Storage Manager Architecture

June 29, 1999

This document was supported by DARPA through Rome Research Laboratory Contract No. F30602-97-2-0247.

1 Preface

This document describes the storage manager built for the SHORE (Scalable Heterogeneous Object Repository) project. The SHORE Project ran from 1992 through 1996, and among its goals was to build a typed, persistent object base with language heterogeneity and Unix compatibility. SHORE's type system, Unix compatibility, and language support are provided in layers above the storage manager. The storage manager provides persistence of untyped data, concurrency control and recovery, and other conventional storage manager functions.

Several research projects have been using this storage manager:

- Paradise at http://www.cs.wisc.edu/paradise
- Predator at http://simon.cs.cornell.edu/Info/Projects/Predator
- DimSum at http://www.cs.umd.edu/projects/dimsum/)

In order that researchers can use and extend the storage manager, this document is meant to describe some of the history, rationale and conventions of the code. It is not a description of the programming interface. It is meant to be a companion to the source code.

1.1 Historical background

The SHORE project software was intended to be a tool for research. Some researchers used the storage manager in toto, using it as a base for their prototypical software. Others modified the storage manager. In particular, two graduate students' theses were related to storage manager functions. For this reason, it was expected that some parts of the storage manager would be replaced as the researchers' work matured. Two functional areas in particular were written by graduate students whose work was never installed into the public storage manager releases: cooperating distributed servers (so-called multi-server capabilities) and disk space management. Near the end of the SHORE project, the sole in-house user of the storage manager was Paradise, which did its own data distribution above the storage manager layer, so integrating the multi-server functionality was not a high priority. Likewise, the space management work was completed about the time the SHORE project ended, hence that work was never installed. As a result, the space management module of the storage manager is correct, but not particularly efficient in space or time.

Gray and Reuter:[1] is a good place to start reading about some of the basics of storage management; much of the basic design of the storage manager comes from ideas in this book.

Another resource is the Mohan papers[2] about ARIES. The storage manager implements ARIES recovery and the B+-tree implementation is straight from Mohan's ARIES/KVL[3] and ARIES/IM[4] papers.

Finally, much of the source code predates the C++ standard and standard template library (STL), so there is some duplication of functionality between the libraries in the storage manager and the standard template library. Similarly, C++ exception handling is not used in the storage manager because it was non-existent when much of this code was written.

2 Organization

The storage manager is a set of libraries. The software was designed for clean separation of functionality into modules with the hope of maximizing unit-testing and software reuse. The organization of the modules into files and directories is, to some extent, an historical artifact of SHORE, but, in general, it reflects a desire to minimize physical (compile- and link-time) dependencies among files, libraries, and unit tests.

The programming language is C++, and as much as possible, each class (or suite of classes) is tested in isolation. Many directories in the source tree contain a subdirectory called `tests/`, which contains unit-tests for the classes defined in the directory above it.

2.1 Configuring and building

There are compile-time configuration constants (e.g., page size) that control the configuration of the entire package. The directory

- `config/`

contains templates for generated Makefiles and it also contains a template from which a file called `shore.def` is made. `Shore.def` is where compile-time constants are defined and where build-time configuration options are set. For example, to configure the storage manager so that it uses a 32KB page size, you must do as follows:

```
cd config/
cp shore.def.example shore.def
vi shore.def
:/SM_PAGESIZE
:s/8192/32768/
```

The convention is to put a comment in `shore.def.example` for every configurable option, along with a comment about that option's purpose. Not all options described there are supported anymore. Not all code enclosed in C preprocessor macro controls are described in `shore.def.example`, either, since some C preprocessor macros are defined by the Makefiles. The most obvious examples are the architecture-dependent macros, such as `SOLARIS2` and `Sparc`.

The software can be built on Solaris 2.5 and 2.6 using GNU Make and g++, imake, and Perl.

It can also be built on NT 4 using a suite of GNU tools, Perl, and the native VC++ 5.0 compiler, or using the Developer Studio environment. The software does not come with a set of Visual Studio project files.

The source directories have some circular dependencies.

2.2 Conventions

2.2.1 Conventions used in this document

A convention used in this document is that `courier` font is used wherever references are made to file names, keyboard input, and code in the storage manager.

Descriptions of classes in this document are not complete; often constructors, destructors, and operators are omitted from the descriptions here. This document is meant to be a companion to the source code.

2.2.2 Coding conventions

Early in the life of the SHORE project, the storage manager used these conventions in the code, but the conventions were not strictly held:

- Types end in `_t`, as in `smsize_t`, unless they are persistent, in which case they end in `_s`, as in `sinfo_s`.
- Manager classes (of which there is generally a single instance) end in `_m`, as in `lock_m` (the lock manager).
- The names of public methods and data members do not start with an underscore, while the names o f private ones do.
- Because namespaces were not part of C++ at the time, classes and enumerations were used to limit global namespace pollution.

- The storage manager makes heavy use of named types in place of primitive types like `int, short.` Instead, `smsize_t, slotid_t`, and the like are used.
- Iterator classes end in `_i`, as in `scan_index_i`, an iterator for scanning an index.

2.2.3 File naming conventions

Input for GNU Make are called `Makefile`, and are generated from `Imakefile`.
One should never edit a `Makefile`. At the top of the source tree is a special file, `GNUMakefile`, which is not generated, and can be edited. It starts the process of building the make files in the subordinate directories.

Source files included by other source files have `.h` suffixes, and all C++ sources are in files with `.cpp` suffixes. This suffix was chosen so that the NT developer studio can identify C++ files.

Generated source files end in `_gen` and a standard prefix. For example, some code generated for logging resides in files `logdef_gen.cpp` and `logfunc_gen.h.`

Input files to Perl scripts, from which code is generated, have `.dat` suffixes.

Perl scripts have `.pl` suffixes.

Include files that describe persistent data structures end in `_s.h`, as they contain descriptions of types that end in `_s.`

2.2.4 Configuration conventions

C preprocessor macros control both the build environment through `Makefile`s and the compilation environment through the use of `-Dmacro` and the `#define`d macros in the file `shore.def.`

2.2.5 Working around compiler bugs

The file `fc/w_workaround.h` contains C preprocessor macros that both identify and, when possible, work around compiler bugs. Comments near the macro definitions describe the bug. It is not always possible to work around a bug with a simple macro; in such cases, comments in the code should identify the nature of the bug and the work-around.

2.2.6 Unit-testing

Most directories have a set of unit tests. These are short programs that test one or more classes defined in the containing directory. These tests can be run by invoking the shell script `all`.
The unit tests are not built my the make files in the containing directory. One must descend to the test directory and type

```
make
```

to build the tests.

2.2.7 Error codes

Each directory has one or more sets of error codes. An error code is an unsigned integer value that is associated with an error message (character string). A set of error codes is a group of values that share a bit mask. Sets are generated by a Perl script from a `.dat` file. Error codes are described in more detail below.

2.3 Generated sources

Some of the sources in the storage manager are generated from Perl scripts. Most of the Perl code that is used resides in various `.pl` files in the directory

- `tools/`

This directory also contains some scripts that are run on an occasional basis, such as those for updating copyrights.

2.4 Source directory hierarchy

The directory

- `src/`

contains sources for four libraries, as follows:

- Common types: `fc/`
- SHORE threads: `sthread/`
- More common types: `common/`
- Storage manager: `sm/`

3 Common types - fc/

This library, called libfc.a, contains template classes for linked lists, hash tables, for class-specific memory allocation, reference-counted pointers, vectors of arithmetic values, and miscellaneous low-level functions. Also included are support for extensible (generated) sets of statistics, and OS-specific statistics. Each layer in the source hierarchy keeps its own statistics, but uses functions provided here.

Nothing in this directory is aware of threads, so none of the code here is thread-safe.

The include files in this directory are exported for client use. That is, they are copied to `installed/include` with the make target "make install".

Virtual tables

This library also contains some classes whose purpose is to present to servers certain low-level information in such a way that servers can easily turn the information into "virtual tables. There are some circular dependencies among the source directories in order to support virtual tables, but the build mechanism takes care of that, as described above.

The premise is that a server will collect miscellaneous run-time information from the storage manager and put that information in to a table (relation). The information gathered is from various and sundry classes in the storage manager, some of them hidden deep beneath the server-storage manager API. To allow the server access to the (sometimes private) data for these disparate classes, we provide a simple canonical form (`vtable_info_array_t`) to which each class converts its data for the server to receive. For example, when the server collects up-to-date transaction data from the storage manager it calls a storage manager method (`ss_m::xct_collect(vtable_info_array_t &t)`, which in turn, visits each transaction instance and puts all the transaction data into a 2-dimensional array of values (a `vtable_info_array_t), t.` Each `t[i]` is a `vtable_info_t,` which represents a tuple or row in the table. `T[i][j]` is a value in the column `j` of the table, and the domain of `j` is an enumeration defined in `vtable_enum.h` Every value in the table is represented by a string; it is already in string form.

