to use configuration file for servers and the formatter, and another for applications.

These files can be installed anywhere; for the purpose of this section, we assume that they are all installed in your home directory, along with your application programs. (See the *EXODUS Storage Manager Installation Manual* to find the files in the Storage Manager software release.)

2.3.2. Preparing Your Disks

The producer and consumer programs use a volume for storing their objects with a single server, and the server uses a log volume. The `formatvol` program is used to format a volume for use as either a data volume or a log volume. If you plan to use a raw disk partition for either volume, ask your system administrator for information on how to set up the device.

The formats of the volumes must be described in the configuration file that `formatvol` reads. In the directory in which you plan to run `formatvol`, create a file called `.sm_config` that looks something like this, with the appropriate substitutions:

```
      formatvol*logformat:  /path/to/logfile: 9000: 1: 1: 1000: 8
      formatvol*dataformat: /path/to/datafile: 8000: 1: 1: 300
```

Substitute the pathnames for files that you want to use for your log volume and data volume. With the options given above, the log volume will be given a volume identifier of 9000, and will consist of 1 cylinder of 1 track each, with 1000 blocks on each track, hence, 1000 blocks will be on the log. The log volume will use 8 Kbyte log pages. The data volume will be given a volume identifier of 8000, and will consist of 1 cylinder of 1 track each, with 300 blocks on each track, hence, 300 blocks will be on the data volume.

Now, run the formatter on volumes 9000 and 8000:

```
      formatvol -vol 9000 -vol 8000
```

If you would like to see the information written on the volumes' headers, do this:

```
      formatvol -dis 9000 -dis 8000
```

The formatter prints:

```
VOLID 9000, version 3, is a LOG volume
   BLOCK SIZES: 8 K slotted, 8 K lg data,  8 K lg hdr
           8 K btree, 8 K idesc
   LAYOUT: 1000 blk/trk; 1 trk/cyl; 1 cyl
           1000 total blocks of 8 KB for 8192.000 KB
   FREE: 0 free, 1000 used
   BITMAP: 1 blk each, freemap @ 2, slotmap @ 4, filemap @ 5
   UNIQUE: start @ 3
   LOG: start @ 7, ctl blk @ 6, blk sz 8 K, #blks 993
                end of log @ dismount: LSN w=0.o=0, LRC w=0.c=1
VOLID 8000, version 3, is a DATA volume
   BLOCK SIZES: 8 K slotted, 8 K lg data,  8 K lg hdr
           8 K btree, 8 K idesc
   LAYOUT: 300 blk/trk; 1 trk/cyl; 1 cyl
           300 total blocks of 8 KB for 2457.600 KB
   FREE: 294 free, 6 used
   BITMAP: 1 blk each, freemap @ 2, slotmap @ 4, filemap @ 5
   UNIQUE: start @ 3
```

Now that you have formatted a log volume and a data volume, you are ready to start a server.

2.3.3. Configuring a Server

Before you start a server, you need to create its configuration file. In the directory in which you expect to run the server, create a file called `.sm_config` that looks something like this, with the appropriate substitutions (in particular, for each occurrence of `/path/to` below):

```
server*bufpages:    500
# Portname need not be identical to log volume id.
# This is just a convenience.
server*portname:    9000
server*diskproc:    /path/to/diskrw
server*logformat:    /path/to/logfile: 9000: 1: 1: 1000: 8
server*dataformat:   /path/to/datafile: 8000: 1: 1: 500
server*logvolume:    9000
```

If the same configuration file is to be used for the formatter and the server, the format options can be made to be recognized by both:

```
[sf]*[rl].logformat: /path/to/logfile: 9000: 1: 1: 1000: 8
[sf]*[rl].dataformat:/path/to/datafile: 8000: 1: 1: 500
```

Now you can start the server. Open a window in which to run the server, and, in the directory containing the server and its configuration file, start the server:

```
sm_server
```

The server is started on a newly formatted log volume, so it automatically regenerates the log. The server prints

```
Server is ready for requests.
```

when it can serve applications.

2.3.4. Compiling and Linking Your Application

An application program must include the header file `sm_client.h`, which, in turn includes `<stdio.h>`, `<setjmp.h>`, `<sys/types.h>`, `<netinet/in.h>`. Applications can be compiled with a C or C++ compiler.

The client library is compiled with C++, so client programs must be linked with a C++ compiler. See the *EXODUS Storage Manager Installation Manual* for more information.

2.3.5. Configuring and Running Your Application

The programs need configuration options to determine where to find the server that manages the data volumes they use, and to determine the sizes of the buffer pools they will use. In the directory in which you expect to run the application programs, Create a file called `.sm_config` that looks something like this, with the appropriate substitutions:

```
# both producer and consumer will use
# 250 page buffer pools:
client*bufpages:    250
# substitute the name or Internet address
# of the host on which the server runs:
client*mount:    8000    9000@serverhost
```

Now you can run the producer and the consumer. It is easiest to create a window in which to run each program. The produce and consumer programs use the environment variable EVOLID to determine the what volume to use. EVOLID must be set in each window.

In window P:

```
# producer <name> <#objects> <object size>
setenv EVOLID 8000
producer P 100 1000
```

In window C:

```
# consumer <name> <#objects>
setenv EVOLID 8000
consumer C 100
```

The producer creates “#objects” objects and writes “name” in each one. The “object size” argument is the size of each object. The consumer reads and destroys “#objects” objects. It prints the sizes of the objects and their names. The “name” given to the consumer program is immaterial, but is helpful for reading the output when running more than one consumer.

The two programs use a single root entry and a single file on the given volume. When a consumer has consumed the last object in a file, it destroys the file and removes the root entry. Each object is produced or consumed in a separate transaction. When both a producer and consumer are running concurrently, deadlocks occur periodically, since both are reading and writing the same file. When a deadlock occurs, the offending program aborts its transaction and tries again. Multiple producer and consumer programs may be started. If the server fails or shuts down, the producer and consumer programs attempt to reconnect every five seconds, and when successful, they continue transaction processing.

2.3.6. Shutting Down the Server

In the window in which the server runs, type the command:

```
shutdown
```

The server prints various messages, among them

```
Clean shutdown: no recovery required on any volumes.
All disk processes  killed.
```

when recovery is not required.

3. CONFIGURATION OPTIONS AND CONFIGURATION FILES

The client library, servers, and administrative programs use configuration options. All the options have a string name, a type, a set of possible values, a default value, and a current value. Client options can be set by a call to an application interface function or by a line in a *configuration file*. Server options can be set on the command line or by a line in the server's configuration file.

Configuration files are Unix files, and are similar in format to the X Window system's resource files. Each line in a configuration file is an *option command* or a *comment*.

A comment is a line that begins with "#" or with "!".

An option command is a line containing an *option descriptor*, white space, and a string representing a value to assign to the option. An option descriptor consists of an *option prefix* followed immediately by an option name and a ":".

The option prefix specifies the type and name of the program or programs for which the option is to be set. The program type is one of "client", "server", and "formatvol". The program name is usually the file name of the program, without its path (an application program can override this). The program type and program name are separated by ".". For example, the complete option descriptor for the option "bufpages" on the server named `serverA` is `server.serverA.bufpages:`.

Wild card characters are allowed in the program type and name. The character "*" represents any portion of the prefix. The "?" character represents any program type or any program name. The expressions describing the program type and the program name are parsed by a regular expression handler, so complex expressions can be used. See the manual page for regex(3).

The names of options can be abbreviated, as long as the abbreviation unambiguously identifies a single option. (This is also true for options appearing on command lines.) Program types and names may not be abbreviated. Option name, program type, and program name matches are case-sensitive.

Configuration options of type Boolean can be set with the Boolean values TRUE or FALSE, or with the strings "yes", "true", "no" or "false". The strings may be abbreviated and are not case-sensitive.

Each setting of an option overrides any previous value for that option.

Below, excerpts from configuration files illustrate ways to use the options.

```
# log volumes for two servers, whose executable
# file names are serverA and serverB
server.serverA.logvolume:          1000
server.serverB.logvolume:          2000
```

```
# turn off progress printing for all servers
server*progress:                 no
# or
server.?.progress:               no
```

```
! all servers and clients have a 1000 page buffer pool
*bufpages:                       1000
# The application foo uses a 500 page buffer pool.
# (overriding the value of 1000, above)
client.foo.bufpages:             500
# Applications beginning with the letter g use 400 pages
client.g*.bufpages:              400
```

4. THE STORAGE MANAGER APPLICATION INTERFACE

The Storage Manager's application interface consists of a set of functions, macros, and variables. The Storage Manager software release contains the header file `sm_client.h`, in which are found the definitions for the macros and types that appear in this document. Function prototypes for the the Storage Manager functions are also found in `sm_client.h`. By convention, words that appear capitalized in the text are either C-preprocessor macros, or C- or C++- defined types, Functions definitions appear in bold face in the text. The rest of this section is divided into sub-sections describing error handling, initialization and shutdown, transactions, buffer management, operations on objects, operations on versions, operations on files, operations on indexes, miscellaneous macros, and administrative functions.

4.1. Handling Errors

Error handling is important to users wishing to write robust client applications. We discuss it first, since most Storage Manager functions return error codes. Although this issue is complex, some of the burden is lightened by the recovery facilities of the Storage Manager. In this section we focus on error codes and error messages.

Almost all Storage Manager functions have integer return codes. All functions (except those used in printing error messages) return either esmNOERROR (zero), which represents success, or esmFAILURE (negative one), which represents an error. When an error occurs, the global variable sm_errno contains an error code. A small positive error code is an error code returned by Unix, as defined in `<errno.h>`. An error code greater than 65,536 is an error returned by the Storage Manager, as defined in `sm_client.h`. The Storage Manager error codes have symbolic names (C preprocessor macros) that begin with *esm*. **The value of sm_errno is not defined when the function returns esmNOERROR.**

Information about error codes can be obtained from the functions sm_Error( ), and sm_ErrorId( ), which are discussed below.

Some errors cause a message to be printed to the file addressed by `sm_ErrorStream`. By default, this file is the standard error file, stderr, as defined in `<stdio.h>`, but the application can change it any time after the Storage Manager is initialized.

Errors differ in severity and have different side effects. The most severe errors are fatal and cause the application to exit (the client library calls *exit(3)*). When the application exits, the servers abort the transaction, if a transaction is active. Fatal errors are caused by internal software problems in the Storage Manager. An example of a fatal error is esmMALLOCFAILED, which occurs when the entire data segment has been allocated by the application and client library, and the Storage Manager cannot proceed.

Less severe errors cause the transaction to be aborted, but leave the application running. When this happens, sm_errno is given the value esmTRANSABORTED, and the client library also sets the global variable `sm_reason`. The range of values for `sm_reason` is the same as the range of values for sm_errno. (The value of `sm_reason` is meaningful only if sm_errno has the value esmTRANSABORTED, and it is unpredictable and meaningless otherwise.) When the server or the client library aborts a transaction and returns esmTRANSABORTED to the application, the transaction is only partially aborted. The application **must** complete the termination of the transaction by calling sm_AbortTransaction( ) (described in the Section 4.3.3, **Transaction Operations**).

Less severe errors are generated by incorrect arguments to client interface functions or the lack of resources, such as buffer space. The application can correct the problem and retry the operation without aborting the transaction.

Finally, some error codes indicate conditions that are not errors at all, such as esmEMPTYFILE, which is returned when an empty file is read.

The following two functions can be used to print more information about the error.

```
char *sm_Error (errorCode)
int          errorCode;       /* error code returned by an sm function /*

char *sm_ErrorId (errorCode)
int          errorCode;       /* error code returned by an sm function /*
```

These are the only Storage Manager functions that do not return an integer. When a client library function returns an error, sm_Error( ) can be called by the application to get a string that provides a brief description of the error. It also provides descriptions of Unix error codes. Sm_ErrorId( ) is used to return the string representation of the error code. For example, the call sm_ErrorId(esmBADOID) returns the string "esmBADOID", and the call sm_Error(esmBADOID) returns the string "invalid object id."

If the client is disconnected from a server (by a server crash, network failure, etc.) the client library tries to reconnect to the server the next time it issues a request to the server. If the server in question is not available, the Storage Manager returns an error such as esmSERVERDIED or a Unix error such as ECONNREFUSED. While the server in question is doing recovery after a restart, esmTRANSDISABLED is returned. The server responds to requests when recovery is completed.

4.2. Initialization and Shutdown Operations

Initialization and shutdown functions are used at the beginning and end of an application program, but most of them can be called at any time. The pertinent functions are sm_SetClientOption( ), sm_GetClientOption( ), sm_ParseCommandLine( ), sm_ReadConfigFile( ), sm_Initialize( ), and sm_ShutDown( ).

Before initializing the Storage Manager client with sm_Initialize( ), a number of client configuration options must be set by the application. Options can be set through calls to sm_SetClientOption( ), sm_ParseCommandLine( ), or sm_ReadConfigFile( ). These options are summarized in Table 1. See Section 3 for information that applies to all options.

Option Name	Option Type	Possible Values	Default Values	Option Description
bufpages	int	> 4	none	# pages in the buffer pool
groups	int	> 3	20	# buffer groups
userdesc	int	> 0	2000	# user descriptors
mount	string	volid port@host	none	where to find server for this volume
lognewpages	Boolean	yes,no,true,false	no/false	client logs new pages
deallocpages	Boolean	yes,no,true,false	yes/true	removes empty pages
pagelock	string	SH,EX	SH	default lock for pages
traceflags	int	>= 0	0	set tracing flags
locktimeout	int	>= 0	30	# 10-second intervals willing to await a lock

Table 1: Client Options

The “bufpages” option sets the size of the client buffer pool in 8 Kbyte pages (or *n* byte pages, for *n*=MIN_PAGESIZE; MIN_PAGESIZE is defined in `sm_client.h`). See Section 4.11.3, **Tuning the Application** for more information about setting this option.

The “groups” option sets the limit on the number of buffer groups that can be opened at once. The default value is 20. See Section 4.6, **Buffer Operations**, for more information about buffer groups.

The “userdescs” option sets the limit on the number of open user descriptors. The number of user descriptors should be set to the maximum number of simultaneous object references that are expected by the application program. The default value is 2000. See Section 4.7, **Operations on Objects**, for more information about user descriptors.

The “lognewpages” option, if “yes”, causes the client to generate log pages for newly allocated pages, and if “no”, causes the server to generate the log pages. Setting this option to “no” results in fewer log records shipped to servers and usually lowers log space requirements for transactions that create objects. With rare patterns of use, setting “lognewpages” to “yes” results in better performance: if the objects that cause new pages to be allocated are small, and if enough work is done between object-creation operations to cause the newly allocated pages to be swapped, the preferred value for “lognewpages” is “yes”. In general, it is difficult to predict which objects will be be created on newly allocated pages. The “lognewpages” option may be set only when a transaction is not active.

The “deallocpages” option, if “yes”, causes the client to deallocate pages that become empty after objects are destroyed. If the option’s value is “no”, these pages remain in the file, and do not get used again unless an appropriate *near-hint* is given when an object is subsequently created. Under most circumstances, the preferred value of “deallocpages” is “yes”. If objects are created and destroyed in a LIFO fashion, and if the near-hint for object creation is NEAR_LAST, the preferred value is “no”.

The “pagelock” option changes the default lock mode for pages. See the Section 4.2, **Initialization and Shutdown Operations**, and Appendix A, **Locking Protocol for Storage Manager Operations** for information about using options.

The “traceflags” option is used to turn on tracing, and is only available in a Storage Manager that was compiled with -DDEBUG. The “traceflags” option takes effect immediately and can be set at any time.