`Files: vtable_info.h, vtable_info.cpp, vtable_enum.h,` files generated by `tools/stat.pl: *collect_gen.h`

Error logging

Here you will find a set of classes, `ErrorLog` and `_debug`, which provide low-level debugging (tracing). This is for logging to an OS file, not to a database storage structure or to a recovery log. These classes are meant to provide a service like the Unix syslog service with C++ ostream syntax. For example, the storage manager creates an instance of an error log, which it uses as follows:

```
smlevel_0::errlog->clog << error_prio
        << "ss_m cannot be instantiated more than once."
        << flushl;
```

The message will be sent to the log only if the storage manager's runtime configuration option

```
sm_errlog_level
```

is higher than or equal to the level of the message. The levels are represented by the enumeration

```
enum LogPriority {
        log_none, log_emerg, log_fatal, log_alert, log_internal,
        log_error, log_warning, log_info, log_debug, log_all
};
```

The I/O manipulators `error_prio, warning_prio, info_prio`, etc., correspond to the log priorities. The storage manager uses only these three I/O manipulators in its messages.

A special log,

```
class __debug : public ErrLog { ... }
```

is used when the storage manager is built with `-DDEBUG` (which is turned on by setting `#define DEBUGCODE ON` in the configuration file `shore.def`). The macros

`DBG()` and DBGTHRD()

send their output to this class along with the line number and file name of the source file in which the macro appears. The class prints to a file only if the class's source file list contains the name of the file in which the macro is used. This allows tracing to turned on and off on a per-source file basis. The destination for tracing messages is determined by

```
setenv DEBUG_FILE <filename>
```

and the source file list is determined by the value of the environment variable `DEBUG_FLAGS`, for example:

```
setenv DEBUG_FLAGS "sm.cpp vol.h"
```

```
Files: errlog.h, errlog.cpp, errlog_s.h, debug.h, debug.cpp
```

Error codes and return codes

Errors are reported from functions by returning an instance of class in this library, called `w_rc_t`, with the alias `rc_t`. This class supports stack traces and integer-to-string mappings for library-specific error sets. The size of the stack trace is a compile-time option (`max_trace` enumeration constant in `w_error.h`

An instance `w_rc_t` is the size of a pointer. It is, in fact, a reference-counted pointer to an instance of `w_error_t`. The `w_error_t` keeps the stack trace and error code. Operations on `w_rc_t` are translated to operations on the underlying `w_error_t.` If a function returns an instance of a `w_rc_t`, and that return value is never *checked*, <u>the</u> <u>destructor for the return value prints an error message:</u> "Error not checked". This is mean to encourage complete error-checking on the part of programmers.

```
Files: w_rc.h, w_rc.cpp, w_error.h, w_error.cpp, w_sptr.h
```

Dynamic memory allocation

The classes

- `auto_delete_t`, and
- `auto_delete_array_t`

are used to clean up heap-allocated objects when leaving a scope, similar to the `auto_ptr` class template in the C++ standard template library.

```
Files: w_autodel.h
```

Factories

Throughout the storage manager, there are examples of classes of fixed size that are instantiated often. A set of template classes allows the storage manager to create *factories* that allocate a set of instances to dole out at a later time. The template `w_fastnew_t` does this, with the intent of avoiding checkerboarding memory in the standard library malloc/free space. The template class `w_threadnew_t` further allows the allocated space to be accounted for, on a per-thread basis.

Both of these classes are derived from a common base class, `w_factory_t`, which can be configured to do some detailed accounting of the allocated memory. When compiled with

```
#define INSTRUMENT_MEM_ALLOC,
```

this class calls two callback functions to account for the space allocated, and it links together all allocated memory. The metadata in the memory header contains a "heap identifier", of the form

```
w_heapid_t
```

which is assigned and used by the callback functions.

This library defines the callback functions

```
extern w_heapid_t w_shore_thread_alloc(
      size_t bytes,
      bool is_free);
extern void w_shore_thread_dealloc(
      size_t bytes,
      w_heapid_t h);
```

The callback functions defined in this library account for all allocations as allocated from a global heap,, which is to say that it keeps a single counter for all allocations; `w_shore_thread_alloc()` always returns the heap id

```
w_heapid_t  w_no_shore_thread_id,
```

and `w_shore_thread_dealloc()` ignores its heap identifier argument.

The threads library contains a different implementation of the callback functions, which account for the allocation and freeing of space in the running thread, and use the identifier of the running thread for the heap identifier.

These classes contain the necessary methods to collect the memory-allocation statistics into a virtual table.

When building a storage manager on NT with Visual Studio, care must be taken not to link in the `fc/` *version of the callback functions.*

```
Files: w_factory.h, w_fctory.cpp, w_factory_fast.h, w_factory_fast.cpp,
w_factory_thread.h, w_factory_thread.cpp, w_shore_alloc.h,
w_shore_alloc.cpp
```

OS statistics

The class `unix_stats` can be used to start and stop the collection of the resource usage statistics available on different Unix flavors. An example of its use is in `fc/tests/stats.cpp.`

```
Files: unix_stats.h, unix_stats.cpp
```

Bitmaps

```
Files: w_bitmap.h, w_bitmap.cpp, w_bitmap_space.h
```

Lists

The storage manager makes heavy use of doubly-linked lists, provided by
```
template <class T> class w_list_t
```
and its companion iterator
```
template <class T> class w_list_i.
```
To use these lists, a structure or class must contain a
```
w_link_t
```
for each list that it wants to be put on at any one time. For example,
```
struct foo {
      int         bar[10];
      short       barnone;
      w_link_t    link;
};
```
The list keeps track of the offset of the link that it will use, so that offset is given when the list is constructed:
```
typedef w_list_t<foo>   foolist;
foolist FL(offsetof(foo, link));
foo* F = new foo;
...
FL.push(F); // put F on the front of the list
            // OR
FL.append(F); // put F at the end of the list
```

To iterate over the list:
```
w_list_i<foo> i(FL);
while (F = i.next())) {
       // use F
}
```
To remove items from the list:
```
F =FL.pop();
```

Lists can be ordered, as implemented by the subclassed templates
```
template <class T, class KEY> class w_ascend_list_t;
template <class T, class KEY> class w_descend_list_t;
```

Such a list is constructed by giving the offset of a key as well as the offset of the link:
```
w_descend_list_t<foo,short) list(
                  offsetof(foo, barnone),
                  offsetof(foo, link));
```

The test program
```
tests/list3.cpp
```
demonstrates the use of these lists.

Files: w_list.h, w_listm.h

Queues

Circular queues are implemented by the classes
```
template <class T> class w_cirqueue_t;
```

which, unlike the lists, does not link its contents; rather it maintains an array of pointers. The size of the array is given when the queue is constructed. The iterators for a queue are
```
template <class T> class w_cirqueue_i;
template <class T> class w_cirqueue_reverse_i;
```
These classes are demonstrated by the test programs
`tests/cq1.cpp` and `tests/cq2.cpp`.

Files: w_cirqueue.h

Timers

```
Files: w_timer.h, w_timer.cpp
```

Shared memory

The Unix implementation of the storage manager uses shared memory for communication among processes. The system calls for allocating and attaching shared memory are encapsulated in a class `w_shmem_t`.

```
Files: w_shmem.h, w_shmem.cpp
```

Pseudo-random numbers

Some of the test programs (`ioperf` and `ssh`) use pseudo-random numbers. The class `random_generator` encapsulates the platform-specific differences in the random-number library functions.

```
Files: w_random.h, w_random.cpp
```

Heaps

```
Files: w_heap.h
```

4 SHORE threads - sthread

The storage manager uses a home-grown threads package that is derived from NewThreads (from the University of Washington). The API is a portable OS interface. The threads are dynamically allocatable, and their maximum stack sizes may be determined at the time they are allocated. The library contains a main thread, an idle thread, and a prioritized ready queue for all user-defined threads. All scheduling is non-preemptive. This library runs on NT, but it is not a mapping to native NT threads.

This layer has been ported to the following Platforms:

- Solaris 2.5/Sparc, x86
- SunOS/Sparc
- AIX4.1/RS6000
- Windows NT/x86

4.1 Non-blocking I/O

The first implementation ran on several variants of Unix, providing non-blocking I/O by forking off processes that communicate via shared memory. Under NT, the cooperating processes are NT threads within the server process. For each disk, a separate "`diskrw`" (disk read-write) process (or NT thread) is forked by the storage manager process. Under Unix, there is also a master `diskrw` process that cleans up after catastrophic failures. (It does not do recovery or restart -- it just cleans up shared memory.) On Unix, the `diskrw` process communicates with the server process using a shared-memory queue and a pipe. The pipe is not used to convey information other than to “kick” the server into checking the shared-memory queues. Process synchronization on the shared-memory queue is accomplished with a test-and-set instruction, if the architecture has one, else with spin locks (at most two processes are vying for any one critical section). On NT, since the `diskrw` and server processes share address spaces, there is no need for the queues and pipes, but because the port to NT is a “shallow”, i.e., minimal-effort, port, there is a pipe between threads, implemented over TCP, which allows the threads layer to work, albeit with unnecessary overhead.

```
Files: diskrw.h, diskrw.cpp, sthread.h, sthread.cpp, spin.h, tsl.h,
tsl.S
```

4.2 Thread synchronization

The threads package provides synchronization services that are used throughout the storage manager:

- `smutex_t` a mutual exclusion variable (lock, mutex variable, binary semaphore) whose methods are `acquire(timeout)`*(wait, P)* and `release()` *(signal, V)*. Only a holder of (a thread that has acquired) a mutex can release it. A mutex has at most one holder at any time.
- `scond_t` a condition variable (a list of waiters, protected by a mutex of the caller's choice), whose methods are `wait(smutext_t, timeout), signal(), broadcast()`, and `bool is_hot()`. Like condition variables described by Silberschatz[5],[6], signaling a `scond_t` when no threads are waiting is a null operation. Any thread can signal a condition variable.

The implementation assumes a uniprocessor environment, so waiting on a mutex or a condition variable always means being put on a list; spinning is not used.

```
Files: sthread.h, sthread.cpp
```

4.3 Statistics

The library keeps some statistics that help debugging and performance analysis. These statistics are not thread-safe, hence, they are approximate. The statistics are kept in a global variable

```
extern sthread_stats SthreadStats;
```

The statistics kept include the number of

- I/O system calls
- select() system calls (on Unix)
- times a thread awaited a latch
- times a thread awaited a condition variable
- times a thread awaited a mutex

An example of its use is in

```
tests/ioperf.cpp
```

```
Files: sthread_stats.dat, sthread_stats.cpp
```

4.4 I/O performance

The test directory contains several unit tests and one very useful program, `ioperf.`
Ioperf is an example of a program that uses the threads for non-blocking I/O in a minimal way, so that it is easy to measure the performance of I/O through the threads package. By default, it performs sequential I/O. If its name is `randomioperf` (by way of a hard link or a rename), it performs pseudo-random I/O instead.

```
Files: tests/ioperf.cpp
```

5 Common Types - common/

This library (`libCOMMON.a`), contains some types that use types defined by the threads library.

It was originally meant to contain types that are shared with client-side code (clients of value-added servers). It also contains some things that are here only for historical reasons, and could just as well be in `libFC.a`.

Hash table

The logical-ID service of the storage manager uses a fixed-sized, thread-safe hash table with LRU replacement.

```
Files: hash_lru.h, hash_lru.cpp
```

Transaction IDs

```
Files: tid_t.h, tid_t.cpp
```

Data vectors

```
Files: vec_t.h, vec_t.cpp, vec_mkchunk.cpp, zvec_t.h
```

Basic types

```
Files: basics.h, basics.cpp, devid_t.h, devid_t.cpp, vid_t.h, vid_t.cpp,
stid_t.h, stid_t.cpp
```

Options

```
Files: opt_error.dat, option.h, option.cpp
```

Latches

Besides mutexes and condition variables, the storage manager uses *latches* to synchronize multi-thread access to memory. The latch implementation is located in `common/`, although it will probably be moved to `sthread/.`

A latch is, like a database lock, can have more than one holder, and can, held in *shared* or *exclusive mode.* Like a mutex, a latch does not perform deadlock detection when a thread waits on it. Latches are implemented by the

```
class latch_t {
public:
      latch_t(const char *const name=0); // latches
                                          // are named
      ~latch_t();

      w_rc_t      acquire(latch_mode_t m, timeout = WAIT_FOREVER);
      void  release();
      w_rc_t      upgrade_if_not_block(bool& would_block);

      bool  is_locked() const;
      int         lock_cnt() const;
      latch_mode_t            mode() const;


      bool  is_hot() const;
      int         num_holders() const;
      bool  is_mine() const;
      const sthread_t * holder() const;
};
```

At most 4 threads can hold a single latch (in shared mode, of course) at any one time. This limit is a compile-time constant.

```
Files: latch.h, latch.cpp
```

The class `auto_release_t` provides for automatic releasing of a mutex when leaving scope.

```
Files: auto_release.h
```

Key-value locks

```
Files: kvl_t.h, kvl_t.cpp
```

Regular expressions

Because the options-processing code uses regular expressions, a copy of freely-available GNU regular expression code is included here. The code has been modified only to suit the storage manager build environment.

```
Files: regex.h, regex.posix.h, regex2.h, regex_cclass.h, regex_cname.h,
regex_engine_i.h, regex_utils.h, regcomp_i.h regerror_i.h, regcomp.cpp,
regerror.cpp, regex.posix.cpp, regex_engine.cpp, regexec.cpp,
regfree.cpp
```

Logical IDs

The logical IDs are:

- volume ID: implemented by `lvid_t`
- record ID implemented by `class lrid_t`, which contains an `lvid_t` and a *serial number*, which is implemented in `serial_t.`

Serial numbers are 32- or 64-bit (determined by a compile-time option) quantities that identify records on a volume. A logical record ID is unique. Each volume that is formatted for use with logical IDs contains two indexes. One maps serial numbers to physical record IDs, and the other is the reverse mapping. Using logical IDs accomplishes two things, at some expense:

1. Logical IDs are smaller than physical IDs, so some space can be saved for databases that include many intra-database references.
2. Databases using logical IDs can be reorganized without requiring a complete dump and reload operation.

The expense of using logical IDs comes from

1. the space overhead for the mappings, and
2. the time overhead for maintaining the mappings.

```
Files: lid_t.h, lid_t.cpp, serial_t.h, serial_t.cpp, serial_t_data.h
```

6 Communication package (missing)

Under the aegis of the Paradise project, a communications library was written to support low-level inter-process, inter-machine communications. The source code for the communications library is not available to the public, but it is mentioned here, because the storage manager's support for distributed transactions was built on this package. Lest a reader consider inserting his own communications library, what follows is a brief description of the communications package that was used.

The communications library provides a communications model based on Endpoints, which are similar to Mach ports. Messages can be passed among threads and among processes, with low overhead in space and time. Endpoints can be shipped along with messages. Delivery is reliable and ordered between a (source) thread and a (destination) endpoint. The API contains "death notifications" - a registration mechanism that gives reliable failure detection.

The communication service assumed by the storage manager is as follows:

`class CommSystem;`
: a handle on the basic communications subsystem

`class NameService;`
: a handle on a service for registering endpoints by name and for looking them up

`class Buffer;`
: a place in which to put messages to be sent, and from which to extract those that are received

`class Endpoint;`
: an entity that sends and receives messages. Endpoints are not connected to form streams, rather, each message is sent to an endpoint as if it were anonymously sent --without any notion of a source endpoint

`class EndpointBox;`
: a collection of Endpoints that can be sent along with messages.

The portions of these classes that are used by the storage manager are:

```
class Buffer {
public:
      static Buffer& null;

      Buffer();
      Buffer(int length);
      Buffer(void *addr, int length);
      ~Buffer();

      Buffer      &operator=(const Buffer &from);// copy

      int   mlength() const; // returns size() (msg length)
};

class EndpointBox {
public:
      EndpointBox(unsigned box_size = default_size);
      EndpointBox(const EndpointBox &box);
      ~EndpointBox();

      EndpointBox &operator=(const EndpointBox &box);

      // Box is an array with origin 0
      w_rc_t      set(unsigned n, const Endpoint &it);
          // put “it” in the “n”th slot in the box
      w_rc_t      get(unsigned n, Endpoint &it);
          // get an endpoint from the “n”th slot
          // and return it in “it”

      unsigned    count() const; // # endpoints in box
};

class NameService {
public:
      static      w_rc_t      startup(CommSystem &, istream &,
                                    NameService *&);
```

```
      ~NameService();

      // register a name
      w_rc_t      enter(const char *name, Endpoint ep);

      // create an endpoint associated with a name
      w_rc_t      lookup(const char *name, Endpoint &ep);

      // un-register a name
      w_rc_t      cancel(const char *name);// remove

      // don't need any more name service
      w_rc_t      shutdown();
};

class CommSystem {
public:
      // locate the CommSystem, return a ptr to it
      static      w_rc_t      startup(CommSystem *&me);
      w_rc_t            shutdown();

      // Create anonymous endpoints
      w_rc_t      make_endpoint(Endpoint &it);

      // ascii representations of endpoints:
      w_rc_t      make_endpoint(istream &spec_text,
                        Endpoint &it);
      w_rc_t      get_spec(Endpoint ep, ostream &spec_text);
      ostream     &print(ostream &) const;
};
class Endpoint {
public:
      Endpoint();
      ~Endpoint();

      // send msg TO this endpoint, and enclose the
      // box of endpoints with the message
      w_rc_t      send(Buffer &msg,
                  EndpointBox &box = emptyBox) const;

      // receive a message bound for this endpoint,
      // and collect the endpoints enclosed in the box
      w_rc_t      receive(Buffer &msg,
                  EndpointBox &box) const;

      // endpoints are reference-counted
      w_rc_t      acquire() const;// increment ref count
      w_rc_t      release() const;// decrement ref count
      w_rc_t      destroy();//clear ref count

      // If this endpoint "dies" (we can't send to it
      // anymore), send the message in "buffer" to
      // the endpoint "target"
      w_rc_t      notify(Endpoint &target, Buffer &buffer);

      // undo the above action -- no longer interested
      // in death notification
```

```
    w_rc_t      stop_notify(Endpoint &target);

    bool  is_valid() const
    bool  operator==(const Endpoint &it) const;
    bool  operator!=(const Endpoint &it) const;

    ostream     &print(ostream &s) const;

};
```

7 Storage Manager - sm/

The storage manager library (`libSM.a`) is the largest library in the group. Because it encompasses a lot of functionality, we break it into a set of smaller, somewhat inter-dependent modules. There is no direct correspondence between modules (identified below) and source files, although the names of the source files should be somewhat indicative of their content.