The “mount” options indicate the locations of the volumes that the applications use. The “mount” option may be used more than once, to *add* new volumes to the client library’s set of usable volumes, or to *change* the location of a volume. The option value consists of a volume’s integer identifier, an Internet address, and a port at which can be found a server that manages the volume. The Internet addresses and port have format *port @ host*, where both the port and the host can be numeric or symbolic. Symbolic port names must be found in the services database used by *getservbyname(3n)*, and symbolic host names must be in the host name database used by *gethostbyname(3n)*. The following example shows three values for the “mount” option that accomplish the same thing in three ways. The volume 1000 is managed by the server listening on port 1152 (which is called “bounty” in the `/etc/services` database) on the local machine, whose Internet address is 128.105.2.153, also known as “pitcairn.isle.edu” to the host-name server.

```
1000 1152@128.105.2.153
1000 bounty@pitcairn.isle.edu
1000 1152@pitcairn.isle.edu.
```

and

```
1000 bounty@128.105.2.153
```

The host name *localhost* **does not work** if you are using distributed transactions (multiple cooperating servers).

Volume identifiers **must identify volumes unambiguously, across all servers.**

For each application or client, **all the host names used for a given server must resolve to the same Internet address**. Using the above example, this means that “128.105.2.153” and “pitcairn.isle.edu” are interchangeable. “Localhost”, which resolves to the Internet address 127.0.0.1, is not interchangeable with “128.105.2.153” or “pitcairn.isle.edu”, even though it addresses the same machine when used by a client on “pitcairn.isle.edu”.

It is acceptable to use two *different* servers running on a machine, by addressing them at different *ports*. This means that

```
1000 1151@pitcairn.isle.edu
```

and

```
2000 1152@pitcairn.isle.edu
```

can serve an application.

The “locktimeout” option limits the time the server waits to acquire a lock on behalf of the client. The value represents a number of 10-second intervals. A value of zero means that the server does not wait at all, and if the lock cannot be acquired immediately, the client operation returns esmFAILURE, with esmLOCKBUSY in sm_errno. The option value can be changed at any time. The value that is in effect at the time a transaction makes its first request to a server is the value used for lock requests on that server for the duration of the transaction. See Appendix A, Section A.3, **Deadlock Detection and Avoidance**, for more information about locks. See also Section 4.4, **Mounting and Dismounting Volumes**, for information concerning the protocol between clients and servers.

To support code that was written before the configuration option facility was added, the client library looks for the environment variable ESMCONFIG. If set, ESMCONFIG indicates a configuration file to read. The file is read using sm_ReadConfigFile( ), with its “programName” argument having the value NULL. It is read before any option is set, so all other functions that set options override those found in the ESMCONFIG file.

```
sm_SetClientOption (optionName, optionValue, valueType)
char            *optionName;         /* IN name of the option to set */
void            *optionValue;        /* IN new value for the option  */
SMDATATYPE      valueType;           /* IN type of optionValue */
```

Sm_SetClientOption( ) sets the option named “optionName” to the value in “optionValue”. The “valueType” arguments indicates the type addressed by “optionValue”. The supported types are SM_int, SM_Boolean, and SM_string. If “valueType” matches the type of the option as specified in Table 1, a simple assignment is done. If “valueType” is SM_string and the option has a different type, a conversion is performed.

```
sm_GetClientOption (optionName, optionValue)
char            *optionName;         /* IN name of the option to get */
void            *optionValue;        /* OUT value for the option  */
```

Sm_GetClientOption( ) retrieves the value for “optionName” and returns it in “optionValue”. It is assumed that the location addressed by “optionValue” matches the type, found in Table 1, for the option. For string-type options, the argument “optionValue” is treated as type “const char **ρq. That is, it should contain the address of a pointer variable that is updated to point to a read-only buffer containing the option value.

```
sm_ParseCommandLine (argc, argv, errorMsg)
int             *argc;               /* IN/OUT number of command line arguments */
char            **argv;              /* IN/OUT command line arguments */
char            **errorMsg;          /* OUT syntax error message */
```

Sm_ParseCommandLine( ) searches the command line, “argv”, for any client options. Command-line options are prefixed by a “-”. The value for the option must follow the option name. The Storage Manager ignores any command-line argument that is not recognized as a Storage Manager client option. If a client option is found, the name and value are removed from “argv” and “argc” is decremented by 2, even if there is an error in the option such as being given an illegal value. If there is an error processing any option, “errorMsg” is changed to point to an error message string.

```
sm_ReadConfigFile (configFile, programName, errorMsg)
char            *configFile;         /* IN name of the configuration file */
char            *programName;        /* IN name of the application */
char            **errorMsg;          /* OUT syntax error message */
```

Sm_ReadConfigFile( ) reads the option configuration file “configFile”, and sets the options indicated. If “configFile” is NULL, the default configuration files `/usr/lib/exodus/sm_config`, `$HOME/.sm_config`, and `./.sm_config` are read in that order, if they exist. The name of the default configuration file `/usr/lib/exodus/sm_config` can be changed with a minor Storage Manager source code change described in the installation manual, *EXODUS Storage Manager Installation Manual*. The "programName" option gives the program name for matching with options in the configuration file. If "programName" is NULL and a previous call to sm_ReadConfigFile( ) had a non-NULL "programName", the previous "programName" is used. If no previous call was made and a "programName" is not given, configuration file lines that contain a program name are not used; only generic entries, such as `client.bufpages: 1000` and `client*bufpages: 1000` are used.

When an error occurs while reading the file, an error message is stored in "errorMsg" and esmFAILURE is returned, as with other Storage Manager functions. The "errorMsg" is describes syntax related errors in the configuration file.

See Section 3 for information about the format of configuration files.

sm_Initialize ( )

Sm_Initialize( ) initializes the Storage Manager's data structures. No Storage Manager functions except option and configuration file functions may be called before sm_Initialize( ) is called. Options that do not have defaults must be set before sm_Initialize( ) is called, otherwise esmFAILURE is returned, sm_errno is set to indicate what the problem is.

sm_ShutDown ( )

Sm_ShutDown( ) closes all the open buffer groups and frees the memory allocated at run-time by the client library. Once the client library has been shut down, it can used again by calling sm_Initialize( ). The client library loses track the information in the "mount" client options, so if sm_Initialize( ) is to be used again, the configuration files must be reread or the mount options must be reset with sm_SetClientOption( ).

Figure 2 shows a simple "hello world" application for the Storage Manager. It sets configuration options, initializes the client library, and shuts down the client library. A more complete program would, begin transactions, perform operations on objects, files, and indexes. More sample programs are included with the software release.

4.3. Transactions

The Storage Manager supports transactions, including concurrency control and recovery. Transactions may involve data managed by several Exodus Storage Manager servers, in which case a two-phase commit protocol, based on Presumed Abort [Moha83], determines the fate of the transaction when the application commits the transaction. The fact that such a transaction is distributed over several servers is invisible to the application. On the other hand, the Storage Manager (server or servers) can cooperate in a two-phase commit procedure with other transaction processing systems when the external two-phase commit functions are used. The external two-phase commit functions also can be used explicitly to invoke the two phases for a transaction that involves only Exodus Storage manager servers. The external two-phase commit functions are described under "Advanced Topics", in Section 4.11.3, **External Two-Phase Commit Functions**,

```
/*
 *  "Hello world" program: demonstrates initialization and shutdown.
 */
#include <stdlib.h>
#include "sm_client.h"

void ErrorCheck (int, char *);

main(int argc, char** argv) {
    int        e;
    char       *errorMsg;

    e = sm_ReadConfigFile(NULL, argv[0], &errorMsg);
    if (e != esmNOERROR) {
        fprintf(stderr, "Configuration file error: %s", errorMsg);
        ErrorCheck(e, "sm_ReadConfigFile");
        exit(0);
    }
    e = sm_ParseCommandLine(&argc, argv, &errorMsg);
    if (e != esmNOERROR) {
        fprintf(stderr, "Command line error: %s", errorMsg);
        ErrorCheck(e, "sm_ParseCommandLine");
        exit(0);
    }

    e = sm_Initialize( ); ErrorCheck(e, "sm_Initialize");
    printf("Hello world!");
    e = sm_ShutDown( ); ErrorCheck(e, "sm_ShutDown");
}

void ErrorCheck (int e, char *func) {
    if (e < 0) {
        fprintf(stderr, "Storage Manager error \"%s\" in %s",
             sm_Error(sm_errno), func);
        exit(1);
    }
}
```

Figure 2: Example Program

Object, file, index, and root entry operations must be performed within the scope of a transaction, or an error is returned. An application can run no more than one transaction at a time. Transactions cannot be nested, suspended, or resumed.

In order to guarantee the semantics of transactions, operations on objects and files acquire *locks*. Appendix A describes the kinds of locks acquired by the client library functions.

4.3.1. Transaction Identifiers

Each transaction has a local transaction identifier, which is assigned by the Storage Manager. The data type TID represents a transaction identifier. The application can treat a TID as an opaque value. The Storage Manager maintains a global variable, Tid, of type TID, which value the application can inspect, but had better not modify.

The application can use the following two macros to give an initial value to a transaction identifier, and to recognize that value.

INVALIDATE_TID (TID tid)

sets the “tid” argument to an invalid transaction identifier.

TID_IS_INVALID (TID tid)

returns TRUE if “tid” is the value given by INVALIDATE_TID( ), FALSE if not. TID_IS_INVALID( ) does not tell if there is an active transaction with the given transaction identifier.

4.3.2. Transaction States

An application is always in one the following states: not running a transaction (INACTIVE), running a transaction (ACTIVE), running a transaction that has been (partially) aborted (ABORTED).

An application is in the INACTIVE state until it calls sm_BeginTransaction( ), and after a call to sm_CommitTransaction( ) or sm_AbortTransaction( ).

If the Storage Manager server or client library aborts a transaction, which sometimes happens because of an error on the part of the application, the application is in the ABORTED state until a call to sm_AbortTransaction( ). While in the ABORTED state, a call to any function other than sm_AbortTransaction( ) returns the error esmTRANSABORTED.

4.3.3. Transaction Operations

sm_BeginTransaction (tid)
TID *tid; /* OUT transaction ID */

Sm_BeginTransaction( ) is called at the beginning of a transaction. The argument “tid” corresponds to a transaction identifier and is assigned by the Storage Manager.

Sm_BeginTransaction( ) **does not** contact any servers or initiate a transaction with any server, since the operation has no arguments to indicate which servers are of interest. It only begins a transaction “locally”. Once a transaction has begun locally, the client library initiates transactions on servers when data references so require.

sm_CommitTransaction (tid)
TID tid; /* IN transaction ID */

Sm_CommitTransaction( ) is called to commit the effects of a transaction. If the commit succeeds, all changes made to data since the beginning of the transaction are guaranteed to be persistent, even in the event of system failure. See Section 4.9.1, **Consistency Guarantees for Files**, for more information about this guarantee. If the commit fails, an error is returned, and the transaction is aborted. When a transaction is committed, all user descriptors (see sm_ReadObject( )) are released. Buffer groups attached to the transaction (see sm_OpenBufferGroup( )) are closed.

sm_AbortTransaction (tid)
TID tid; /* IN transaction ID */

Sm_AbortTransaction( ) aborts a transaction. Sm_AbortTransaction( ) releases all the user descriptors that were created during the transaction (see sm_ReadObject( )). Buffer groups attached to the transaction (see sm_OpenBufferGroup( )) are closed.

The persistent data appear as if the transaction never began. The execution state of the application program is not affected by calling sm_AbortTransaction( ). The result is that the transient data in the program's address space do not match the state of the persistent data. The problem can be alleviated to some degree by judicious use of *setjmp(2)*, *longjmp(2)*, and lexical scoping in the application program. The following macros, which are defined in `sm_client.h`, do that:

SM_BEGIN_TRANSACTION (tid, abortCode)

```
TID         *tid;            /* transaction ID */
int         abortCode;       /* location to store abort code */
```

SM_BEGIN_TRANSACTION begins a transaction block (i.e. it opens a new lexical scope in C or C++). The transaction ID is placed in "tid". The argument "abortCode" **must** be a variable. This variable can be checked at the end of the transaction to determined if it was aborted.

SM_COMMIT_TRANSACTION (tid)

```
TID         tid;             /* transaction ID */
```

SM_COMMIT_TRANSACTION ends a transaction block. When this statement is executed, the transaction is committed, assuming no error occurs during commit. Immediately after the SM_COMMIT_TRANSACTION statement, the "abortCode" variable given in the SM_BEGIN_TRANSACTION statement should be checked to see if any error occurred. If no error occurred, "abortCode" is set to esmNOERROR. Otherwise, "abortCode" is set to the value given in SM_ABORT_TRANSACTION.

SM_ABORT_TRANSACTION (abortCode)

```
int         abortCode;       /* error to return on abort */
```

SM_ABORT_TRANSACTION aborts the active transaction (i.e. sm_AbortTransaction( ) is called) and resumes execution at the line immediately following the SM_COMMIT_TRANSACTION statement for the transaction. The SM_ABORT_TRANSACTION macro does not need to be called within the lexical scope of the transaction block. It can be called in any function operating in the dynamic scope of the transaction. The "abortCode" argument sets the "abortCode" variable given in SM_BEGIN_TRANSACTION.

When a SM_ABORT_TRANSACTION is called, the program's control is transferred to the program point after the SM_COMMIT_TRANSACTION statement. The stack pointer is restored to the level of the transaction block, so functions on the program's stack after it are not completed. **For C++, this means that destructors are not called for any local variables in those functions.**

Examples of using both the transaction macros and functions can be found in the producer-consumer example given in the Storage Manager software release.

4.4. Mounting and Dismounting Volumes

An application program **does not** need to mount and dismount volumes explicitly. In most cases, the client library automatically mounts a volume when the application makes its first reference to that volume. An application that does not explicitly mount a volume may, when it performs its first operation on an object, find that the server for that object is not running. Writing programs to handle such common errors can be difficult, so it may be more convenient to mount volumes before proceeding with operations on data. Sm_MountVolume( ) serves that purpose. If that server has not yet been contacted, sm_MountVolume( ) establishes a connection to the server and mounts the volume. It does not begin a transaction. (See Section 4.3.3, **Transaction Operations** to understand how transactions are begun.)

When an application exits or calls sm_ShutDown( ), connections to servers are severed, and the servers dismount the volumes used by the application. A server severs its connections and dismounts the volumes if an application is *inactive* for a significant time. An application is inactive if it has no transaction running.

An application can dismount volumes explicitly, causing the volumes to be dismounted at the server. An application that continues to run after it is finished using the Storage Manager would do well to use sm_ShutDown( ). If it is inappropriate to use sm_ShutDown( ), but such an application is finished with a set of volumes, it would do best to dismount the volumes, particularly if the volumes are likely to be reformatted.

```
sm_MountVolume ( volid )
VOLID       volid;              /* IN volume to mount */
```

Sm_MountVolume( ) causes the volume identified by “volid” to be mounted. A side effect of the operation is that the client library has established a connection with the server that manages this volume.

If the volume cannot be mounted, sm_MountVolume( ) returns esmFAILURE and a value in sm_errno that describes the reason: esmNOSUCHVOLUME (the client library cannot identify the server for this volume because there is no “mount” option for this volid), esmTRANSABORTED (the transaction was aborted during the previous operation, and the next thing the application must do is abort the transaction), esmSERVERDIED (connection with server was severed during the mount operation), or any Unix error message from `<errno.h>` (such as ENETDOWN and ECONNREFUSED), which indicate that the server is not running or is unreachable through the network.

```
sm_DismountVolume ( volid )
VOLID       volid;              /* IN volume to dismount */
```

The “volid” argument identifies the volume to be dismounted. If the volume is not mounted, the operation returns esmFAILURE, and the client library returns esmBADVOLID in sm_errno.

4.5. Root Entries

The root entry facility is designed for applications to get a handle to data on a volume. [1] A common use of a root entry is to associate a string name with an object identifier for an object containing information about the contents of the volume. For example, in a database system, this might be the object identifier for the catalog.