Before we delve into the various parts of the storage manager, a few words are in order concerning the storage manager's use of threads, interactions with clients and other servers, and class instantiations.

7.1 Threads

The storage manager is a library. The server (code that uses the library) is responsible for its interactions with clients and cooperating servers. For the server to accomplish this without blocking the entire process, the server must use the threads library that the storage manager uses.

It is assumed that the server will run as a collection of threads (of the SHORE threads variety).
Some of those threads will be invoking the storage manage; others might not. For a thread to use any storage manager functions, it must be a special thread: one derived from the class

```
    smthread_t
```

which is derived from `sthread_t`, the basic SHORE thread. Before any storage manager services can be used, the server will create a single instance (one instance in the address space of the process) of the storage manager:

```
    sm = new ss_m();
```

At this time, the storage manager performs recovery and creates instances of the various manager classes that compose the storage manager: it creates a lock manager, a buffer pool manager, etc. Some of these managers create instances of `smthread_t` threads that run in the background. Thereafter, all storage manager activity occurs in server threads (presumably acting on behalf of clients).

Synchronization among threads is accomplished with `smutex_t`, `s_cond_t` (from the threads package) and `latch_t` (built using `smutex_t`) classes.

```
Files: smthread.h, smthread.cpp, smthreadstats.cpp
```

7.2 Identifiers

The storage manager supports both logical and physical identifiers. This document describes only the physical identifiers. Each class in the API has pairs of methods -- one method that takes logical identifiers and one that takes physical identifiers. The logical ID methods can be understood from the descriptions of physical-ID methods in this document.

7.2.1 Physical IDs

The physical IDs that are used throughout the storage manager are:

- volume ID: given to a volume when the volume is formatted. Implemented by the `class vid_t.`
- store ID (file ID, index ID): identifies all storage structures. Implemented by the `class stid_t`, which contains a `vid_t` and a store number, `snum_t.`
- page ID: implemented by the `class lpid_t,` which contains a `stid_t` and a page number, `shpid_t.`
- record ID: implemented by the `class rid_t`, which contains a `lpid_t` and a `slotid_t.`

7.2.2 Logical IDs (lid_m)

The data types for logical IDs are defined in the common library and include files (q.v.).

7.3 Disk space management (vol_m, io_m)

The “volume manager” manages space on a volume. The storage manager API permits extensible volumes, but the implementation is limited to one volume per operating system disk partition or file.

The API:

- A volume is a set of pages on a single “logical device”, which can contain any number of volumes, depending on how it is formatted.
- A “logical device” consists of one or more physical devices.
- A physical device is a Unix raw partition or an OS file.

The IMPLEMENTATION:

- at most one volume per logical device.
- a logical device == one physical device.

All pages are the same size (a compile-time constant). This is an assumption that pervades the storage manager.

The storage manager modules use different types of pages. Each module derives its own private page types from a common base class. The common base class manages the common page header and the slot table for the page. Thus, every page is a slotted page, even though some managers use only one slot. (This gives rise to some confusing terminology in the storage manager: "slotted page" usually refers to file pages only.) The volume manager allocates and deallocates pages without regard to their type. The volume manager is responsible only for a few derived page types: those that contain only metadata for managing the allocation and
deallocation of pages.

Together, each page type and the common base class for pages account for each byte on the page. The accounting functions distinguish metadata from user data from bytes wasted on alignment This permits a complete accounting of every byte on a volume.

The volume manager arranges pages on a volume into "stores". Pages are allocated to stores in “extents” (groups of contiguous pages). Hence, the unit of *reservation* is an extent, while the unit of *allocation* is a page. There is NO clustering support in the storage manager other than extents. The number of pages in an extent is a compile-time constant; the default is 8.

Regardless of its page type, every reserved or allocated page resides in a store. Each store has a property that determines the way updates to the store are logged.

At any given time, each page is either fully logged (all updates to the page are logged) or not logged (no updates to the page are logged). The allocation metadata are kept on pages that are always fully logged.

Each store has a logging property whose meaning can be expressed in terms of the logging properties of the pages in the store, as follows:

- Regular: a regular store's pages are fully logged The store is marked with a `store_flag_t` value, `st_regular`.
- Temporary: a temporary store's pages are not logged. A temporary store can persist between transactions. It is removed on mount or dismount of its volume. The store flag for a temporary store is `st_tmp.`
- Load: a store created with this property starts out looking like a temporary store, but it is changed to a regular store when its creating transaction commits. This change is enforced by the storage manager. Such a store has a store flag of `st_load_file`.
- Insert: a store cannot be created with this property. A regular store can be changed to an insert store (`st_insert_file`), which allows newly allocated pages to be unlogged until commit, and committed pages to be regular. Only stores that are used for files can be used this way (stores used by indexes cannot).

This permits bulk-loading with minimal logging, followed by fully-logged updates, followed by large, minimally-logged append operations. Changing the logging property of a store does not require updating each page in the store.

A transaction creates and deletes stores when it creates and deletes storage structures. Deletion of stores is postponed until the transaction commits so that the pages allocated to the deleted store remain reserved. lest the deleting transaction aborts. The deleting transaction could (in theory) reallocate such pages, however that is never done. Instead, when a store is deleted by a transaction, the head of the store is marked "for deletion", and the store's identifier is appended to a list held by the transaction. When the transaction ends, the stores on the list are destroyed if the transaction commits, or not it the transaction aborts. Stores that are marked `st_load_file` by a transaction are also put on a list held by the transaction. At commit time, the stores on this list are re-marked so they become `st_regular`. Because these are logged actions, and they occur if and only if the transaction commits, the storage manager guarantees that the ending of the transaction and the re-marking and deletion of stores is atomic. This is accomplished by putting the transaction into a state `xct_freeing_space` (see Transaction manager), and writing a log record that indicates the state change. The space is freed, stores are converted to regular stores as needed, and a final log record is written to indicate that the transaction has truly ended. In the event of a crash while a transaction is freeing space, the recovery module searches all the store metadata for stores marked "for deletion" and deletes those that would otherwise have been missed in the redo phase.

Pages are reserved for a store in units of `ss_m::ext_sz,` a compile-time constant that indicates the size of an *extent*. An extent is a set of contiguous pages. Extents are represented by persistent data structures, `extnode_t`, which are linked together to form the entire structure of a store. A `stnode_t` holds metadata for a store and sits at the head of the chain of extents that forms the store, and the extents in the store list are marked as having an owner, which is the store id of the store to which they belong. A store id is a number of type `snum_t`, and an extent id is a number of type `extnum_t`. Scanning the pages in a store can be accomplished by scanning the list of `extnode_t`s. Each extent has a number, and the pages in the extent are arithmetically derived from the extent number; likewise, from any page, its extent can be computed. Free extents are not linked together; they simply have no owner (signified by an `extnode_t::owner == 0`).

7.3.1 Page allocation and deallocation

Searching for free extents is a round-robin scheme that involves inspecting the `extnode_t` structures on a volume. Since these structures sit at the head of the volume on pages dedicated to them, such searches are relatively inexpensive. Similarly, searches for free `stnode_t` structures are limited to a few pages at the head of a volume. When a volume is formatted, the first several pages of the volume are dedicated to store and extent metadata; the number of pages so dedicated is a function of the size of the volume given to the format method by the server. Store numbers and extent numbers are 32-bit unsigned values, so there is a limit of 4 billion of each on a volume.

Page allocation depends on a *policy* that differs with context: When objects are appended to a file by means of the append-object API (`append_file_i`), pages are allocated at the end of the file by locating the last extent and allocating pages in that extent, and adding extents as needed. When objects are created by any other API method, extents in the file are searched and unallocated pages in allocated extents may be used. Such pages are found by looking in a cache of recently-used pages, and by a linear search through the file as a last resort. Before resorting to a linear search, the storage manager consults histogram (hints) that it keeps for recently-used files, to see if a search is likely to yield anything.

Pages and extents that are freed by a transaction are kept in a list for deferred deallocation when the transaction commits. This prevents other transactions from allocating such pages and extents and thereby interfering with rollback.

7.3.2 One-page storage structures

Creating a file or an index normally results in allocating a minimum of one extent , since the storage structures allocate stores An extent is a set of contiguous pages, so, this can results in waste if the storage structure will never need so many pages. To avoid this waste, a volume has a special store whose pages can be allocated to storage structures individually (rather than by the extent). This is used by the B-tree implementation to allow indexes to start out small (one page) and grow to a store when needed. (This can only be done if logical IDs are used for the index, since the index is known by its store ID when physical IDs are used.) This mechanism was put into place for the higher layers of SHORE, which implemented Unix-style directories as small B-trees.

```
Files: sm_io.h, sm_io.cpp, vol.h, vol.cpp
```

7.3.3 Space reservation on a page

Because fine-grained locking (slot-level -- i;.e., records in files) is the default, special care must be taken to reserve space on a page when slots are freed (records are deleted) so that rollback can restore the space on the page. In the case of B-trees, this is not a concern, since undo and redo are handled logically -- entries can be re-inserted in a different page. But in the case of files, records are identified by physical ID, which includes page and slot, so records must be reinserted just where they first appeared. Like Mohan's scheme[7], the transaction freeing space can re-use the space it freed. Unlike Mohan's scheme, this algorithm does not distinguish contiguous free space and non-contiguous free space, nor does it use the *Commit_LSN* idea to identify committed data.

Holes in a page are coalesced (moved to the end of the page) as needed, when the total free space on the page satisfies a need but the contiguous free space does not. Hence, a record truncation followed by an append to the same record does not necessarily cause the shifting of other records on the same page.

Space-reservation metadata are kept in a

```
class space_t {
public:
      int   usable(xct_t* xd);

      rc_t  acquire(int amt, int slot_bytes,
                        xct_t* xd, bool do_it=true);
      void  release(int amt, xct_t* xd);

      void  undo_acquire(int amt, xct_t* xd);
      void  undo_release(int amt, xct_t* xd);
```

```
private:
      tid_t       _tid;
      int2_t      _nfree;
      int2_t      _nrsvd;
      int2_t      _xct_rsvd;
};
```

When a transaction destroys a record, it releases the space for the record's slot by calling the method `release(amount_in_bytes, transaction_ptr)`. The `space_t` keeps track of

- the number of *free* bytes on the page (`_nfree`)
- the number of *reserved* bytes on the page (`_nrsvd`)
- the transaction ID of the *youngest* active transaction freeing space on this page (`_tid`)
- the number of reserved bytes freed by the youngest transaction (`_xct_rsvd`).

The free byte count is maintained for all page types. The rest are kept only for those page types that require space reservation.

When the youngest transaction to reserve space on a page commits, the reserved space on that page can be released, and the page's `_tid`, `_nrsvd`, and `_xct_rsvd` are cleared.
A new page has no pre-allocated slots. Creating a record consists of

- find an allocated but unused slot, if there is one, and acquire space for the record
- if there are no unused slots on the page, allocate a new slot -- this means acquire space for the record and for the slot.

Destroying a record consists of releasing the space for the record and marking the slot free. Space for the slot is never released because slots that are in use must keep their slot IDs (array indices).

During rollback, a transaction can use any reserved space. During forward processing, a transaction can use any space that *it* reserved. The space that was reserved by a transaction is known only if that transaction is the youngest one reserving space on the page, so this scheme does not always allow every transaction to make the maximum use of its reserved space on a page, but this scheme is simple and requires very little overhead on the page, while guaranteeing that transactions can roll back and reclaim whatever space they need on each page.

None of the changes to metadata in `space_t` are logged. Instead, the operations of freeing a slot and allocating a slot are logged (with log record types `page_mark` and `page_reclaim`, respectively).

```
Files: page.h, page.cpp, page_s.h
```

7.4 Buffer manager (bf_m)

The buffer manager is the means by which all other modules (except the log manager) read and write pages. A page is read by issuing a `fix` method call to the buffer manager. If the page requested cannot be found in the buffer pool, the requesting thread blocks, waiting for a cooperating process or thread to read in the page.

On Unix, the buffer pool is in shared memory, where cooperating processes can find and place pages in response to read and write requests. On NT, a cooperating NT thread performs the read. The buffer pool's size is a run-time configuration option. All frames in the buffer pool are the same size, and they cannot be coalesced, so the buffer manager manages a set of pages of fixed size, and all cooperating processes trade in pages of the given size.

The buffer manager forks background threads to flush dirty pages to their respective disks. It makes an attempt to avoid hot pages and to minimize the cost of I/O by sorting and coalescing requests for contiguous pages. (This is the extent of disk scheduling that is done: the `diskrw` process/thread does not do disk scheduling.) Groups of contiguous pages are written to disk in a single vectored write request. Statistics kept by the buffer manager tell the number of resulting write requests of each size.

The buffer manager writes dirty pages even if the transaction that dirtied the page is still active (steal policy). Pages stay in the buffer pool as long as they are needed, except when chosen as a victim for replacement (no force policy).

The replacement algorithm is clock-based (it sweeps the buffer pool, noting and clearing reference counts). This is a cheap way to achieve something close to LRU; it avoids much of the overhead and mutex bottlenecks associated with LRU.

7.4.1.1 Fixing pages

The buffer manager maintains a hash table that maps page IDs to buffer frame control blocks (`bfcb_t`), which in turn point to frames in the buffer pool. The `bfcb_t` keeps track of the page in the frame, the page ID of the previously-held page, and whether it is in transit, the dirty/clean state of the page, the number of page fixes (pins) held on the page (i.e., reference counts), the *recovery LSN* of the page (see Log manager, Checkpoints, below), etc. The control block also contains a latch. A page is always fixed in a *latch mode*, either `LATCH_SH` or `LATCH_EX.` Page fixes are expensive (in CPU time, even if the page is resident.

Each page type defines a set of *fix* methods that are virtual in the base class for all pages: The rest of the storage manager interacts with the buffer manager primarily through these methods of the page classes. The macros `MAKEPAGECODE` are used for each page subtype; they define all the `fix` methods on the page in such a way that `bf_m::fix()` is properly called in each case.

```
static rc_t       fix(
      page_s*&          page,
      const lpid_t&           pid,
      uint2_t                 tag,
      latch_mode_t            mode,
      bool              no_read,
      store_flag_t&           out_stflags,
      bool              ignore_store_id = false,
      store_flag_t            stflags = st_bad);
```

`Fix()` looks up the page identified by `pid` in the buffer pool, reading the page in if necessary. The `tag` indicates the type of page this page should be (or will become), `mode` indicates the latch mode to use when latching the page (frame). `No_read`, if true, prevents the buffer manager from reading in the page if it's not found. This allows the caller to grab a page frame and immediately format the page in that frame. It is used when pages are being allocated. The `out_stflags` is where the buffer manager returns the flags that describe the logging characteristics of the store to which the page belongs. This is a performance hack -- it allows the storage manager to avoid excess looking-up of store flags, which might entail reading another page (the page containing the store header). If `ignore_store_id` is true, the buffer manager allows the store ID on the page to differ from that in the page ID given. This happens when a page is deallocated from one store and is allocated to another. The page is not always written to disk after the deallocation. (For example, when a store is destroyed, rather than visit each page in the store to change the store ID on the page, the head of the store is marked free, and the pages are updated when they are read in by the buffer manager.) When a new page is being allocated and formatted, a `fix()`call is made to allocate a frame for the page, and the new store flags are passed to the buffer manager in that call. The buffer manager keeps track of the store flags for each page (this is kept in the control block) because the buffer manager is responsible for ensuring the WAL protocol is used; this involves inspecting the LSN on each page, but pages of temporary files do not have legitimate LSNs, since changes to those pages are not logged. All this means that the buffer manager has to treat some pages as special cases, and those cases are detected by inspecting the store flags for a page.