[1] Root entries cannot be created on temporary volumes.

A root entry is a string and data pair stored in a special location on a volume, called the root area. The string, called the name, is used to identify the entry. The name string must be null-terminated. The maximum lengths of the name (including the terminating null) and data are defined by MAX_ROOTNAME_SIZE and MAX_ROOTDATA_SIZE respectively. An error is returned if the available number of root entries is exceeded. Names and data are limited to 32 bytes each, and approximately 90 root entries can reside in a volume’s root area.

sm_SetRootEntry (volid, name, data, dataLength)

```
VOLID      volid;          /* IN volume identifier */
char       *name;          /* IN name to store data entry under */
void       *data;          /* IN data entry to be stored */
int        dataLength;     /* IN length of the data */
```

Sm_SetRootEntry( ) is creates or updates an entry. The “name” argument is the name of the entry and the “data” argument is the data to be stored. The number of bytes in the data is given in “dataLength”. For example, to store the contents of the variable “rootOid” under the name “root-obj”, use `sm_SetRootEntry(volid, “root-obj”, (char*) &rootOid, sizeof(rootOid))`.

Sm_SetRootEntry( ) obtains an exclusive lock on the root area of the volume, so updates to root entries should be performed in a short transaction.

sm_GetRootEntry (volid, name, data, dataLength)

```
VOLID      volid;          /* IN volume identifier */
char       *name;          /* IN name of the entry */
void       *data;          /* OUT data stored under name */
int        *dataLength;    /* IN/OUT length of the data */
```

Sm_GetRootEntry( ) retrieves the root entry named “name”. The data is placed in “data” and the length of the data is returned in “dataLength”. If “dataLength” is initialized with a value greater than or equal to zero, the maximum number of bytes copied to “data” is “dataLength”. If “dataLength” is initialized with a value less than zero, the entire length of the data is copied to “data”.

Sm_GetRootEntry( ) obtains a share lock on the root area of the volume. This share lock blocks other transactions from updating or removing root entries until the transaction is committed or aborted. If no root entry exists for “name”, esmFAILURE is returned and sm_errno is set to esmBADROOTNAME.

sm_RemoveRootEntry (volid, name)

```
VOLID      volid;          /* IN volume identifier */
char       *name;          /* IN name of entry */
```

Sm_RemoveRootEntry( ) removes the root entry stored under “name”. Sm_RemoveRootEntry( ) obtains an exclusive lock on the root area of the volume, so removal of root entries should be performed in a short transaction.

4.6. Buffer Operations

The Storage Manager buffer manager implements the concept of a *buffer group*, as proposed in the DBMIN buffer management algorithm [Chou85]. The essence of the DBMIN algorithm is that competing uses of the buffer pool may be allocated their own buffers, to minimize competition for the buffers and to eliminate thrashing in the buffer pool.

All uses of the buffer pool are made through a buffer group. A buffer group is a container of page buffers, with a limit on the number of *fixed* pages it can contain. Fixed pages are guaranteed to remain in the buffer pool until they are *unfixed*. Their locations (virtual addresses) may change, but the pages remain in the virtual address space of the buffer pool. Each buffer group has a replacement policy, which controls the replacement of unfixed pages within the buffer group.

Buffer groups can be opened and closed at any time, whether or not a transaction is running. If a buffer group is opened in a transaction, it may be "attached" to the transaction, which means that the buffer group is closed by the client library when the transaction ends. An attached buffer group can be closed explicitly by the application before the transaction ends.

The following two macros can be used with buffer groups to give an initial value to a buffer group index and to recognize that value.

INVALIDATE_BUFGROUP (int bufgroup)

sets the "bufgroup" argument to an invalid buffer group index.

BUFGROUP_IS_INVALID (int bufgroup)

returns TRUE if "bufgroup" is the value given by INVALIDATE_BUFGROUP( ), FALSE if it is not. BUFGROUP_IS_INVALID( ) does not tell if there exists a buffer group with the given index.

```
sm_OpenBufferGroup (groupSize, policy, groupIndex, flags)
int         groupSize;        /* IN the maximum group size in pages */
int         policy;           /* IN the group's replacement policy */
int         *groupIndex;      /* OUT the group's index */
FLAGS        flags;           /* IN buffer group attributes */
```

Sm_OpenBufferGroup( ) opens a new buffer group. The "groupSize" argument specifies the size of the buffer group in MIN_PAGESIZE pages. The sum of the sizes of all open buffer groups cannot exceed the size of the buffer pool. (See Section 4.11.3, **Tuning the Application**.) The choice for "policy" is least-recently-used (BF_LRU) or most-recently-used (BF_MRU). BF_LRU and BF_MRU are defined in `sm_client.h`. The argument "groupIndex" is filled by the Storage Manager and must be used in subsequent references to the buffer group. (All operations on files and objects require a buffer group index.)

The "flags" indicates whether the buffer group is to be associated with a transaction. NOFLAGS indicates that it is not. TRANS_GROUP indicates that the buffer group is associated with the current transaction. The group is closed by the client library when the active transaction ends. If TRANS_GROUP is used, a transaction must be running at the time sm_OpenBufferGroup( ) is called.

The effect of sm_OpenBufferGroup( ) is to reserve "groupSize" pages in the client's buffer pool. No buffer group is opened on the server.

```
sm_BufferGroupInfo (groupIndex, maxPages, fixedPages, unfixedPages)
int             groupIndex;         /* IN the group to inspect */
int             *maxPages;          /* OUT max fixed pages allowed     */
int             *fixedPages;        /* OUT current # of pages fixed    */
int             *unfixedPages;      /* OUT current # of pages unfixed  */
```

Sm_BufferGroupInfo( ) returns information about the open buffer group identified by "groupIndex". The function returns the buffer group's size limit in pages in "maxPages". In "fixedPages", it returns the number of pages currently fixed in the buffer group. See the next section for more information about these functions. The argument "unfixedPages" refers to all buffer pages that belong to the buffer group, but are not fixed, that is these pages may be removed from the buffer pool if space is needed for fixed pages.

```
sm_CloseBufferGroup (groupIndex)
int             groupIndex;         /* IN the group being closed */
```

Sm_CloseBufferGroup( ) closes the open buffer group identified by "groupIndex".

4.7. Operations on Objects

An object in the Storage Manager is a container of bytes. It can be empty. It can have as many as 2^{31} bytes, if the volume on which it resides is large enough. An object must fit on a single volume (storage device or partition). When an object is created, the Storage Manager gives the object a unique object identifier. An object identifier is described by a structure of the type OID, defined as follows:.

```
typedef struct {
    SHORTPID      pid;          /* 32-bit page address of the object's header */
    SLOTINDEX     slot;         /* 16-bit slot number of the object on the page */
    VOLID         volid;        /* 16-bit identifier of the volume */
    UNIQUE        unique;       /* 32-bit number generated at creation time */
} OID;
```

The first three fields of an OID are the physical address of the object; they identify a volume, a page within the volume, and a *slot* on the page. An object's identifier never changes. The client library sometimes moves objects, such as when an object grows beyond the size of a page, at which time the object is marked as *forwarded*, but its OID remains unchanged.

The "unique" field of an OID is special 32-bit value that is generated when the object is created and used to detect dangling and corrupted OIDs. The generation of unique numbers is discussed in Appendix B.

Every time an object is accessed by its OID, the Storage Manager validates the OID. The application can use the following macros to give an illegitimate initial value to an OID, and to recognize that value:

```
INVALIDATE_OID (OID oid)
```

sets the "oid" argument to an invalid object identifier.

OID_IS_INVALID (OID oid)

returns TRUE if "oid" is the value given by INVALIDATE_OID( ), FALSE if it is not.

Each object has an *object header*, which describes the object, and which can be retrieved without retrieving the object's data. The structure of an object header is shown below:

```
typedef struct {
    TWO     properties;    /* a bit vector */
    TWO     tag;           /* supplied by the application */
    int     size;          /* size of the object in bytes */
} OBJHDR;
```

The "tag" is a two-byte field that the Storage Manager does not interpret. It is for use by the application. No restriction is put on the contents of "tag" fields. As its name implies, the "size" field is the size of the object in bytes. The "properties" field is a read-only bit-vector that indicates the presence or absence of the following properties of objects:

P_LARGEOBJ	set if the object is a large object.
P_MOVED	set if this object has been forwarded to another page.
P_FROZEN	set if the object is a frozen version.
P_VERSIONED	set if the object is a frozen version or a descendent of a frozen version.

Each object resides in a *file* on a *volume*. When an object is created, the application tells the client library in which file to place the object. Files and their uses are discussed in the next section; details of their use are not pertinent to understanding the operations on objects.

Before an operation can be performed on an existing object, the object, or at least the affected parts of the object, must be brought into the application's address space. This is called *pinning* the object or its parts. When the object is no longer needed, it must be *unpinned*, to make room for other objects to be pinned[2]. When the client library pins and object in order to perform an operation on behalf of the application (for example, appending bytes to an object), the client library pins the necessary parts of the object and unpins them before it returns control to the application. When the application pins part of an object for its own purposes (such as writing over bytes in the object), the pinned part is placed in the client's buffer pool, and the client library creates a "handle" for the the object. The handle is called a *user descriptor*. The application can refer to an object only through user descriptors. The application must unpin the object by *releasing* the user descriptor when it is done using the object.

A user descriptor is called *valid* if and only if the byte range it addresses is pinned. An application can pin an object or overlapping parts of an object any number of times, having any number of valid user descriptors for the same data in an object. (This is not wise for performance reasons, but it can be done.)

The client library functions that pin ranges of bytes return user descriptors to describe the bytes pinned. Functions that require that the range of bytes they affect be pinned take user descriptors

[2] Objects are pinned; pages are fixed. The gist of the two verbs is the same.

as input arguments. The client library functions that do not take user descriptor arguments do not ultimately change the quantity of bytes pinned or the number of pages fixed in the buffer pool. **Such functions may change the ranges of bytes addressed or the bytes themselves, but they do not change the quantity of bytes addressed.** (For example, the function sm_InsertInObject( ) may affect valid user descriptors even though it does not take and user descriptors as arguments.)

User descriptors have the following form:

```
typedef struct {
    char      *basePtr;      /* ptr to start of data */
    int       byteCount      /* number of bytes accessible */
    int       objectSize;    /* total size of object */
    TWO       userFlags;     /* properties field from object header */
    TWO       type;          /* for use only  by E */
    TWO       flags;         /* for use only  by E */
    TWO       tag;           /* tag field from the object header */
    OID       oid;           /* oid of object being referenced */
} USERDESC;
```

The “basePtr” field of a user descriptor points to the start of the object’s data in the buffer pool, while the “byteCount” field indicates the number of bytes accessible to the application program through this user descriptor. The value “objectSize” is the length of the entire object. The “user-Flags” field holds a copy of the properties field from the object’s header. The “type” and “flags” fields are used by the E language’s persistent virtual machine. Finally, the “tag” field contains a copy of the “tag” field in the object’s header.

An object’s data is referenced indirectly via the “basePtr” field. **References by the application must always be indirect via “basePtr”**. The indirection is necessary because there are times when the Storage Manager moves an object in the buffer pool, and the “basePtr” of each user descriptor that references the object is updated to account for the move.

The remainder of this section describes the Storage Manager functions for operating on objects. It is divided into sub-sections that describe creating and destroying objects, pinning and unpinning parts of objects, modifying objects, and using object headers.

4.7.1. Creating and Destroying Objects

```
sm_CreateObject (groupIndex, fid, nearHint, nearObj, objHdr, length, data, oid)
int         groupIndex;    /* IN buffer group to use */
FID         *fid;          /* IN file in which object is to be placed */
int         nearHint;      /* IN flag indicating where to create the new object */
OID         *nearObj;      /* IN create the new object near this object */
OBJHDR      *objHdr;       /* IN the object’s header */
int         length;        /* IN amount of data */
void        *data;         /* IN the initial data for the object */
OID         *oid;          /* OUT the new object’s OID */
```

Sm_CreateObject( ) creates an object in the file identified by “fid”. If “objHdr” is not NULL, the “tag” field in the header of the new object is initialized with the contents of the “tag” field in the header structure addressed by “objHdr”. When “data” is not NULL, the object is initialized with the data addressed by the argument “data”; in this case, “length” specifies how much data to copy. When “data” is NULL, an object of size “length” is created and filled with zeroes.

The argument “nearHint” specifies where the new object should be created. The following values, defined in `sm_client.h`, are near hints: NEAR_OBJ, NEAR_FIRST, and NEAR_LAST. If “nearHint” is set to NEAR_OBJ, the new object is created near the object designated by “nearObj”. If “nearHint” is set to NEAR_FIRST or NEAR_LAST, “nearObj” is ignored and the new object is created near the first or last object in the file, respectively.

If sm_CreateObject( ) is successful, the OID structure pointed to by “oid” is filled with the OID of the new object. Sm_CreateObject( ) does not leave the new object pinned.

```
sm_DestroyObject (groupIndex, oid)
int          groupIndex;      /* IN buffer group in use */
OID          *oid;            /* IN the object to destroy */
```

Sm_DestroyObject( ) destroys an object. If any user descriptors are valid for the object when the object is destroyed, they are made invalid, and they must be released with sm_ReleaseObject( ), described below.

4.7.2. Pinning and Unpinning Objects

The following two functions change the number of pages fixed in the client buffer pool. All the other functions that operate on objects fix pages temporarily and unfix the pages before returning.

```
sm_ReadObject (groupIndex, oid, start, length, userDesc)
int          groupIndex;      /* IN buffer group in use */
OID          *oid;            /* IN object to read */
int          start;           /* IN starting offset of read */
int          length;          /* IN amount of data to read */
USERDESC     **userDesc;      /* OUT descriptor to access the data */
```

Sm_ReadObject( ) reads part or all of the object identified by “oid” into the buffer group identified by “groupIndex”. If “length” is READ_ALL, the entire object is read (assuming that the size of the entire object is not greater than the amount of unpinned space in the buffer group). Otherwise, the bytes to be read are specified by “start” and “length”.

Sm_ReadObject( ) pins the specified range of bytes in the buffer pool and returns a user descriptor to the caller. **Bytes pinned in the buffer pool by sm_ReadObject( ) remain pinned until they are explicitly released by sm_ReleaseObject( ).**

While sm_ReadObject( ) can be used to get information about the object (from the object header) by giving it a length of zero, sm_ReadObjectHeader( ) is the preferred way to meet the same objective. Sm_ReadObject( ) performs work that is unnecessary when only the object header is of interest, and it always fixes at least one page in the buffer pool, even if the given length is zero.

The user descriptor consumes resources that must be freed with sm_ReleaseObject( ), even if the object is not pinned (zero is given for “length”).

```
sm_ReleaseObject (userDesc)
USERDESC   *userDesc;              /* IN descriptor returned by ReadObject */
```

Sm_ReleaseObject( ) unpins a range of bytes of an object that was pinned by sm_ReadObject( ), and frees the resources associated with the user descriptor. If the user descriptor is not valid, sm_ReleaseObject( ) sets sm_errno to esmBADUSERDESC and returns esmFAILURE.