If the page is still in the buffer pool, a refix is faster than a fix:

```
static rc_t             refix(
      const page_s*           p,
      latch_mode_t            mode);
```

Likewise, if a page is fixed, latched in share mode, and an exclusive latch is needed, it is preferable to upgrade the latch than to unfix the page and fix it again in the desired mode:

```
static void             upgrade_latch(
      page_s*&                p,
      latch_mode_t            m);
```

When unfixing a page, the caller determines whether the page is to be marked dirty. The caller also has an opportunity to fool the clock algorithm by setting the ref "bit" to something other than the default:

```
static void       unfix(
      const page_s*&          buf,
      bool                    dirty = false,
      int                     refbit = 1);
```

To mark a page dirty without unfixing the page, use the following, which is used by the generated logging code:

```
static rc_t       set_dirty(const page_s* buf);
```

The following methods allow pages or sets of pages to be discarded from the buffer pool without being written to disk One can discard a single page, all the pages in the buffer pool that belong to a given store or volume, or the entire buffer pool.

```
static void       discard_pinned_page(
                        const page_s*& buf);
static rc_t       discard_store(stid_t stid);
static rc_t       discard_volume(vid_t vid);
static rc_t       discard_all();
```

One or more may be forced to disk (and discarded from the buffer pool if the argument `flush` is `true`):

```
static rc_t       force_store(
      stid_t                  stid,
      bool              flush = false);
static rc_t       force_page(
      const lpid_t&           pid,
      bool              flush = false);
static rc_t       force_volume(
      vid_t                   vid,
      bool                    flush = false); static
rc_t                    force_all(bool flush = false);
```

7.4.1.2 Buffer manager internals

Unless you are hacking the buffer manager, you can safely skip this section.

The buffer manager implementation is split into two files, `bf.cpp` and `bf_core.cpp`. `Bf_core.cpp` contains the thread-safe hash table implementation, which takes care of latching the pages. The following methods of class `bf_core_m` bear discussion. The type `bfpid_t` is the buffer manager's notion of a page ID. The buffer manager uses `bfpid_t` to override the `operator==` for page IDs because *in the context of the buffer manager, pages are identified by volume ID and page number only; the store ID is not germane*, and if used in a comparison, yields errors because pages can move from one store to another.

```
w_rc_t      find(
      bfcb_t*&          ret,
      const bfpid_t& p,
      latch_mode_t      mode = LATCH_EX,
      int               timeout =
            sthread_base_t::WAIT_FOREVER,
      int4_t            ref_bit = 0
);
```

`Find()` is the basic look-up function. It searches the hash table for a frame containing the page identified by `p.` The method returns in error if the page is not found. If the page is found, it is latched in the given mode. If the page cannot be read before the given timeout expires, the method returns in error. The last argument allows the caller to set the reference "bit" (which is not a bit) to fool the clock algorithm used to clean the buffer pool.

```
w_rc_t      grab(
      bfcb_t*&          ret,
      const bfpid_t& p,
      bool&       found,
      bool&       is_new,
      latch_mode_t      mode = LATCH_EX,
      int               timeout =
            sthread_base_t::WAIT_FOREVER);
```

The method `grab()` finds a frame into which the caller will read a page identified by page ID `p.`. First, it looks for the page in the table; if found, it latches the frame and returns. If the page is not in the table, a frame is grabbed, latched, and marked as *in-transit*. It might be that the frame chosen contains another page, and that other page needs to be replaced. The caller then forces out the old page before it reads in the page it wants. Any function that calls `grab()` must subsequently call `publish()` after the frame is filled.

```
void        publish_partial(bfcb_t* p);
void        publish(
      bfcb_t*     p,
      bool        error_occured = false);
```

`Publish()` and `publish_partial()` change the state of the control regarding in-transit pages. When `grab()` returns an in-use frame and a page replacement occurs, the caller uses `publish_partial()` after flushing out the old page. `Publish_partial()` wakes up any threads that were waiting on the page that was written out.. After the desired page is read into the frame, `publish()` is called to make the new page generally available to other threads, and to wake up any threads waiting for the new page.

```
w_rc_t      pin(
      bfcb_t*     p,
      latch_mode_t      mode = LATCH_EX);

void        unpin(
      bfcb_t*&    p,
```

```
      int          ref_bit = 0,
      bool         in_htab = true);
```

`Pin()` and `unpin()` adjust the pin count of a frame, given the control block. Using these methods bypasses the hash-table lookup.

```
Files: bf.h, bf.cpp, bf_core.h, bf_core.cpp
```

7.5 Storage structures

The storage manager supports three storage structures. Storage structures may have their own navigable structures in addition to the underlying structure of the store(s) that compose them. In the case of small-object files, no such structure exists; in the case of B-trees, of course, it does.

7.5.1 The directory (dir_m)

Every storage structure that is accessible to a server through the API appears in a *store directory*. The directory is a B-tree index that contains information about the storage structures.

The index key is the identifier of the storage structure (`stid_t`, which serves as a file id, a B-tree id, an rtree id, and a store id). The data stored for each store (`sdesc_t`) includes some persistent information and some transient information (e.g., an approximate last page of the store). The persistent data are stored in an `sinfo_s`, and include

- kind of structure: B-tree, file, rtree
- if a B-tree, is it unique?, what concurrency control does it use (kvl, IM, none)?
- what stores compose this storage structure?
- what page is the root of this storage structure?
- if this is an index, what is the type of the key of this index?

Store descriptors are cached so that the disk need not be used every time a storage structure is looked up in the directory.

```
Files: dir.h, dir.cpp, sdesc.h
```

7.5.2 File manager (file_m)

Files are groups of variable-sized records (or "objects"). A record is the smallest persistent datum that has identity. Records may also have headers. As records vary in size, so their storage representation varies. The storage manager changes the storage representations as needed.

A file comprises one or two stores. One store is always allocated for slotted (small-record) pages. If any large records exist in the file, the file will also contain a store for large-record pages.

The scan order of a file is the physical order of the records in the file. See the note in the section below, **Scanning files.**

Every record, large or small, has metadata called a *tag*, represented by `class rectag_t`, which contains the following (and more).

```
struct rectag_t {
      uint2_t   hdr_len;// length of user header
      uint2_t   flags;// enum recflags_t
```

```
      smsize_t  body_len;     // true length of the record
      serial_t  serial_no;// logical serial number in file
};
```

The `hdr_len` tells the length of the "user header", which may be of size 0. The `flags` is a bit mask that indicates, among other things, the representation used to store the record. The bit mask will include one of the following values:

- `t_small` the record is small enough to fit on a single file page.
- `t_large_0` the record is too large to fit on a single file page; it has the representation described below
- `t_large_1` the record was converted from a `t_large_0` to a representation that uses a 1-level tree
- `t_large_2` this representation uses a 2-level tree

Every record's tag can be manipulated (internally by the storage manager -- not by a server) through

```
class record_t {
public:
      // You can look directly at the tag
      rectag_t  tag;

      bool is_large() const;
      bool is_small() const;

      // get size of user header or body
      smsize_t hdr_size() const;
      smsize_t body_size() const;
      // get pointer to user header or
      // some part of body -- whatever is
      // pinned
      const char* hdr() const;
      const char* body() const;

      // offset of the part of the body that's pinned
      int body_offset() const

      // page id of page containing the given offset
      // into the record's body
      lpid_t pid_containing(smsize_t offset,
            smsize_t& start_byte,
            const file_p& page) const;
};
```

There are two APIs for servers to manipulae individual objects:

- methods of the storage manager class (`ss_m::create_rec ()`, `ss_m::append_rec()`, etc.), and
- the class `pin_i`. (`Pin_i` is an *iterator* over the pages of an object, hence its name.) The `pin_i` class permits small records or single pages of large records to be pinned in the buffer pool under the control of the server. It contains methods for reading and updating records in whole or in part.

Regardless of the API used, updates to records are accomplished by copying out part or all of the record from the buffer pool to the server's address space, performing the update there, and handing to the storage manager the new data (part or all of the updated record). This simplifies logging of updates in the storage manager.

When a record is *pinned* (fixed in the buffer pool), the page containing the header and tag may be in the buffer pool, and at the same time, at most one page of the record's body is placed in the buffer pool. A server manipulates the pinned state of a record through the

```
      class pin_i {
```

```
public:
// flags that describe the pin state of the record
      enum flags_t {
            pin_empty         = 0x0,
            pin_rec_pinned          = 0x01,
            pin_hdr_only            = 0x02,
            pin_separate_data = 0x04,
            pin_lg_data_pinned      = 0x08  // large data page is
            pinned
      };

      pin_i();
      ~pin_i();

      // pin the record identified by “rid”.  We’re
      // interested in the portion of the record that
      // starts at “start”
      // lock the record in the given mode.
      w_rc_t      pin(
            const rid_t rid,
            smsize_t start,
            lock_mode_t lmode = SH);

      // release the record from the buffer pool
      void unpin();

      // repin efficiently repins a record after
      // its size has changed, or after it has been
      // unpinned.
      w_rc_t repin(lock_mode_t lmode = SH);

      // conditionally pin: only if the page is
      // cached in the buffer pool.
      w_rc_t            pin_cond(
            const rid_t&          rid,
            smsize_t        start,
            bool&           pinned,
            bool            cond = true,
            lock_mode_t     lmode = SH);

      // pin the next range of bytes in the record,
      // freeing the page of the body that’s now pinned
      // Parameter eof is set to true if there are
      // no more bytes to pin.
      // When eof is reached, the previously
      // pinned range remains pinned.
      w_rc_t            next_bytes(bool& eof);

      // is something currently pinned
      bool              pinned() const;

      // is the entire record pinned
      bool              pinned_all() const

      // return true if pinned *and*
      // pin_i is up-to-date with the LSN on
      // the page.  in other words, verify that
```

```
            //  the page has not been changed by another
            // pin_t after this pin_i pinned the page
            bool        up_to_date() const;

            // return offset of first byte pinned
            smsize_t    start_byte() const;
            // how many bytes from start_byte() are pinned
            smsize_t    length();

            // size of user header
            smsize_t    hdr_size() const;
            // size of entire body
            smsize_t    body_size() const;

            bool  is_large() const;
            bool  is_small() const;

            // record id
            const rid_t&      rid() const;
            // pointer to user header
            const char* hdr() const;
            // pointer to first pinned byte of body
            const char* body();

            // These record update functions
            // duplicate those in class ss_m
            // and are more efficient.
            // They can be called on any pinned record
            // regardless of where and how much is pinned.
            w_rc_t      update_rec(smsize_t start,
                  const vec_t& data, int* old_value = 0);
            w_rc_t      update_rec_hdr(smsize_t start,
                  const vec_t& hdr);
            w_rc_t      append_rec(const vec_t& data);
            w_rc_t      truncate_rec(smsize_t amount);

            lpid_t      page_containing(smsize_t offset,
                  smsize_t& start_byte) const;
};
```

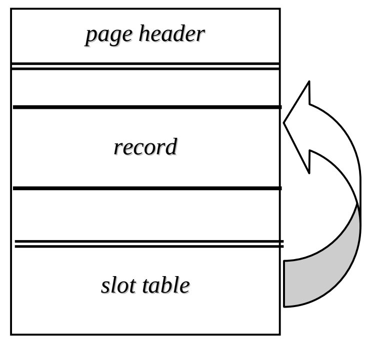
When a server manipulates a record through the `ss_m` methods, the storage manager pins the records, performs the operation and unpins the record. When a server manipulates a record through the `pin_i` methods, the record remains pinned between storage manager operations. *The server must unpin or repin records at the proper times in order to avoid latch-latch deadlocks or latch-lock deadlocks.* Pinning a record amounts to

1. locking the record (and page, per hierarchical locking)
2. fixing the page in the buffer pool (acquiring a latch on the page for the thread that performs the pin)
3. computing the page offset of the record's tag

If a server pins two different records that reside on the same page, an update to one of the records might invalidate the `pin_t` for the other record because the update might cause the page to be rearranged and the offsets to change. For this reason, a server might have to call the `repin()` method. Furthermore, if the server has two or more threads operating on behalf of the same transaction, the lock acquisition will provide no deadlock protection among the threads, and the storage manager does not perform deadlock detection for latches. Finally, if two or more threads pin more than one record each, on behalf of two different transactions, and if a strict ordering is not imposed on the pins, the two threads may find themselves in latch-lock deadlock.

7.5.2.1 Small records (`t_small`)

Small records are those small enough to fit (one or more) on a slotted page. When a small record grows too large to fit on a slotted page, the storage manager moves the object and changes its storage representation.

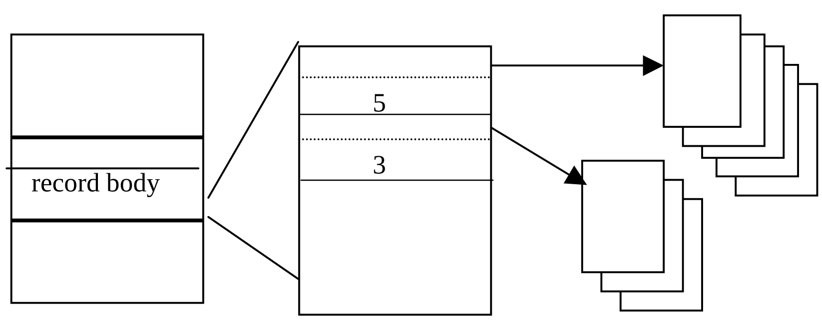


7.5.2.2 Large records

Every large record has a small amount of metadata on a slotted page. It may also have a header on the slotted page.

7.5.2.2.1 Large records: `t_large_0`

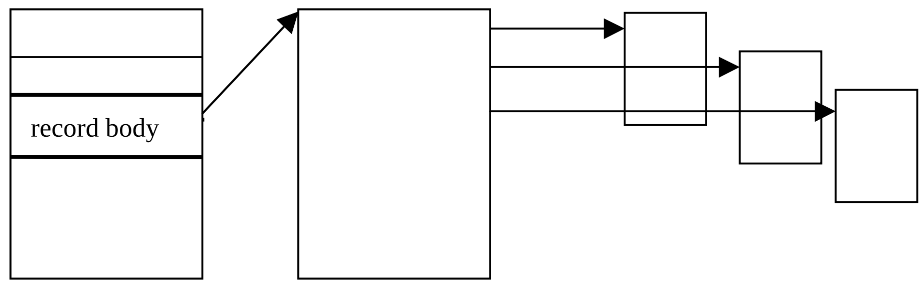
The body portion of the record in the slotted page is a list of chunks (`lg_tag_chunks_s`). A chunk represents a set of contiguous pages, up to 65536 pages long (since 2 bytes are used for the length of a chunk). Thus, for a storage manager compiled with 8K pages, a `t_large_0` record is between 8096 (largest small object size) and about 21 gigabytes in size (if each chunk is 65K pages long). If there is bad clustering in the object, a `t_large_0` might have to be converted to a `t_large_1` when it is only 5 pages in size.



7.5.2.2.2 Large records: `t_large_1, t_large_2`

The body portion of the record in the slotted page contains no information about leaf pages; rather, it points to the root of a 1-level index for the record (it is a `lg_tag_indirect_s`). Large record index pages use one slot, which is simply an array of page Ids. Thus, for 8K-sized pages, 2030 page Ids fit, meaning that a `t_large_1` record can be up to about 16.4 megabytes. 2-level indexes (`t_large_2`) are not yet implemented.

A `t_large_1` record looks like this:



The storage manager automatically performs record-level, two-phase locking by default. The server can choose coarser granularity for the locking, and (as described later) it also has limited ability to free locks.

7.5.2.3 Scanning files

The API for scanning files is the iterator class `scan_file_i`, which provides optional 1-page prefetch. The prefetch mechanism was added to see if the access patterns of the Paradise server offered sufficient opportunity to exploit prefetch that it might be worth re-designing the interface between the buffer manager and the I/O layer to allow a “real” implementation of prefetching. The existing prefetch mechanism consists of a thread that communicates with the scan iterator, and accepts prefetch requests from the iterator. The thread then fixes the page and unfixes it, in the hope that by the time the scan iterator fixes the page, the page will then be in the buffer pool. This is a haphazard way to accomplish prefetch, and involves a lot of overhead: fixing pages is not cheap.

The storage manager provides a class for rapidly appending objects to a file (`append_file_i` -- a misnomer because it is not an iterator, but it derives from `scan_file_i`). This amounts to keeping the last page in a file fixed in the buffer pool (through the `append_file_i)` to avoid excess fixing of that page.

The scan order of a file is the physical order of records in the file. If the `append_file_i` API is used to append records to a file, the scan order will be the append order. If any other methods are used to create objects in the file, all bets are off.

7.5.2.4 Allocating space for records

When the `append_file_i` API is used to create records, the storage manager keeps the records in the order in which they were appended; this can waste disk space, but it preserves the sort order of sorted files. When any other method is used to create records, a different policy applies. The storage manager keeps a small cache of recently-used pages for each store; if a request cannot be satisfied by pages in this cache, the storage manager checks a histogram describing the store to determine whether it is worthwhile to perform a linear search of the file for allocated-but-unused space.

```
Files: sm.h, pin.h, pin.cpp, smfile.cpp, file.h, file_s.h, file.cpp.
lgrec.h, lgrec.cpp, scan.h, scan.cpp
```

7.5.3 B-tree manager (btree_m)

The B-tree indexes are B+-trees, but "B-tree" is commonly used in the sources and documentation.

When a B-tree is created, the server ("user" of the storage manager API) chooses
1. the key type for the index, from

- signed, unsigned 1-, 2-, 4-byte integers
- fixed-length strings
- variable-length strings
- IEEE float and double
- combinations of the above

2. optional prefix compression
3. which locking protocol to use with the index, among
   - no locking
   - KVL
   - IM
   - A variant of KVL that provides higher concurrency by avoiding next-key locking, but does not avoid phantoms, and so it does not permit range scans.

The values associated with the keys are opaque to the storage manager, except when IM is used, in which case the value is treated as a record ID, but no integrity checks are done. It is the responsibility of the server to see that the value is legitimate in this case.

B-trees can be bulk-loaded from files of sorted key-value pairs.