4.7.3. Modifying Objects

Four functions modify objects: sm_WriteObject( ), sm_InsertInObject( ), sm_AppendToObject( ), and sm_DeleteFromObject( ). Sm_WriteObject( ) cannot be used to change the size of an object, only to overwrite parts of an object. The other three functions can change the size of an object. These functions provide substantial flexibility, and their efficiency varies. Changing the size of a small object (one that fits on a disk page) is relatively inexpensive. It is less expensive than reading and writing the object. For large objects, performing many small-size changes can be expensive in CPU time and buffer space utilization. If a large object is pinned several times simultaneously, through different user descriptors, updates to the object are very expensive. If a large number of small-size changes is required, we recommend accumulating the changes and performing them in larger chunks.

```
sm_WriteObject (groupIndex, start, length, data, userDesc, release)
int          groupIndex;         /* IN buffer group in use */
int          start;              /* IN starting offset of write */
int          length;             /* IN amount of data to be written */
void         *data;              /* IN pointer to the data */
USERDESC     *userDesc;          /* IN descriptor returned by ReadObject */
BOOL         release;            /* IN whether to release the object */
```

Sm_WriteObject( ) overwrites the region of bytes from (userDesc->baseptr + start) to (userDesc->baseptr + start + length - 1) with the data addressed by the “data” argument. The given byte range must have been pinned (which means that the user descriptor must be valid). If “release” is TRUE, the range of bytes given by “userDesc” is unpinned when sm_WriteObject( ) returns. If “data” is NULL, the region is filled with zeroes. **All updates to objects must be performed using sm_WriteObject( )** so that the updates can be logged, and the transaction semantics can be guaranteed.

```
sm_InsertInObject (groupIndex, oid, start, length, data)
int          groupIndex;         /* IN buffer group in use */
OID          *oid;               /* IN object we’re inserting into */
int          start;              /* IN starting offset of insert */
int          length;             /* IN amount of data being inserted */
void         *data;              /* IN data to insert */
```

Sm_InsertInObject( ) inserts “length” bytes of data into an object, beginning at the offset “start”. If “data” is NULL, the inserted region is filled with zeroes. If there are any valid user descriptors (those for which sm_ReleaseObject( ) has not been called) for the object at the time the insertion takes place, they are reestablished if necessary. After the insertion, the base pointers of the valid user descriptors point to the byte within the object indicated by the “start” argument to the sm_ReadObject( ) operation that created the user descriptor. For example, an object’s first five bytes, "ABCDE" are pinned by sm_ReadObject( ), which was called with a “start” offset of zero and a “length” of five. Sm_ReadObject( ) returns a user descriptor, U, which addresses "ABCDE". Sm_InsertInObject( ) inserts "ZZ" at “start” offset zero. The user descriptor U now addresses "ZZABC", which are pinned, while the bytes "DE" are no longer pinned.

```
sm_AppendToObject (groupIndex, oid, length, data)
int          groupIndex;        /* IN buffer group in use */
OID          *oid;              /* IN object we are appending data to */
int          length;            /* IN amount of data being appended */
void         *data;             /* IN data to append */
```

Sm_AppendToObject( ) appends “length” bytes of data to the end of an object. Outstanding user descriptors are handled the same way as sm_InsertInObject( ). If “data” is NULL, the appended region is filled with zeroes.

```
sm_DeleteFromObject (groupIndex, oid, start, length)
int          groupIndex;        /* IN buffer group in use */
OID          *oid;              /* IN object we’re inserting into */
int          start;             /* IN starting offset of delete */
int          length;            /* IN amount of data being deleted */
```

Sm_DeleteFromObject( ) deletes “length” bytes of data from an object, beginning with the byte indicated by the offset “start”. Sm_DeleteFromObject( ) is analogous to sm_InsertObject( ). All valid user descriptors affected by the deletion are, if possible, reset to point to the new absolute offset within the object. There are two cases when this is not possible.

(1) The object’s size is now smaller than the starting offset of a user descriptor. The “basePtr” field in the user descriptor is set to NULL and the user descriptor is made invalid. The user descriptor must be released by sm_ReleaseObject( ) so that its resources can be reclaimed.

(2) The object’s size is now smaller than the original byte range addressable by a user descriptor. The size of the range addressable by the descriptor is reduced to reflect the new size of the object.

4.7.4. Object Headers

```
sm_ReadObjectHeader (groupIndex, oid, objHdr)
int          groupIndex;        /* IN buffer group in use */
OID          *oid;              /* IN read this object’s header */
OBJHDR       *objHdr;           /* OUT place to put the header */
```

Sm_ReadObjectHeader( ) reads an object’s header into the structure addressed by “objHdr”. This function is the preferred one to use to determine if an object’s identifier is valid. If the object’s identifier is invalid, Sm_ReadObjectHeader( ) returns esmFAILURE and puts esmBA-DOID in sm_errno.

```
sm_SetObjectHeader (groupIndex, oid, objHdr)
int           groupIndex;      /* IN buffer group in use */
OID           *oid;            /* IN set this object's header flags */
OBJHDR        *objHdr;         /* IN the new header */
```

Sm_SetObjectHeader( ) modifies an object’s header. Only the “tags” field is modified; the other fields are read-only.

4.8. Versions of Objects

In order to allow efficient updating of shared data, the Storage Manager offers versions of objects. Versions come in two kinds: *working versions* and *frozen versions*. A working version of an object is one that can be modified. Every object has at least one version, which is the object itself. A working version may be frozen, after which it can no longer be modified.

A new working version, called a *descendent*, can be made of a frozen object. The descendent looks like a new object that is a copy of the frozen object from which it came. The Storage Manager determines when it is necessary and efficient to make a copy of the frozen object, and makes the copy at that time.

```
sm_CreateVersion (groupIndex, nearHint, parentObj, nearObj, oid)
int           groupIndex;      /* IN buffer group to use */
int           nearHint;        /* IN flag indicating where to create the new version near */
OID           *parentObj;      /* IN object to create a version of */
OID           *nearObj;        /* IN create the new version near this object */
OID           *oid;            /* OUT the new version's OID */
```

Sm_CreateVersion( ) creates a new version of the object “parentObj” in the file containing “parentObj”. The arguments “groupIndex”, “nearHint”, and “nearObj” are used as in sm_CreateObject( ). The object identifier of the new version is returned in “oid”. **The object identified by “parentObj” must be a frozen version**. The new version is a working version. The new version can be destroyed using sm_DestroyObject( ). When a new version is created, the P_VERSIONED property is set in the object header.

Like sm_CreateObject( ), sm_CreateVersion( ) does not leave anything pinned in the buffer pool.

```
sm_FreezeVersion (groupIndex, oid)
int           groupIndex;      /* IN buffer group to use */
OID           *oid;            /* IN object to be frozen */
```

Sm_FreezeVersion( ) marks an object as frozen, preventing subsequent modification of the object, and allowing new working versions to be made from this object. When an object is frozen, both the P_VERSIONED and the P_FROZEN properties are set in the object header. Once frozen, an object cannot be unfrozen. A frozen object can be destroyed.

4.9. Operations on Files

A Storage Manager file is a flexible container in which objects are place when they are created. No object exists outside a file.

The objects in a file can be *scanned*, meaning that they are visited exactly once.

Files do not have preallocated space or ownership properties. Various consistency guarantees can be associated with files, with the effect that updating data in different files has different costs.

The Storage Manager offers operations for creating, destroying, scanning, bulk-loading files, and for changing the consistency guarantees associated with files. Some operations on files acquire locks on entire files. The locks acquired are described in Appendix A.

A file is identified by a unique file identifier or FID. The Storage Manager does not provide a way to find all files or file identifiers that exist, so it is left to the application to keep track of its file identifiers. For example, consider an application that embeds file identifiers in objects to create a logical hierarchy of files. The application had best destroy the files in a depth-first fashion, lest it lose a file identifier before the file it identifies is destroyed.

The following two macros can be used to give a file identifier an illegitimate initial value, and later to recognize that value:

INVALIDATE_FID (FID fid)

sets “fid” to an invalid file identifier.

FID_IS_INVALID (FID fid)

returns TRUE if “fid” is the invalid identifier given by INVALIDATE_FID( ), FALSE otherwise.

The rest of this section describes operations on files and operations that concern entire files of objects.

4.9.1. Consistency Guarantees for Files

The *log level* of a file determines what level of consistency is maintained for the file in the event that a transaction aborts or a server crashes. There are two log levels for files on data volumes: LOG_ALL and LOG_SPACE. LOG_ALL indicates that consistency is maintained for user data and meta-data. LOG_SPACE indicates that meta-data are guaranteed to be consistent. This means that all objects are available and that they are the correct size, but their contents may be corrupted. Files that have their log level set to LOG_SPACE are flushed when the transaction is committed. **Data pages for large objects (objects that do not fit on a single disk page) may not be flushed, so there is no guarantee that the data is safely on disk until the server dismounts the volume.** The log level is not a permanent attribute of a file. When an application sets the log level for a file, the setting lasts until it is changed or until sm_ShutDown( ) is called. If, in a transaction, the log level for a file is changed from LOG_SPACE to LOG_ALL, the Storage Manager guarantees only that the meta-data are consistent.

LOG_ALL is the default log level for data files. LOG_SPACE is designed to conserve log space and increase performance for those files whose data integrity is not critical. For example, results of a query may be stored in a file with its log level set to LOG_SPACE, since file can be regenerated, in the event of a failure. To conserve log space when loading a large file, the log level for a file may be set to LOG_SPACE. Once the loading transaction is committed, the log level should be set to LOG_ALL.

Files on temporary volumes can have only one log level: LOG_NONE. See Section 5.1.3, **Temporary Volumes**, for more information about temporary volumes.

Sm_SetLogLevel( ) is used to change the log level for a list of files:

```
sm_SetLogLevel (logLevel, fileCount, fids)
int         logLevel;          /* IN log level */
int         fileCount;         /* IN number of files to set level for */
FID         fid[];             /* IN list of files to set level for */
```

The “logLevel” argument takes the values LOG_SPACE and LOG_ALL. The “fileCount” argument indicates the size of the last argument, “fid[]”, which is a list of file identifiers of the files whose log levels are to be affected by this function. It is not an error for a file in the list already to have the given log level.

If “fileCount” is zero, **all** files are given “logLevel”.

The volumes on which the files reside must be available for mounting, and a side effect of setting the log level is that the volumes are mounted.

Sm_SetLogLevel( ) has no effect on files that reside on temporary volumes (see Section 5.1.3, **Temporary Volumes**).

```
sm_CreateFile (groupIndex, volid, fid)
int          groupIndex;       /* IN buffer group in use */
VOLID        volid;            /* IN the volume in which to place the file */
FID         *fid;              /* OUT the file ID of the new file */
```

Sm_CreateFile( ) creates a new file on the volume indicated by “volid”. The file identifier of the new file is returned in the structure to which “fid” points. The caller is responsible for allocating space for the FID.

```
sm_DestroyFile (groupIndex, fid)
int          groupIndex;       /* IN buffer group in use */
FID         *fid;              /* IN the file to destroy */
```

Sm_DestroyFile( ) destroys the file identified by “fid”. The objects in the file are destroyed along with the file. Disk space is released when the transaction is committed.

```
sm_GetFirstOid (groupIndex, fid, oid, objHdr, emptyFlag)
int          groupIndex;       /* IN buffer group in use */
FID         *fid;              /* IN the file */
OID         *oid;              /* OUT first OID */
OBJHDR      *objHdr;           /* OUT the object’s header */
BOOL        *emptyFlag;        /* OUT empty file flag */
```

Sm_GetFirstOid( ) retrieves the object identifier and the object header of the first object in the file designated by “fid”. The first object is the first object on the first physical page in the file. If the file does not contain any objects, “emptyFlag” is set to TRUE.

```
sm_GetLastOid (groupIndex, fid, oid, objHdr, emptyFlag)
int          groupIndex;           /* IN buffer group in use */
FID          *fid;                 /* IN the file */
OID          *oid;                 /* OUT last OID */
OBJHDR       *objHdr;              /* OUT the object's header */
BOOL         *emptyFlag;           /* OUT empty file flag */
```

Sm_GetLastOid( ) retrieves the object identifier and the object header of the last object in the file designated by "fid". The last object is the last object on the last physical page in the file. If the file does not contain any objects, "emptyFlag" is set to TRUE.

```
sm_GetNextOid (groupIndex, baseOid, nextOid, objHdr, endMarker)
int          groupIndex;           /* IN buffer group in use */
OID          *baseOid;             /* IN next relative to this object */
OID          *nextOid;             /* OUT OID of the next object */
OBJHDR       *objHdr;              /* OUT the object's header */
BOOL         *endMarker;           /* OUT end-of-file flag */
```

Sm_GetNextOid( ) retrieves the object identifier and the object header of the next object in the file relative to the object addressed by "baseOid". "EndMarker" is set to TRUE when end-of-file is reached (i.e., when there is no next object for sm_GetNextOid( ) to return).

The next object is that which resides *physically* next in the file. There is no way to scan a file's objects in the order in which they were inserted in the file.

The preferred method for retrieving all the objects in a file is to use scans, described in the next sub-section. Scans are more efficient than using sm_GetNextOid( ), which is present for backward compatibility.

```
sm_GetPreviousOid (groupIndex, baseOid, prevOid, objHdr, endMarker)
int          groupIndex;           /* IN buffer group in use */
OID          *baseOid;             /* IN previous relative to this object */
OID          *prevOid;             /* OID of the previous object */
OBJHDR       *objHdr;              /* OUT the object's header */
BOOL         *endMarker;           /* OUT start-of-file flag */
```

Sm_GetPreviousOid( ) retrieves the object identifier and object header of the previous object in the file relative to the object addressed by "baseOid". "EndMarker" is set to TRUE when start-of-file is reached (i.e., when there is no next object for sm_GetPreviousOid( ) to return). Much like sm_GetNextOid( ), the previous object is the object that is *physically* previous in the file.

4.9.2. Scanning Files

The objects in a file can be visited most efficiently by scanning the file. During a *scan*, the client library locks the entire file so that while one application is using the file, objects cannot be inserted, deleted, or changed by another application. **The Storage Manager does not support a single application's modifying a file during a scan.**

The client library also some information about the state of the scan and the structure of the file being scanned. The information is stored in a *scan descriptor*, a structure of type **SCANDESC**, which is meant to be treated as *opaque* by the application.

sm_OpenScanWithGroup (fid, type, groupIndex, scanDesc, oid)

```
FID         *fid;               /* IN file to scan */
int         type;               /* IN type of scan -- UNUSED */
int         groupIndex;         /* IN buffer group for use in scan */
SCANDESC    **scanDesc;         /* OUT returned scan descriptor */
OID         *oid;               /* IN optional oid to begin scan -- UNUSED */
```

Sm_OpenScanWithGroup( ) initializes a scan on the file indicated by “fid”. A scan descriptor is passed back in “scanDesc”, for use in subsequent scan calls. Using the scan mechanism can be considerably more efficient that using the sm_GetNextOid( ) call or sm_ReadObject( ). Scans use a buffer group, “groupIndex”. This group should be created with the most-recently-used replacement policy, and its size should be tuned to reflect the buffering requirements for the scan. The buffer group should have a size of at least five pages.

Objects are scanned in the order in which they physically reside on disk. After sm_OpenScanWithGroup( ) returns, the scan pointer is before the first object in the file. This is true even if the file is empty, in which case the first call to sm_ScanNextObject( ) returns a flag indicating the end-of-file condition. The “type” and “oid” arguments are not used and are present for backward compatibility.

sm_OpenScan (fid, type, groupSize, scanDesc, oid)

```
FID         *fid;               /* IN file to scan */
int         type;               /* IN type of scan -- UNUSED */
int         groupSize;          /* IN size of buffer group in pages */
SCANDESC    **scanDesc;         /* OUT returned scan descriptor */
OID         *oid;               /* IN optional oid to begin scan -- UNUSED */
```

Sm_OpenScan( ) is like sm_OpenScanWithGroup( ), but it is less flexible, and it is provided for backward compatibility. It is identical to sm_OpenScanWithGroup( ) except that it creates a buffer group with the most-recently-used replacement policy and size “groupSize”. “GroupSize” should be at least five (pages). The buffer group is destroyed when the scan is closed.

sm_ScanNextObject (scanDesc, start, length, retDesc, eof)

```
SCANDESC    *scanDesc;          /* IN scan descriptor */
int         start;              /* IN starting offset in object */
int         length;             /* IN number of bytes to read */
USERDESC    **retDesc;          /* OUT descriptor to access the data */
BOOL        *eof;               /* OUT end of file indicator */
```

sm_ScanNextObject( ) reads the next object in the file and pins the object as if sm_ReadObject( ) were used. “ScanDesc” is the scan descriptor returned when the scan was opened. “Start” is the starting offset within the object to return.

“Length” is the length of the object read to perform. If “length” is READ_ALL, the entire object is read (assuming that the size of the entire object is not greater than the amount of unpinned space in the buffer group). To obtain the object header and OID information for the object, use a “length” of zero.

sm_ScanNextObject( ) returns a user descriptor for the object, if there is one to pin, whether or not any bytes are pinned. “Eof” is set to TRUE and “retDesc” is set to NULL when there are no more objects to be scanned. Each call to sm_ScanNextObject( ) releases the user descriptor returned by the previous scan call, so **sm_ReleaseObject( ) must not be used** on user descriptors that are acquired by scanning files.

```
sm_ScanNextBytes (scanDesc, length)
SCANDESC  *scanDesc;              /* IN scan descriptor */
int        length;                /* IN number of bytes to read */
```

Sm_ScanNextBytes( ) is useful when a file being scanned contains very large objects that cannot be expected to fit in memory. A sm_ScanNextObject( ) call can be made with a relatively small length to read in the first section of an object. Sm_ScanNextBytes( ) is used subsequently to iterate over the rest of that object, with each call reading in the next “length” bytes of the current scan object. The iteration can be controlled by observing the objectSize field of the user descriptor. esmENDOFOBJECT is returned if there are no more bytes to be read in the current object.

```
sm_CloseScan (scanDesc)
SCANDESC  *scanDesc;              /* IN scan descriptor */
```

Sm_CloseScan( ) closes the scan associated with “scanDesc”. It releases the scan descriptor and the user descriptors and data pinned during the scan.

4.9.3. Bulk-loading Files

WARNING: the file bulk load facility does not work properly in version 3.1. We recommend that it not be used.

```
sm_OpenLoad (fid, type, groupSize, fillFactor, loadDesc)
FID        *fid;                  /* IN file to scan */
int        groupSize;             /* IN size of load buffer group */
float      fillFactor;            /* IN fill percentage */
LOADDESC   **loadDesc;            /* OUT returned load descriptor */
```

Sm_OpenLoad( ) prepares to load a set of objects into a file in bulk. Bulk loading a file can be more efficient than using a series of sm_CreateObject( ) calls. The file, indicated by “fid”, need not be empty, in which case the new objects are added to the end of the file. The load mechanism creates and uses its own buffer group; the size of the buffer group is “groupSize”. The “fillFactor” argument is ignored; it is present for future extensions. A *load descriptor*, “loadDesc” is returned for use in subsequent operations (sm_LoadNextObject( ) and sm_CloseLoad( )).

```
sm_LoadNextObject (loadDesc, length, data, oid)
LOADDESC   *loadDesc;             /* IN load descriptor */
int        length;                /* IN length of the object */
void       *data;                 /* IN the object's data */
OID        *oid;                  /* OUT returned new object id */
```

Sm_LoadNextObject( ) creates a new object if size “length” in the file for which the “loadDesc” was opened. The new object is initialized with “data”. If “data” is NULL, the object is filled with zeroes. Sm_LoadNextObject( ) returns an object identifier for the new object in “oid”.

```
sm_CloseLoad (loadDesc)
LOADDESC  *loadDesc;                    /* IN load to close */
```

Sm_CloseLoad( ) ends the bulk-load operation.

4.10. Operations on Indexes

The Storage Manager’s index facility associates keys with fixed-length elements. The keys can be any basic C data type (SM_int, SM_long, SM_short, SM_float, SM_double) or strings (SM_string). The size of the element is fixed when the index is created.

B^+tree index and linear hashing index functions are implemented. B^+tree provides fast index lookup on all kinds of queries, especially range queries. Linear hashing provides even faster index lookup and supports linear space growth for dynamically growing indexes, but it supports only exact-match queries. More information about linear hashing can be found in [Litw88].

A key is fully described by the **KEY** structure:

```
typedef struct {
        TWO  length;         /* length of the key */
        void*  valuePtr;     /* pointer to value of the key */
} KEY;
```

Index keys are compared according to the key type given when the index is created. The key type determines the number of bytes considered in a key comparison. In the case of keys that are strings, the length fields in the keys in question determine the number of bytes compared. Strings are compared one character at a time. The client library does not terminate strings with nulls. When two strings of different lengths are compared, the shorter string is compared with the corresponding substring of the longer string. If the shorter string and the corresponding substring are equal, the longer string is considered to be the larger of the two. This means that "abc " is longer than "abc".

Characters are compared as ASCII values.

4.10.1. Creating and Destroying Indexes

When an index is created, the client library creates a handle, by which the index is identified in subsequent operations. The handle is an *index identifier*, a structure of type IID. The value of the index identifier can be treated as an opaque value by the application.

The following macros can be used it give an illegitimate initial value to an index identifier, and later to recognize that value:

INVALIDATE_IID (IID iid)

sets “iid” to an invalid index identifier.

IID_IS_INVALID (IID iid)

returns TRUE if “iid” has the value given by INVALIDATE_IID( ), FALSE if not.

The rest of this section describes the functions that operate on indexes.

```
sm_CreateIndex(volume, groupIndex, ndxType, keyType, maxKeyLen, elSize, unique, ndx)
VOLID           volume;         /* IN volume on which index is to be built */
int             groupIndex;     /* IN the buffer group to use */
SMTYPE          ndxType;        /* IN SM_BTREENDX, SM_HASHNDX, etc */
SMDATATYPE      keyType;        /* IN SM_int, SM_long, SM_string, etc */
int             maxKeyLen;      /* IN maximum key length of a key in the index */
int             elSize;         /* IN element size (mpl of 4, < SM_MAXELEMLEN) */
BOOL            unique;         /* IN TRUE if key is unique */
IID*            ndx;            /* OUT returned index identifier */
```

Sm_CreateIndex( ) creates an index that resides on “volume”. [3] “NdxType” specifies the type of index (SM_BTREENDX for B^+tree or SM_HASHNDX for linear hashing). “KeyType” indicates the data type of the key. The maximum length of a key in the index is given in “maxKeyLen”. The size of the elements in the index is given in “elSize”. The element size must be a multiple of four and less than SM_MAXELEMLEN (20). If “unique” is FALSE, the index is able to store multiple elements under the same key. An index identifier is returned in “ndx” upon successful completion.

```
sm_DestroyIndex(ndx, groupIndex)
IID*            ndx;            /* IN id of index to destroy */
int             groupIndex;     /* IN which buffer group to use */
```

Sm_DestroyIndex( ) destroys the index associated with “ndx”.

```
sm_SetLHashLoadThreshold(ndx, groupIndex, load)
IID*            ndx;            /* IN index identifier */
int             groupIndex;     /* IN which buffer group to use */
float           loadFactor;     /* IN the load factor to use for linear hashing */
```

Sm_SetLHashLoadThreshold( ) changes the load factor for a linear hashing index from the default 75% to the given “loadFactor”. The default load factor, 75%, yields the best access time and space utilization. See [Litw88] for information about linear hashing and when it might be useful to change the load factor. The load factor can be set only on a newly created index.

4.10.2. Inserting and Removing Index Elements

[3] Indexes on temporary volumes are not implemented. (Section 5.1.3, **Temporary Volumes**). If the volume given is temporary, sm_CreateIndex( ) returns esmFAILURE, with error code esmNOTIMPLEMENTED.

```
sm_InsertEntry(ndx, groupIndex, key, elem)
IID*            ndx;            /* IN index identifier */
int             groupIndex;     /* IN which buffer group to use */
KEY*            key;            /* IN key to insert */
void*           elem;           /* IN element associated with key */
```

Sm_InsertEntry( ) inserts a <key, elem> pair into the index “ndx”. If “ndx” is a unique index and the key to be inserted already appears in the index, sm_InsertEntry( ) returns an error in sm_errno. If the index is not unique, there is no limit to the number of duplicate keys as long as different elements are associated with them.

```
sm_RemoveEntry(ndx, groupIndex, key, elem)
IID*            ndx;            /* IN index identifier */
int             groupIndex;     /* IN which buffer group to use */
KEY*            key;            /* IN key to remove */
void*           elem;           /* IN element associated with key */
```

Sm_RemoveEntry( ) removes a <key, elem> pair from the index “ndx”.

4.10.3. Loading Indexes in Bulk

The Storage Manager provides a bulk-load facility for efficiently loading an empty index. When the application begins a bulk-load operation, the client library allocates a temporary run-buffer, which is used for sorting runs. Henceforth, the application uses sm_InsertEntry( ) repeatedly to load elements into index; no other index operations are allowed during a bulk-load. Each sm_InsertEntry( ) operation for the index inserts a <key, elem> pair into the temporary run buffer. The run buffer is sorted and written to the work file as a “sorted-run” when it is full. When the application terminates the bulk-load operation, the client library merges the sorted-runs into a sorted stream, from which the index is built from the bottom, up.

Entries cannot be removed during a bulk-load operation.

```
int sm_BeginIndexLoad(ndx, groupIndex, workVolume, runSize)
IID*            ndx;            /* IN index identifier */
int             groupIndex;     /* IN the buffer group to use */
VOLID           workVolume;     /* IN work volume */
int             runSize;        /* IN size of each sorted run in pages */
```

Sm_BeginIndexLoad( ) prepares to load the index given in “ndx”, using the buffer group “groupIndex”. Sm_BeginIndexLoad( ) uses the volume named by “workVolume” for the sorted runs. Using a temporary volume for the work volume yields the best performance (see Section 5.1.3, **Temporary Volumes**).

The “runSize” argument determines how many MIN_PAGESIZE pages to fill before ending a run. The larger “runSize”, the more memory is consumed by the bulk-load, with a commensurate improvement in speed. Sm_BeginIndexLoad( ), if it is used, must be the first operation performed on an index.

int sm_EndIndexLoad(ndx)

IID*	**ndx;**	**/* IN index identifier */**

Sm_EndIndexLoad( ) concludes the bulk-load and builds the index.

int sm_AbortIndexLoad(ndx)

IID*	**ndx;**	**/* IN index identifier */**

sm_AbortIndexLoad( ) aborts the bulk-loading of an index. All resources used by the index are freed.

4.10.4. Scanning Indexes

Indexes are used by posing queries with the sm_FetchInit( ) operation. A query requests all the elements whose key values lie in a range. The results of the query are fetched, one element at a time, with the sm_FetchNext( ) operation. An index scan uses a *cursor*, a value of the type SMCURSOR. A cursor can be treated by the application as an opaque value. The following two macros give a cursor an invalid initial value and recognize that value:

INVALIDATE_CURSOR (SMCURSOR cursor)

sets “cursor” to an invalid index scan cursor.

CURSOR_IS_INVALID (SMCURSOR cursor)

returns TRUE if “cursor” is the value given by INVALIDATE_CURSOR( ), FALSE if not.

The rest of this section describes the functions used to scan indexes.

sm_FetchInit(ndx, groupIndex, bound1, cond1, bound2, cond2, cursor)

IID*	**ndx;**	**/* IN index identifier */**
int	**groupIndex;**	**/* IN which buffer group to use */**
KEY*	**bound1;**	**/* IN starting bound of the scan */**
SMCOND	**cond1;**	**/* IN starting condition */**
KEY*	**bound2;**	**/* IN ending bound of the scan */**
SMCOND	**cond2;**	**/* IN ending condition */**
SMCURSOR*	**cursor;**	**/* OUT returned pointer if non-NULL */**

Sm_FetchInit( ) begins a scan on the index “ndx”. The arguments “bound1” and “cond1” specify the beginning search condition. “Bound2” and “cond2” specify the ending search condition. The conditions can be SM_EQ, SM_G, SM_L, SM_GEQ, or SM_LEQ. The “cursor” argument is initialized by sm_FetchInit( ) and used by sm_FetchNext( ). The caller is responsible for allocating the space for the cursor and the client library is responsible for the value of the cursor.

The direction of the scan (ascending or descending) is determined by the bounds and conditions of the query. The beginning and end of an index are specified with the macros SM_BOF and SM_EOF. For linear hashing indexes (type SM_HASHNDX), the value that can be used for “cond1” and “cond2” is SM_EQ.

Several examples of queries follow:

(1) Scan from key1 = “10” to key2 = “30” inclusively:
sm_FetchInit(..., key1, SM_GEQ, key2, SM_LEQ, cursor) --- ascending

sm_FetchInit(..., key2, SM_LEQ, key1, SM_GEQ, cursor) --- descending

(2) Scan from key1 = “10” to the end of the index:
sm_FetchInit(..., key1, SM_GEQ, SM_EOF, cursor) --- ascending
sm_FetchInit(..., SM_EOF, key1, SM_GEQ, cursor) --- descending

(3) Scan the whole index:
sm_FetchInit(..., SM_BOF, SM_EOF, cursor) --- ascending
sm_FetchInit(..., SM_EOF, SM_BOF, cursor) --- descending

sm_FetchNext(cursor, retKey, retElem, eof)

SMCURSOR*	**cursor;**	**/* IN cursor from sm_Fetch( ) */**
KEY*	**retKey;**	**/* OUT returned key (optional) */**
void*	**retElem;**	**/* OUT elem */**
BOOL*	**eof;**	**/* OUT to TRUE if EOF reached */**

Sm_FetchNext( ) fetches the next element returned by a query. The element is returned in the structure addressed by “retElem”. A copy of the key can also be returned to the caller. If “retKey” is NULL, no key is returned. If “retKey” points to a key structure, the key is returned in that structure. The “length” field in the key structure must indicate amount of space available in the target of the “valuePtr” field. This must be enough for the longest key in the index. The caller is responsible for allocating space for “retKey” and “retElem”.

sm_FetchNext( ) returns FALSE in “eof” if an element is returned. If there are no more elements that satisfy the query, TRUE is returned in “eof”.

4.11. Advanced Topics

4.11.1. External Two-Phase Commit Functions

The Storage Manager can particpate in transactions coordinated by other software modules that employ the two-phase commit “presumed abort” transaction semantics and protocol. (For the purpose of this section, the reader is assumed to be familiar with the “presumed abort” protocol.) The coordinator in such a situation is external to the Storage Manager; it is assumed to have its own stable storage, and it is assumed to recover from failures in a *short time* (the precise meaning of which is given forthwith).

A prepared transaction, like an active transaction, consumes log space on one or more Exodus servers, beginning at a fixed location in each log. A Storage Manager server’s log is like a circular buffer; it wraps and reuses the beginning of the log. If long-running or prepared transactions are still in the system, the server eventually tries to re-use log space consumed by the oldest transaction, at which point it effectively runs out of log space. A coordinator must resolve its prepared transactions before the servers run out of log space. The amount of time involved is a function of the size of the log on the participating servers and the load on those servers.

For the purpose of this discussion, the portion of a global transaction that involves a single Exodus Storage Manager transaction is called a *thread* of the global transaction. Each thread has, in addition to its local transaction identifier, a global transaction identifier. Global transaction identifiers are provided by the application or some external authority, and must be unique. A global transaction identifier has type GTID, defined in `sm_client.h`, as follows:

```
#define MAXOPAQUELEN 255
typedef struct {
    int         length;    /* maximum MAXOPAQUELEN bytes */
    u_char      opaque[MAXOPAQUELEN];
} GTID;
```

The Storage Manager does not interpret the contents of the opaque part of the global transaction identifier.

An application that invokes the external two-phase commit protocol can find itself in any of the transaction states mentioned in Section 4.3.2 ("Transaction States"). It can also find itself in the PREPARED state after a call to sm_PrepareTransaction( ). An application in PREPARED state calls sm_CommitTransaction( ) or sm_AbortTransaction( ) to complete the transaction and return to the INACTIVE state.

While the coordinator for a global transaction is external to the Storage Manager, a single Storage Manager server corresponds with the client library and coordinates the Storage Manager servers that participate in the thread. If the application should crash during a two-phase commit, a new application program (representing the global coordinator) must run, and it must contact the Storage Manager that is acting as the thread's coordinator. In order to locate the proper server, a two-phase commit process begins by informing the client library that a transaction is a thread of a global transaction, and by identifying the thread's coordinator. The function sm_Enter2PC( ), described below, accomplishes this.

```
sm_Enter2PC (tid, gtid, handle)
TID             tid;                /* IN transaction ID */
GTID            *gtid;              /* IN global transaction ID */
COORD_HANDLE  *handle;              /* OUT for use if client crashes */
```

The application supplies the local and global transaction identifiers. The client library identifies a thread coordinator, and produces a handle for the application to write to stable storage. The handle identifies the thread coordinator; it is used by sm_Recover2PC( ) if the client crashes before the two-phase commit is completed.

The handle must be written to stable storage before the first phase of the commit begins, otherwise the application and Storage Manager may not be able to recover from a subsequent application failure.

```
sm_PrepareTransaction (tid, vote)
TID             tid;                /* IN transaction ID */
VOTE            *vote;              /* OUT result of first phase */
```

The application calls sm_PrepareTransaction( ) to begin the first, or prepare, phase of a two-phase commit. sm_PrepareTransaction( ) determines if the participating servers are able to commit the transaction, and directs them to prepare to commit if they are. If any of the participating servers is unable to commit the transaction, the vote returned is NOVOTE, sm_PrepareTransaction( ) sets sm_error to esmTRANSABORTED, sm_reason to esmTRANSNOTPREPARED, and returns esmFAILURE; the application must call sm_AbortTransaction( ).

If all participating servers are able to commit, and any of them logged updates during the transaction, the vote is YESVOTE, and the transaction state becomes PREPARED. If the transaction did not update any data on any of the servers, the vote is READVOTE, and the transaction state becomes INACTIVE. Sm_PrepareTransaction( ) returns esmNOERROR if the transaction becomes prepared (all servers vote YESVOTE) or committed (all server vote READVOTE).

If an error occurs during the prepare phase, sm_PrepareTransaction( ) returns esmFAILURE. If it is a recoverable error, the client library returns an error code specific to the error in sm_errno (such as esmTRANSDISABLED if a server is performing recovery), and the application can try again to call sm_PrepareTransaction( ). Some errors, on the other hand, cause the transaction to be aborted, in which case sm_PrepareTransaction( ) returns esmTRANSABORTED in sm_errno, and a vote of NOVOTE.

If an application crashes during the first phase, the application must retry the prepare phase and complete the transaction. If it does not retry the prepare phase, and the transaction was indeed prepared before the application crashed, the prepared transaction consumes resources indefinitely, and eventually its servers will run out of log space.

Once a transaction is prepared, an application must invoke the second phase by aborting or committing the transaction (calling sm_AbortTransaction( ) or sm_CommitTransaction( ), respectively). It is an error to commit a global transaction thread without first preparing the transaction, and it is an error to do anything else without completing the second phase.

When an error occurs during the second phase, the application cannot tell if the second phase completed (the transaction indeed committed or aborted). It is alway safe to try again to complete the transaction by calling sm_AbortTransaction( ) or sm_CommitTransaction( ) again.

If the second phase fails because the network connection between the client and the thread coordinator breaks (esmSERVERDIED or esmNOTCONNECTED), the client must reconnect to the thread coordinator before the second phase can be finished. The following function does that:

```
sm_Continue2PC (tid, willing2block)
TID         tid;               /* IN transaction ID */
BOOL        willling2block;    /* IN ok to block indefinitely */
```

If “willing2block” is TRUE, the client library blocks until it connects to the thread coordinator. If this is inappropriate for the application, “willing2block” must be FALSE, and the client library tries once to contact the thread coordinator.

If the application crashes, its replacement must use sm_Recover2PC( ), below, instead of sm_Continue2PC( ) to resolve the transaction.

```
sm_Recover2PC (gtid, handle, willing2block, tid)
COORD_HANDLE  *handle;         /* IN handle for thread coordinator */
GTID          *gtid;           /* IN global transaction ID */
BOOL          willing2block;   /* IN ok to block indefinitely */
TID           *tid;            /* OUT local transaction ID */
```

When the application crashes (exits) after a transaction is prepared but before its second phase is completed, a “recovery” application program must be run within a short time to finish the two-phase commit and resolve the transaction. This recovery application must use sm_Recover2PC( ), supplying the global transaction identifier and the handle returned by sm_Enter2PC( ) for that global transaction. The client library contacts the server identified in the handle, which conveys to the client library all that is needed for the application to enter or to retry the second phase. The transaction's local transaction identifier is returned by sm_Recover2PC( ) for the application to use in its subsequent call to sm_CommitTransaction( ) or sm_AbortTransaction( ).

The thread coordinator may not be available, in which case the client library keeps trying to connect or it will return an error (such as ECONNREFUSED), depending on the value of "willing2block". If "willing2block" is FALSE, the client library tries only once to connect the thread coordinator.

4.11.2. Administrative Operations

The following functions can be applied to one or more servers. Each function takes two arguments that determine which servers are of interest. The first argument is of type FLAGS, and takes one of the following values:

```
VOL_ALL      /* the servers for all volumes */
VOL_USED_SINCE_INIT    /* servers for all volumes used */
VOL_USED_IN_TRANSACTION   /* servers used in this transaction  */
VOL_BY_VOLID      /* the second argument applies */
```

The client library keeps a list of volumes and the servers that manage those volumes. The list is created from the information given in the configuration files and information passed to the library through sm_SetClientOption( ), The flag VOL_ALL directs the client library to apply the administrative operation to the server that manages each volume in its list of known volumes. The flag VOL_USED_SINCE_INIT directs the client library to apply the administrative operation to each server contacted since sm_Initialize( ) was called. The flag VOL_USED_IN_TRANSACTION directs the client library to apply the administrative operation to each server contacted so far for participation in the current transaction. (It does not apply to servers to be contacted for the first time later in the transaction.) The flag VOL_BY_VOLID directs the client library to apply the administrative operation to the server that manages the volume identified by the second argument. The second argument is a volume identifier VOLID, which is ignored when the flags argument is VOL_ALL, VOL_USED_SINCE_INIT, or VOL_USED_IN_TRANSACTION.

Ideally the administrative operations would only be performed by trusted clients, but the Storage Manager does not restrict their use.

sm_TakeCheckpoint (flags, volid, numCheckpoints)

FLAGS	**flags;**	**/* IN which servers are of interest */**
VOLID	**volid;**	**/* IN which server is of interest */**
short	**numCheckpoints;**	**/* IN number of checkpoints to take */**

Sm_TakeCheckpoint( ) sends a request to the server to take a number of checkpoints. In most circumstances, a value of one for the "numCheckpoints" argument is appropriate. A value greater than 1 can be used to ensure that the server flushes all pages that were dirty when the first checkpoint was taken. (This is useful for experimenting with the recovery facility).

sm_ChangeCheckpointFrequency (flags, volid, frequency)

```
FLAGS          flags;          /* IN which servers are of interest */
VOLID          volid;          /* IN which server is of interest */
int            frequency;      /* IN number of log records between checkpoints */
```

Sm_ChangeCheckpointFrequency( ) changes the frequency of checkpoints taken by the server. The checkpoint frequency is based on the number of log pages written. More information about checkpoint frequency can be found in Section 5.3, **Tuning the Server**.

sm_ShutdownServer (flags, volid, options)

```
FLAGS          flags;          /* IN which servers are of interest */
VOLID          volid;          /* IN which server is of interest */
FLAGS          options;        /* IN shutdown options */
```

Sm_ShutdownServer( ) directs servers to shut down. The “options” argument indicates what a server should do before exiting. The following flags are available: NOFLAGS, SHUT_TAKE_CHECKPOINT, SHUT_DUMP_CORE, SHUT_ABORT_TRANS, SHUT_COMMIT_TRANS, SHUT_CLEAN_VOLUMES. These can be combined with the logical “or” operator.

If NOFLAGS is given, the server kills the disk processes and exits.

SHUT_TAKE_CHECKPOINT directs the server to take a checkpoint before exiting.

SHUT_DUMP_CORE directs the server to dump a core file debugging (see core(5)).

SHUT_COMMIT_TRANS directs the server to wait until the running transactions either commit or abort before it shuts down.

SHUT_ABORT_TRANS directs the server to abort all running transactions before shutting down. When SHUT_COMMIT_TRANS or SHUT_ABORT_TRANS is used, clients cannot start any new transactions.

SHUT_CLEAN_VOLUMES directs the server to write dirty pages to disk before exiting. To shut down a server after which recovery is not required, use either SHUT_COMMIT_TRANS | SHUT_CLEAN_VOLUMES or SHUT_ABORT_TRANS | SHUT_CLEAN_VOLUMES.

sm_ServerStatistics (flags, volid, numServers, stats, reset)

```
FLAGS          flags;          /* IN which servers are of interest */
VOLID          volid;          /* IN which server is of interest */
int            *numServers;    /* OUT # servers contacted */
SERVERSTATS    **stats;        /* OUT servers’ statistics */
BOOL           reset;          /* IN TRUE = reinitialize counters */
```

Sm_ServerStatistics( ) obtains statistics about one or more servers. For each server contacted, a set of statistics is returned. The client library allocates space for the statistics, and the **application is responsible for freeing that space** (see the manual page for malloc(3)). The “flags” indicate which servers are of interest, and the number of servers contacted is returned in “*numServers”. On return from sm_ServerStatistics( ), the “*stats” pointer addresses an array of “*numServers” SERVERSTATS structures. This array must be freed by the application with one call to *free(3)*.

If “reset” is TRUE, the statistics labeled as counters below are reset to zero.

The SERVERSTATS structure looks like this:

```
typedef struct {
    int          numClients;         /* # clients connected */
    int          numTrans;           /* # transactions in progress */
    int          numVolumes;         /* # volumes mounted */
    int          freeLogSpace;       /* approximate # bytes free log space */
    int          chpntFreq;          /* checkpoint frequency */
    int          totalCommits;       /* # transactions committed */
    int          totalAborts;        /* # transactions aborted */
    int          diskReads;          /* # disk reads */
    int          diskWrites;         /* # disk writes */
    MESSAGESTATS       msgStats;     /* server’s message counters */
} SERVERSTATS;
```

The MESSAGESTATS structure contains statistics about the client-server protocol and the server-server protocol. A set of these statistics is kept by the client library a set is kept by each server. The client library’s statistics are found in the global structure

extern MESSAGESTATS MsgStats;

The MESSAGESTATS structure contains the following counters for each message type: messages sent, messages received, replies received with an error indication, replies received with no error, messages sent with no reply requested. The counters for replies have two different meanings, depending on which set statistics is concerned. The servers count the replies *sent* with and without error indications, and the number of requests that the server *received* that did not require a reply at all. The client library counts the replies *received* with and without error indications, and the number of requests that the client *sent* that did not require a reply at all.

The following function prints the MESSAGESTATS structure:

```
sm_PrintMessageStats (file, stats)
FILE             *const      file;             /* IN where to print */
MESSAGESTATS     *const      msgStats;         /* IN what to print */
```

The following function tells if a mounted volume is temporary volume, a data volume, or a log volume. See Section 5.1, **Managing Volumes**, for information about volumes.

```
sm_VolumeProperties (volid, properties)
VOLID            volid;              /* IN which volume is of interest */
int              *properties;        /* OUT the properties */
```

Sm_VolumeProperties( ) returns a set of bits that tell whether the given volume is a data volume or a temporary volume. The “volid” argument is the volume identifier of the volume in question. If the volume is not mounted when Sm_VolumeProperties( ) is called, Sm_VolumeProperties( ) mounts it.

VOLPROP_TEMP indicates that the volume is temporary (see Section 5.1.3, **Temporary Volumes**). If the bit VOLPROP_TEMP is not set in the result, the volume is a data volume. A log volume cannot be mounted by a client, and an attempt to get a log volume’s properties results in an error.

```
sm_AddServerVolume (flags, volid, option, value)
FLAGS           flags;          /* IN which servers are of interest */
VOLID           volid;          /* IN which volume is of interest */
char       *option;             /* IN which format option to use */
char       *value;              /* IN value for the format option */
```

Sm_AddServerVolume( ) adds a volume to the list of mountable volumes on one or more servers (although it seldom makes sense to do this on more than one server with a single pair of arguments). The “flags” argument indicates which servers are of interest. The “volid” argument is the volume identifier of the volume that will determine which server to contact when “flags” == VOL_BY_VOLID. The “option” is one of the server’s format options (“dataformat” or “tempformat”). The “value” argument is the value to be given the option named in “option”.

Sm_AddServerVolume( ) adds the named volume to the server’s list of known volumes, but the server does not try to mount the volume or verify that the volume exists or is valid. Sm_AddServerVolume( ) fails if the value given conflicts with another volume already in the server’s table, either in the path name or the volume identifier. If your objective is to change the format information for a path name that is in the server’s table, first remove the existing format information (using sm_RemoveServerVolume( ), described below), and subsequently add the new information.

```
sm_RemoveServerVolume (flags, volid, volid2remove)
FLAGS           flags;          /* IN which servers are of interest */
VOLID           volid;          /* IN which volume id of server of interest */
VOLID           volid2remove;   /* IN which volume to remove */
```

Sm_RemoveServerVolume( ) removes “volid2remove” from one or more servers’ lists of mountable volumes. The volume cannot be removed from a server’s table while the volume is in use. it must be dismounted before it is removed.

See also Section 5.1, **Managing Volumes**.

4.11.3. Tuning the Application

The size of the application’s buffer pool, determined by the “bufpages” option, is the primary tuning parameter that is under the control of applications. The “bufpages” option indicates the number of MIN_PAGESIZE pages in the buffer pool. It should be set large enough to hold the application’s working set of objects. The buffer pool must not exceed the size of physical memory available to the client.

5. USING STORAGE MANAGER SERVERS

Storage Manager servers provide disk, file, transaction, concurrency control, and recovery services to clients. In most respects, users do not have to understand how servers work, but there are a few things that administrators should know; we focus on those things in this section. The first half of this section explains how to manage volumes. The second half explains how to operate a server.

5.1. Managing Volumes

Servers store data on *volumes,* which can be Unix files or raw disk partitions. Each server is composed of a server process and one *disk process* for each mounted volume. When a server requires I/O, it asks the appropriate disk process to read from or write to the server's buffer pool, which is located in a Unix System V shared-memory segment. The disk processes perform I/O so that the server never blocks when I/O is required. The server mounts a volume before using it, and the server dismounts the volume when it is no longer in use. Mounting a volume consists in forking a disk process for that volume. Dismounting the volume consists in flushing all dirty pages to the disk and killing the volume's disk process.

Volumes are created with the `formatvol` program, which establishes a volume's identifier, size, type, and other characteristics. Volumes come in three types: log volumes, data volumes, and temporary volumes.

5.1.1. Log Volumes

Log volumes are used to store log information for aborting transactions and for recovery. The server has one log volume mounted at all times.

5.1.2. Data Volumes

Data volumes are used to store objects and indexes that are meant to exist after a transaction ends. Changes to data volumes are logged so that transactions can be aborted or committed with reliability, and so that recovery can be performed after a crash.

5.1.3. Temporary Volumes

Some applications store temporary private data and do not need concurrency control or recovery. The Storage Manager provides temporary volumes for this purpose. Locks are not acquired for data in temporary volumes, and updates to temporary volumes are not logged. Temporary volumes are less costly to use than data volumes are, but the data on them cannot be shared among transactions. The data on temporary volumes are deleted at the conclusion of the transaction that creates them, regardless of whether the transaction is committed or aborted. Temporary volumes cannot contain root entries.

The server can serve many data volumes and temporary volumes simultaneously.

5.1.4. Raw Partitions and Unix Files

A volume can be a Unix file or a Unix raw partition. When a raw partition is used, data are transferred between the server's buffer pool and the disk by the disk process, bypassing the Unix file system's buffer pool.

When a Unix file is used, the data are written to the Unix file system's buffer pool, and the operating system worries about flushing the data to the disk. In this case, the server forces the data to the disk periodically with a Unix *fsync( )* system call.

5.1.5. Formatting Volumes

Before a volume can be used, it must be formatted. This is done using the `formatvol` program, which can also display information about previously formatted volumes. Formatvol uses the configuration options "dataformat", "tempformat", and "logformat" to determine what characteristics to give volumes that it formats. The options have values that list the following information:

path	The Unix path name of the volume, e.g., `/dev/rz2c`.
volid	The volume identifier for this volume, an integer, e.g., 8000.
#cyl	The number of cylinders on this disk, e.g., 1224 for a DEC RZ55. May be 1.
#trk/cyl	The number of tracks per cylinder e.g., 15 for a DEC RZ55. May be 1.
#sect/trk	The number of sectors or blocks per track e.g., 36 for a DEC RZ55. May be the number of *blocks* in the file. A block is MIN_PAGESIZE bytes; MIN_PAGESIZE is defined in `sm_client.h`. (This is determined by the Storage Manager, not by the device.) [4]
#KB/pg	**For logformat only**. This gives the page size for log pages, in kilobytes. The value given here may be 4 or larger, and must be a power of 2.

Formatvol collects the format information from the options in the configuration files, after which it determines which volumes to format or to display by processing the options "volume" and "display" from the command line. The options that formatvol understands are summarized in Table 2.

Option Name	Option Type	Option Description
tempformat	string,int,int,int	path,volid,#cyl,#trk/cyl,#sect/trk
dataformat	string,int,int,int	path,volid,#cyl,#trk/cyl,#sect/trk
logformat	string,int,int,int,int	path,volid,#cyl,#trk/cyl,#sect/trk,#KB/pg
volume	int	volume to format - command line only
display	int	volume to display - command line only

Table 2: Formatvol Options.
Fields are separated by white space, commas, colons or semicolons.

For example, to print information about the volumes with volids 8000 and 4000 use:

```
    formatvol -dis 8000 -dis 4000
```

[4] The format of a volume does not affect performance with most modern disks. The easiest way to format volumes it to use use 1 cyl, 1 track/cyl, and let the sect/trk account for the size of the entire volume.

To format a data volume with volid 8000 and a temporary volume with volid 4000 use:

```
      formatvol -vol 8000 -vol 4000
```

Formatting a volume writes a volume header and initializes the bitmaps that describe the free blocks on the volume. A volume that is reformatted after being used loses all its data.

The Storage Manager does not prevent a volume from being formatted while it is in use by a server, even though **it will cause the server to crash unrecoverably**. Be certain that a volume is not mounted before you format it! [5] A volume is unmounted when all clients that are using the volume have completed transactions on it and have unmounted it. (A client may unmount a volume explicitly with sm_DismountVolume( ), or by shutting down with sm_ShutDown( ) or *exit( )*.)

During recovery, a server mounts the volumes that need recovery. The volumes are dismounted when recovery is completed. If a volume was in use at the time its server crashed, **do not reformat the volume until a new server recovers the data on that volume**. If you do, the server's log will be inconsistent with the data on the volume, and the server will crash during recovery, and it will be unable to recover from that. You can reformat the data volumes and the log volume to get a server running again, but you will have lost all data on the volumes.

The log volume is mounted whenever the server is running, so a log volume can be formatted ONLY when the server is not running.

5.1.6. Size Requirements for Log Volumes

How large should a log volume be? The answer depends on the expected transaction mix. More specifically, it depends on the age of the oldest (longest running) transaction in the system and the amount of log space used by all active transactions. Here are some general rules to determine the amount of free log space available in the system.

(1) The physical log is circular. Log space between the first log record generated by the oldest active transaction and the most recent log record generated by any transaction cannot be reused.

(2) Log space for a transaction is available for reuse when the transaction has committed or completely aborted. Aborting a transaction causes log space to be used, so space is *reserved* for aborting each transaction. Enough log space must be available to commit *or abort* all active transactions at all times.

(3) Only space starting at the *beginning* of the log can be reused. This space can be reused if it contains log records only for transactions meeting rule 2.

(4) All sm_WriteObject( ) calls require log space twice the size of the space written in the object. All calls that create, grow, or shrink objects require log space equal to the size created, inserted, or deleted. Log records generated by these calls (generally one per call) have an overhead of approximately 50 bytes.

(5) File operations are logged, but the space requirements for them are most often negligible, since they are relatively rare operations, and are often performed in short

[5] The Storage Manager ought to lock volumes with Unix file locks, but Unix does not provide an adequate mechanism for locking and unlocking files in the context of crash recovery.

transactions.

(6) The amount of log space *reserved* for aborting a transaction is equal to the amount of log space generated by the transaction (for the purpose of committing the transaction).

(7) When insufficient log space is available for a transaction, the transaction is aborted.

(8) The log should be at least 1 Mbyte (250 pages).

For example, consider a transaction T1, which creates 300 objects of size 2,000 bytes, writes 20 bytes in 100 objects, and is committed. T1 requires at 615 Kbytes for the creates and 9 Kbytes of log space for the writes. Since log space must be reserved to abort the transaction, the log size must be over 1.248 Mbytes to run this transaction. Assuming T1 is the only transaction running in the system, all the log space it uses and reserves becomes available when it completes. If another transaction, T2, is started at the same time as T1, but is still running after T1 is committed, only the reserved space for T1 is available for other transactions. The portion of the log used by T1 and T2 is not available until T2 is finished.

Transactions that fail because of insufficient log space are commonly those that load a large number of objects into a file during the creation of a database. A solution to this problem is to load the file in a series of smaller transactions. When the last transaction is committed, the load is complete. If the load needs to be aborted, a separate transaction is run to destroy the file.

5.1.7. Backing Up Volumes

The Storage Manager does not support media recovery, so backing up critical data volumes is wise. A volume may be backed up when it is unmounted and needs no recovery. If a volume is stored on a Unix file, a simple copy of the file can be used as a backup. For volumes stored on a raw disk partition, the Unix *dd(1)* command can be used to backup the volume to a Unix file and to restore it. For example, to save a copy of the raw device `/dev/rrz4d` in the Unix file backup.rrz4d use:

```
dd if=/dev/rrz4d of=backup.rrz4d.
```

To restore the backup, use:

```
dd if=backup.rrz4d of=/dev/rrz4d.
```

5.2. Using the Server

In this section we explain how to operate a Storage Manager server. For the purpose of this discussion, we use only one server, although any number of servers can be used to manage any number of volumes. We begin with starting and configuring the server. Next, we discuss what the server does during normal operation. We follow this with instructions for shutting the server down. Finally, we explain how the server recovers from failure.

5.2.1. Starting the Server

The server is composed of two executable files: `sm_server` and `diskrw`. `Sm_server` is the main server program. `Diskrw` is started by the server, as a separate process for each mounted volume, for performing asynchronous disk I/O. These processes communicate with the server through sockets, semaphores, and shared memory. By default, the server assumes `diskrw` is located in the user's path. An option, described below, can be used to change this assumption.

When the server is started, it processes configuration options. These options are discussed further below. Second, the server allocates the buffer pool. The buffer pool is located in shared memory, so the operating system must have shared-memory support. Furthermore, the machine on which the server runs must have enough shared memory to accommodate the entire buffer pool. If not enough shared memory is available, the server prints a message, indicating how much shared memory it is trying to acquire, and exits.

Third, the server mounts the log volume. If the log volume is newly formatted, it is *regenerated*. When a log volume is regenerated, the entire log is cleared and written to disk. This will take noticeable time if the volume is large. If the log is not regenerated, recovery analysis is performed.

If no volumes require recovery, all phases of recovery complete in less than one second. If the analysis determines that any volumes require recovery (due to a previous failure of some sort: operating system failure, machine failure, internal error, or because a user killed the server), recovery is performed. Data volumes that were mounted at the time of the failure are remounted, updates by committed transactions are restored, and all transactions in progress at the time of failure are aborted. When recovery is complete, the data volumes are dismounted and a checkpoint is taken.

The server now begin to process requests from clients.

5.2.2. Configuring the Server

There are several *configuration options* that can be set when the server is started. A brief description of the options is given in Table 3. Most options have default values, but some do not, and these *must* be given values, either on the command line or in a configuration file. See Section 3 for general information that applies to all options.

Option values are read from the the default configuration files `/usr/lib/sm_config`, `$HOME/.sm_config`, and `./.sm_config` in that order, if they exist. If the command-line option “skipdefault” is given, these default files are not read.

Options on the command line are read after the default files are read. Command-line options are prefixed by a “-”. In addition to options, a server accepts the command-line *flags* given in Table 4. Command-line flags are prefixed by a “-”.

When given the “help” flag, a server prints a list of the available options and flags, and exits.

The “skipdefault” flag prevents a server from reading the default configuration files. It must be the first argument on the command line if it is used.

The “force” flag prevents a server from checking with the user before regenerating the log.

The “background” flag causes the server to disconnect from its controlling terminal. This flag is available for users who run the server from shells that, like the Bourne shell, do not have real job control.

We now describe each option from Table 2.

The “config” option specifies a configuration file to read after default configuration files have been read. This option is effective only on the command line.

The “verbose” option is used to turn on and off printing of the option values at startup. Options are printed to the file specified by “errorfile” option (q.v.).

Option Name	Option Type	Possible Values	Default Values	Option Description
config	string	file name	/usr/lib/sm_config $HOME/.sm_config ./.sm_config	read a configuration file defaults is read unless skipdefault is set
verbose	Boolean	yes no	no	print configuration options
bufpages	int	> 32	none	number of buffer pool pages
logvolume	string	path name	none	name of the log volume
portname	string	name or number	exodussm	port name or port number for a server; if a name, it must be in `/etc/services`
errorfile	string	file name	- (stderr)	file for errors, warnings, progress
regenlog	Boolean	yes no	no	clear the log,
shutdown	Boolean	yes no	no	shut down after recovery or regeneration of log
checkpoints	int	> 1	100	checkpoint frequency (based on number of log pages)
diskproc	string	file name	/usr/lib/exodus/diskrw	disk I/O program name
intercache	Boolean	yes no	yes	allow caching of pages at the client between transactions
progress	Boolean	yes no	no	control progress printing
maxclients	int	> 0	20	maximum number of clients to be served simultaneously
maxthreads	int	> 1	function(maxclients)	maximum number of threads.
traceflags	int	hex number	0x0	set tracing flags. Available if server is compiled with -DDEBUG.
tempformat	string			see Table 2.
dataformat	string			see Table 2.
logformat	string			see Table 2.
maxaddvolumes	int	small number >= 0	0	increases volume table size
wrapcount	int	>=0	0	starting wrap count for log

Table 3: Server Options

Flag Name	Flag Effect
help	print a message and exit
skipdefault	do not read default configuration files must be the first argument on the command line
force	do not confirm log regeneration option
background	put in background (for use with Bourne shell)

Table 4: Server Command-Line Flags

The “bufpages” option indicates the number of MIN_PAGESIZE pages to be used for a server’s buffer pool. The option must be given for a server to run. This option determines the size of the shared memory segment allocated by the server. The shared memory segment will be

MIN_PAGESIZE*bufpages bytes long plus a few KB extra. Section 5.3, **Tuning the Server**, for more information about setting this option.

The “logvolume” option gives the path name of the volume that contains the log. A value must be given for the log volume.

The “portname” option indicates a port number or the symbolic name of a port entry in `/etc/services`. The server connects to this port and listens for client requests on it. To enable clients to locate a server with a symbolic port name, the port name must to present in `/etc/services` on both the client and server machines. If no port name is given, a server looks for an entry “exodussm”, registered for use with TCP, in `/etc/services`.

By using port numbers instead of symbolic names avoids the need for entries in `/etc/services`. See the Unix manual page for services(5). An example entry for the default server name is:

```
exodussm    1152/tcp              # exodus storage manager
```

The “errorfile” option directs server error messages and diagnostics to the given file. A value of “-” means that *stderr* is used.

The “regenlog” option causes the log on the log volume to be regenerated. **This overwrites all log records, so it should not be done unless the server was last shut down cleanly**. Server automatically regenerate their logs when they are started with a newly formatted log volumes. When the option is set to “yes”, a confirmation is requested. The confirmation can be disabled by starting the server with the “force” option.

The “shutdown” option causes a server to shut down immediately after performing recovery or regenerating the log.

The “checkpoints” option sets the checkpoint frequency for a server. The value represents the number of log pages written between checkpoints.

The “progress” option causes a server to print messages tracing its progress. This is used for debugging; it slows the server.

The “diskproc” option specifies the path name of the disk I/O program to be used by the server.

The “intercache” option allows experiments to be run with and without inter-transaction caching of pages on the client.

The “maxclients” option determines the number of clients a server can server at any one time. Servers create internal tables whose size depends on this value.

The “maxthreads” value, determined by the “maxclients” value, should be sufficient, but can be overridden. If a server recovers from a failure without running out of threads, it has enough threads to handle client requests. If numerous distributed transactions are active at the time of a server failure, it is possible, but unlikely, that the server will not be able to recover with the default number of threads.

The “traceflags” option is available only with a server that was compiled with debugging (the -DDEBUG flag). It is useful for programmers who are modifying the Storage Manager source code and testing their changes.

The “dataformat”, “logformat”, and “tempformat” options are as described in Section 5.1.5, **Formatting Volumes**. Servers can mount and use volumes given in these options.

The “maxaddvolumes” option indicates how large the mount table will be. The server reads its configuration files, counts the volumes named in the format options, and creates a mount table large enough to mount this many volumes and “maxaddvolumes” more. This is a strict limit to the number of volumes that the server can mount (at any one time) as long as it is running. The value of “maxaddvolumes” should not be boosted frivolously, because the size of the mount table affects the amount of shared memory required by the server. The default value is 0.

The “wrapcount” option is rarely needed. The server will tell you if you ever need to set this option. It is needed if you add volumes after the server starts (maxaddvolumes > 0), and a volume that you are add was updated by a server running on a log that differs from the current log (or the log was regenerated since the added volume was last mounted.)

5.2.3. Normal Operation of Servers

During normal operation, servers listen for connections and requests from clients and monitor terminal input. Error messages are printed on the servers terminals when interesting events occur, for example, when a deadlock is detected, or a transaction is aborted by a server because of a problem such as insufficient log space.

5.2.3.1. Server Commands

The following commands can be invoked from the standard input to the server: “help”, “shutdown”, “kill”, “crash”, “checkpoint”, “printstats”, “clearstats”, “progress”, “user”, “addvolume”, “rmvolume”, “listvolumes”, “listmount”, “listdistr”, “source”, “redirect”. When the server is compiled with profiling (-DPROFIL, -p), the server accepts the “profil” command. When the server is compiled with debugging (-DDEBUG), the server also accepts the “traceflags” and “tracelevel” commands.

The “help” command provides a list of the commands.

The “shutdown” command instructs the server to abort all active transactions and cleanly shut down. The “kill” command causes the server to halt immediately after displaying the status of mounted volumes. The “crash” command has the same effect as the “kill” command, except that a core dump is produced as well.

The “checkpoint” command causes the server to take a checkpoint immediately. Checkpoints are taken periodically by servers. The default frequency is once every 100 log pages, but this can be changed by an application program (see sm_ChangeCheckpointFrequency( ) in Section 4.11.2, **Administrative Operations**).

The “printstats” command prints general server statistics. The “clearstats” command clears any counters among the statistics.

The “progress” command reverses the value of the “progress” option.

The “user” command reverses the value of an internal flag that determines whether or not the server prints a message when a user (application) error is encountered. (There is no option to control this.)

The “addvolume” command adds a volume to the server’s table of mountable volumes. The “addvolume” command takes a format-option name and a format-option value. For example, to add the data volume 8000, type

```
      addvolume dataformat /path/to/datafile:8000:1:1:300
```

A volume cannot be added if the given format information conflicts with other information in the table.

The "rmvolume" command removes a volume from the server's table of mountable volumes. The command takes a volume identifier. For example, to remove the data volume 8000, type

```
      rmvolume 8000
```

A volume cannot be removed if it is in use.

The "listvolumes" command prints the server's table of mountable volumes.

The "listmount" command prints a list of the volumes that are in some state of use: mounted, being mounted or being dismounted. It also prints the number of free "mount slots", which indicates how many more volumes could be mounted at any one time, given the server's configuration. To allow more volumes to be mounted at once, shut the server down, boost the value of the "maxaddvolumes" option, and restart the server.

The "listdistr" command prints information about prepared distributed transactions. These transactions consume space in the log, and if they are not aborted or committed, eventually the server will fail because it will have run out of log space. See Section 4.3, **Transactions**, Section 4.11.1, **External Two-Phase Commit Functions** for information about distributed transactions.

The "source" command takes one argument, the path name of a file from which to read commands. The server processes these commands, and when it reads the last command in the file, it resumes reading from the terminal. If the path name is missing or is `/dev/tty`, reading resumes from the terminal.

The "redirect" command takes two arguments. The first argument indicates which output stream is to be redirected: messages to the terminal or error messages. The second argument is the path name of a file to which the output is written. When the output is redirected again, the stream is flushed to the given file and the file is closed. To redirect output to the terminal, use `/dev/tty` or omit the path name.

The "profil" command causes the server to dump its profiling information to disk. This command is available only on a server that was compiled with profiling on (-DPROFIL -p). See the manual page for prof(1).

The "traceflags" command may take an integer argument, which may be a hexadecimal number, such as "0xfa3", in which case it sets the server's trace flags word to that value. The command is available only with a server that was compiled with debugging on (-DDEBUG -g). The meanings of the trace flags are found in the server's source code, in `src/include/global_trace.h`. When "traceflags" is used with no argument, it prints the value of the trace flags word.

The "tracelevel" command is available with a server that was compiled with debugging on (-DDEBUG -g). When used with no argument, it prints the trace level for the trace flags that are on. When given an integer argument (1, 2, or 3), it sets the trace level for the trace flags that are on.

5.2.4. Shutting Down the Server

The server can be shut down several ways. One method is to use one of the above-mentioned commands. Another is to run the "shutserver" program, described below, at the end of this section. A third way to shut down a server is to call sm_ShutdownServer( ) in a client program.

A server may also shut itself down because of a fatal error, such as the unexpected death of a disk process or a bug. A fatal error causes the server to report the state of all the mounted volumes, dump core, and exit.

The server allocates a Unix System V shared-memory segment and a semaphore set when it starts. If a server is shut down in a controlled fashion, it removes the segment and semaphore set. These resources are not removed when the server is terminated by `kill -9 <server process>` typed in the shell, by the "kill" or "crash" command given to the server's terminal monitor, or when the server process is killed by a debugger. **If you use any one of these means to terminate a server, you must use ipcrm(1) to remove the resources.** See the manual pages for ipcs(1) and ipcrm(1) for more information. If the segments and semaphore sets are not removed, eventually the operating system will run out of segments, and you will be unable to start a new server.

If a server shuts down without having committed or aborted all its active transactions and flushed all its dirty pages to disk, recovery is required when the server is restarted. When a server shuts down, it prints the status of all the mounted volumes. It indicates if recovery is necessary on those volumes.

5.2.4.1. Running the Shutserver program

The `shutserver` program is invoked:

```
      shutserver [-m machine] [-s servername] [-h].
```

The "machine" specifies the name of the machine on which runs the server to be shut down. If "-m machine" is not given, the program uses the machine on which `shutserver` is executed. The "servername" is the name of the server in `/etc/services`, If "-s servername" is not given, "exodussm" is used. The "-h" option prints a brief help message.

5.2.5. Recovery

When a server is started after a failure it automatically performs recovery. The time it takes for recovery depends on several factors, including the number of transactions in progress at the time of the failure, the number of log records generated by these transactions, and the number of log records generated since the last checkpoint.

Recovery has three phases. After each phase, the server prints information about the time and I/O operations required to perform the phase.

The first phase is *analysis*. The log is scanned to determine what transactions were active and which volumes were mounted at the time of the failure.

After analysis, the volumes are mounted and the *redo* phase is performed. In the redo phase, data are restored to their state at the time of the failure.

In the last phase, the *undo* phase, the server aborts the transactions that were active at the time of the crash. The volumes are dismounted, and a checkpoint is taken.

For details of recovery in the Storage Manager, see [Fran92].

5.3. Tuning the Server

There are several tuning parameters in the Storage Manager server. The following sections describe each one.

5.3.1.1. The Size of the Buffer Pool

The size of a server's buffer pool is determined by the "bufpages" option, which indicates the number of MIN_PAGESIZE pages in the buffer pool. If a server is the primary process on a machine, it should have a buffer pool close to the size of available shared memory. When both an application and a server are running on the same machine, choosing a buffer pool size is more difficult. A "proper" choice depends on the behavior of the applications and their interactions with servers. A good rule of thumb is that that clients should have the adequate buffer space, to minimize client-server interaction.

The buffer pool must fit in the available shared memory of the machine on which the server runs. The server will let you know if it cannot acquire enough shared memory when it starts. See the manual pages for ipcs(1) and ipcrm(1) to find out how much shared memory is in use. See your system administrator to find out how much shared memory has been configured for your systems if you find that you cannot run a server with a buffer pool of adequate size, and no shared memory segments are being wasted.

5.3.1.2. The Size of Log Pages

The log page size is determined when a log volume is formatted. For a transaction mix dominated by transactions that generate more than a few kilobytes of log information, the larger the log page size, the better. For short running transactions, such as those found in transaction processing benchmarks, 8 Kbyte log pages give good results.

5.3.1.3. Checkpoint Frequency

The checkpoint frequency is based on the number of log pages written. The default frequency is every 100 log pages. The frequency can be determined by setting the "checkpoint" configuration option. It can be changed in a running server by an application that calls sm_ChangeCheckpointFrequency( ). More frequent checkpoints tend to shorten the time required to recover after a server fails at the expense of processing time during normal operation. Checkpoints also cause the server's dirty pages to be flushed to disk, which may also improve performance during normal operation.

6. REFERENCES

[Care86] M. Carey, D. DeWitt, J. Richardson, and E. Shekita, *Object and File Management in the EXODUS Extensible Database System,* **Proc. of the 1986 VLDB Conf.**, Kyoto, Japan, Aug. 1986.

[Care89] M. Carey, D. DeWitt, E. Shekita, *Storage Management for Objects in EXODUS*, **Object-Oriented Concepts, Databases, and Applications**, W. Kim and F. Lochovsky, eds., Addison-Wesley, 1989.

[Chou85] H. Chou and D. Dewitt, *An Evaluation of Buffer Management Strategies for Relational Database Systems*, **Proc. of the 1985 VLDB Conf.**, Stockholm, Sweden, Aug. 1985.

[Fran92] M. Franklin, M. Zwilling, C.K.Tan, M. Carey, and D. DeWitt, *Crash Recovery in Client-Server EXODUS*, **Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data**, San Diego, CA, June 1992.

[Gray78] J. N. Gray, *Notes on Database Operating Systems*, **Lecture Notes in Computer Science 60, Advanced course on Operating Systems**, ed. G. Seegmuller, Springer Verlag, New York 1978.

[Gray88] J. Gray, R. Lorie, G. Putzolu, I. Traiger, *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, **Readings in Database Systems**, ed. M. Stonebraker, Morgan Kaufmann, San Mateo, Ca., 1988.

[Litw88] W. Litwin, *Linear Hashing: A New Tool for File and Table Addressing*, **Readings in Database Systems**, ed. M. Stonebraker, Morgan Kaufmann, San Mateo, Ca., 1988.

[Moha83] C. Mohan, B. Lindsay, *Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions*, **Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing**, Montreal, Canada, August, 1983.

[Moha89] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, ACM Transactions on Database Systems,* Vol. 17, No 1, March 1992.

[Rich87] J. Richardson and M. Carey, *Programming Constructs for Database System Implementation in EXODUS*, **Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data**, San Francisco, CA, May 1987.

[exoArch] *EXODUS Storage Manager Architecture Overview*, unpublished, included in EXODUS Storage Manager software release.

A. APPENDIX : Locking Protocol for Storage Manager Operations

The Storage Manager performs concurrency control using the standard hierarchical two-phase locking protocol (see [Gray78], [Gray88]) for locking files and object pages. The lock hierarchy contains two granularities: file-level, and page-level. Locking for index operations is performed with a non-two-phase protocol, that allows multiple clients to read and update the same index. This section describes the lock modes used in the system, lists the locks requested for each Storage Manager file and object operation, and explains how deadlocks are handled. Lock acquisition and release are *implicit* in all relevant operations, so clients cannot explicitly manage their own locks.

A.1. Lock Modes

Files are locked in one of six modes: no lock (NL), shared (S), exclusive (X), intent to share (IS), intent to exclusive (IX), share with intent to exclusive (SIX) [Gray78], [Gray88]. Only shared and exclusive locks are obtained on pages. Determining whether two locks are compatible (eg., when a client holds a lock on a file and another client wants to obtain a lock on it as well) can be done using a table. Table A.1 is a lock compatibility table for the six file lock modes. Each row indicates a lock that some client can hold, and each column indicates a lock desired by another client. The Y and N table entries indicate (yes or no) whether the locks are compatible or not.

Another table can be used to express lock convertibility. A lock conversion occurs when a client holds a lock in some mode and requests an operation that requires a different mode for the lock. Table A.2 is a lock convertibility table for the six file lock modes. Each row indicates a lock that the client already holds and each column indicates the new lock mode requested. The entries represent the resulting lock mode obtained.

Lock	Lock Requested					
Held	NL	IS	IX	S	SIX	X
NL	Y	Y	Y	Y	Y	Y
IS	Y	Y	Y	Y	Y	N
IX	Y	Y	Y	N	N	N
S	Y	Y	N	Y	N	N
SIX	Y	Y	N	N	N	N
X	Y	N	N	N	N	N

Table A.1: Lock Compatibility

Lock	Lock Requested					
Held	NL	IS	IX	S	SIX	X
NL	NL	IS	IX	S	SIX	X
IS	IS	IS	IX	S	SIX	X
IX	IX	IX	IX	SIX	SIX	X
S	S	S	SIX	S	SIX	X
SIX	SIX	SIX	SIX	SIX	SIX	X
X	X	X	X	X	X	X

Table A.2: Lock Convertibility

A.2. Locks Obtained by Operations

The locks mentioned above are obtained on two types of structures in the Storage Manager: files and pages. Only the pages that contain object headers and root entries are locked; large object data pages and file index pages are not locked. The entire root entry page is locked when a root entry is used.

Table A.3 lists all of the locks obtained by the various Storage Manager operations. The column labelled “File Lock” indicates what lock mode is used for locking the file in question. The column labelled “Page Lock” indicates what lock mode is used for locking pages containing the objects or root entries in question. Locks are held until the end of the transaction in which they were acquired.

Some applications may find it necessary to acquire more restrictive locks on pages and files to avoid conflicts during lock-upgrade requests. For example, consider an application that reads an object (with sm_ReadObject( )) and subsequently writes it (with sm_WriteObject( )). When the object is read, a share lock is acquired for the object’s page. When the object is written, a lock-upgrade request is sent to the server to obtain an exclusive lock on the page. This extra message is relatively expensive and can lead to potential deadlock if other clients are locking the page as well. To avoid this problem, the “pagelock” option can be used to change the default lock modes used when the client library locks a page. See Table 1 and the discussion of client options in Section 4.2, **Initialization and Shutdown Operations** for information about setting client options. See Appendix A for more information about lock modes and the Storage Manager’s locking protocols.

A.3. Deadlock Detection and Avoidance

With each lock request, a server analyzes its local waits-for graph and detects local cycles, or “local deadlocks”. The request that would cause a deadlock is denied (returns esmFAILURE),

Operation	File Lock	Page Lock	Comments
sm_Initialize( )	-	-	no locks needed
sm_ShutDown( )	-	-	no locks needed
sm_OpenBufferGroup( )	-	-	no locks needed
sm_CloseBufferGroup( )	-	-	no locks needed
sm_SetRootEntry( )	-	X	root entry page
sm_GetRootEntry( )	-	S	root entry page
sm_RemoveRootEntry( )	-	X	root entry page
sm_CreateFile( )	X	-	
sm_DestroyFile( )	X	-	
sm_GetFirstOid( )	S	-	
sm_GetLastOid( )	S	-	
sm_GetNextOid( )	S	-	
sm_GetPreviousOid( )	S	-	
sm_OpenScan( )	S	-	
sm_OpenScanWithGroup( )	S	-	
sm_ScanNextObject( )	-	-	no locks needed
sm_CloseScan( )	-	-	no locks needed
sm_OpenLoad( )	X	-	
sm_LoadNextObject( )	-	-	no locks needed
sm_CloseLoad( )	-	-	no locks needed
sm_CreateObject( )	IX	X	unordered file
sm_DestroyObject( )	IX	X	
sm_ReadObject( )	IS	S	
sm_ReadObjectHeader( )	IS	S	
sm_ReleaseObject( )	-	-	no locks needed
sm_WriteObject( )	IX	X	
sm_InsertInObject( )	IX	X	
sm_AppendToObject( )	IX	X	
sm_DeleteFromObject( )	IX	X	
sm_CreateVersion( )	IX	X	
sm_FreezeVersion( )	IX	X	

Table A.3: Locks Obtained by Operations

and the client library returns esmLOCKCAUSEDDEADLOCK to the application in the global variable sm_errno.

Distributed transactions may also cause a deadlock. The servers do not detect deadlocks that involve other servers. Global deadlocks are avoided by timing out locks. Each request that awaits a lock is aged. When its age exceeds the time given by the client’s “locktimeout” option, the request is denied (returns esmFAILURE), and the client library returns esmLOCKBUSY to the application in the global variable sm_errno.

When an application’s request fails with esmLOCKBUSY or esmLOCKCAUSEDDEADLOCK, the application must abort its transaction, to free the locks it holds, and it must start its transaction again.

B. APPENDIX : Generation of Unique Numbers for OIDs

The “unique” field of an OID is special 32-bit value that is generated when the object is created and used to detect instances where the OID has become dangling or corrupted. The values that are stored in “unique” fields are generated by Storage Manager servers. Disk volumes are partitioned into blocks of 32 pages, and for each partition a 32-bit counter is maintained. When a new page is allocated, it is allotted a range (100) of unique numbers to use during object creation. The counter in the partition containing the new page is incremented to reflect the allotment. When this allotment has been exhausted, a request is made to the server for another allotment. When an object is created in a particular partition, the “unique” field of the new object’s OID is set to the next available number in the range on the page. While this strategy does not guarantee that OIDs are unique for all time, the probability of a dangling OID that maps to the same page and the same slot, and has the same “unique” field as a valid OID is very low. As a result, “unique” fields can be used virtually to guarantee the validity of an OID. We adopted this approach instead of using unique-for-all-time logical OIDs with a surrogate index in order to avoid the extra disk I/Os that might be needed to translate a logical OID to a physical address.

TABLE OF CONTENTS