The implementation of B-trees is straight from the Mohan ARIES/IM and ARIES/KVL papers. Those two papers give a thorough explanation of the arcane algorithms, including logging considerations. Anyone considering changing the B-tree code is strongly encouraged to read these papers carefully. Some of the performance tricks described in these papers are not implemented here. For example, the ARIES/IM paper describes performance of logical undo of insert operations if and only if physical undo is not possible. The storage manager always undoes inserts logically.

```
Files: sm.h, smindex.cpp, btree.h, btree.cpp, btree_impl.h,
btree_impl.cpp, btree_p.h, btree_p.cpp,  btcursor.h, btcursor.cpp,
btree_bl.cpp, lexify.h, lexify.cpp, sort.h, sort_s.h, sort.cpp
```

7.5.4 R-tree manager (rtree_m)

The spatial indexes in the storage manager are R*-trees[8], a variant of R-trees that perform frequent restructuring to yield higher performance than normal R-trees. The entire index is locked.

```
Files: sm.h, smindex.cpp, rtree.h, rtree.cpp
```

7.6 Lock manager (lock_m)

The lock manager's API allows explicit acquisition of locks by other modules in the storage manager. Freeing locks is automatic at transaction commit and rollback. There is limited support for freeing locks in the middle of a transaction. This is accomplished by creating a *quark* (`sm_quark_t`). A quark is a marker in the list of locks held by a transaction. When the quark is destroyed, all locks acquired since the creation of the quark are freed. Quarks cannot be used while more than one thread is attached to the transaction, although the storage manager does not strictly enforce this (due to the cost).

The lock manager enforces two lock hierarchies:

- Volume, store, page, record
- Volume, store, key-value

A lock identifier (`lockid_t`) contains all the necessary information for the lock manager to do this. The lock manager does not verify that lock identifiers refer to any existing object. Other than the way the lock identifiers are inspected for the purpose of enforcing the hierarchy, lock identifiers are considered opaque data by the lock manager.

The lock manager escalates up the hierarchy by default. The escalation thresholds are based on run-time options. They can be controlled (set, disabled) on a per-object level. For example, escalation to the store level can be disabled when increased concurrency is desired. Escalation can also be controlled on a per-transaction or per-server basis.

The lock manager contains hooks for distributed deadlock detection (described later).

The lock manager uses a hash table whose size is determined by a configuration option. The hash function used by the lock manager is known not to distribute locks evenly among buckets. This is partly due to the nature of lock IDs. The lock manager code contains several hash functions that can tested, and the lock manager can be instrumented with some expensive statistics to measure the effectiveness of the hash function. To do so, you must

```
#define EXPENSIVE_LOCK_STATS
```

(which can be done in the generic configuration file, `shore.def`).

In anticipation of porting the storage manager to a preemptive threads environment, the lock manager can be configured to use a mutex on each bucket rather than a mutex on the entire table.

```
Files: lock.h, lock.cpp, lock_core.h, lock_core.cpp, lock_s.h
lock_s_inline.h, lock_x.h
```

7.7 Transaction manager

Transaction management is not encapsulated into a single manager; rather, it is largely implemented in the class `xct_t` (a transaction) with certain functionality in other managers. For example transaction management is closely related to lock management, and there are some classes that know a little about both (`xct_lock_info_t`, `lock_cache_t`).

By default, transactions have standard (degree 3) ACID properties. The storage manager assigns a local transaction ID when a transaction is begun (the server cannot choose its transaction Ids). Distributed transactions are not fully implemented in the storage manager, however there are all the hooks necessary for full distributed transactions to be implemented in a server. The server is responsible for bookkeeping (associating a global transaction ID with a set of local transaction IDs -- one in each server process). The storage manager API contains functions that allow the server to inform the storage manager of the coordinator's name (opaque to the storage manager) for logging purposes, for preparing a local transaction, and for recovering prepared transactions after a crash-restart. More will be said about distributed transactions later. For the purpose of this section, a transaction refers to a local transaction (which might happen to be a "branch" of a global transaction).

The server may choose to chain transactions (commit one transaction, begin another, and transfer the locks from the committed to the new transaction). Partial rollback is provided through savepoints.

7.7.1 Multi-threaded transactions

Several server threads (threads of control, as opposed to a branch of a global transaction) may participate in a transaction under limited circumstances. Participating in a transaction occurs when any storage manager activity occurs while the thread is "attached" to a transaction. (Attaching consists in the existence of a reference from a thread instance to a transaction instance. Threads may be attached to no transaction.) For example, the transaction cannot be committed or rolled back (fully or to a savepoint) while more than one thread is attached to a transaction. Only one thread of the transaction can use the log at a given time, since the transaction ID written in log records does not contain any thread ID.

7.7.2 Transactions and logging

Each transaction caches the last log record written until that log record is forced to the log by the logging protocol. Caching the last record allows compensations to be piggy-backed on the log record, reducing logging overhead. Log records for redoable-undoable operations contain both the redo- and undo- data, hence an operation never causes two different log records to be written for redo and for undo. This, too, controls logging overhead.

The protocol for applying an operation to an object is as follows:
1. Lock the object.
2. Fix the page(s) affected in exclusive mode.
3. Apply the operation.
4. Write the log record(s) for the operation.
5. Unfix the page(s).

The protocol for writing log records is as follows:
1. Grab the transaction's buffer in which the last log record is to be cached by calling `xct_t::get_logbuf(logrec_t*& l)`
2. Write the log record in the buffer.
3. Release the buffer with
   `xct_t::give_logbuf(logrec_t *l, const page_p *page)`,
   passing in the second argument the fixed page that was affected by the update being logged. This does several things:
   1. writes the transaction ID, previous LSN for this transaction into the log record
   2. flushes the record to the log and remembers this record's LSN
   3. marks the given page dirty.

Between the time the log buffer is grabbed and the time it is released, the buffer is held exclusively by the one thread that grabbed it, and updates to the log buffer can be made freely.

The above protocol is enforced by the storage manager in helper functions that create log records; these functions are generated by Perl scripts from the source file `logdef.dat`. (See Log manager, Log record types below.)

Some logging records are *compensated*, meaning that the log records are skipped during rollback. Compensations may be needed because some operation simply cannot be undone. The protocol for compensating actions is as follows:

1. Fix the needed pages.
2. Grab an *anchor* in the log. This is an LSN for the last log record written for this transaction.
3. Update the pages and log the updates as usual.
4. Write a compensation log record and free the anchor.

Grabbing an anchor prevents all other threads in a multi-threaded transaction from gaining access to the transaction manager.

In some cases, the following protocol is used to avoid excessive logging by general update functions that, if logging were turned on, would generate log records of their own.

1. Fix the pages needed in exclusive mode.
2. Turn off logging for the transaction.
3. Perform the updates by calling some general functions. If an error occurs, undo the updates explicitly.
4. Turn on logging for the transaction.
5. Log the operation. If an error occurs, undo the updates with logging turned off..

6. Unfix the pages.

The mechanism for turning off logging for a transaction is to construct an instance of `xct_log_switch_t.` When the instance is destroyed, the original logging state is restored. The switch applies only to the transaction that is attached to the thread at the time the switch instance is constructed, and it prevents other threads of the transaction from using the log (or doing much else in the transaction manager) while the switch exists.. An example from the B-tree code:

```
{// open scope
xct_log_switch_t toggle(OFF);
rc = leaf.insert(key, el, slot);
if(rc) {
      leaf.discard();
      return rc.reset(); // force caller to check rc
}
}// close scope
```

7.7.3 Concurrency control

The storage manager locks data when they are read and updated. The locking is implicit, but explicit locking can be done with calls to `ss_m::lock()`. Arbitrary unlocking is not allowed. Two-phase locking is used unless *quarks* are used by the server (see Lock manager, below). Each transaction keeps a list of the locks it holds, so that the locks can be logged when the transaction is prepared and released at the end of the transaction. Furthermore, to avoid expensive lock manager queries, each transaction keeps a cache of the last 5 locks of each kind in the lock hierarchies. This close association between the transaction manager and the lock manager is encapsulated in several classes in the file

```
lock_x.h
```

which is well-commented.

`Files: lock_x.h, sm.h, sm.cpp, xct.h, xct.cpp xct_impl.h, xct_impl.cpp.`

7.8 Log manager (log_m)

The storage manager performs ARIES-style logging and recovery. This means the logging and recovery system has these characteristics:

- uses write-ahead logging (WAL)
- repeats history on restart before doing any rollback
- all updates are logged, including those performed during rollback
- compensation records are used in the log to bound the amount of logging done and guarantee progress in the case of repeated failures and restarts.

7.8.1 Log Sequence Numbers (LSNs)

Write-ahead logging requires a close interaction between the log manager and the buffer manager: before a page can be flushed from the buffer pool, the log might have to be flushed.
This also requires a close interaction between the transaction manager and the log manager.
All three managers understand a log sequence number (LSN) , implemented by

```
class lsn_t.
```

Log sequence numbers serve to identify and locate log records in the log, to timestamp pages, identify timestamp the last update performed by a transaction, and the last log record written by a transaction. Since every update is logged, every update can be identified by a log sequence number. Each page bears the log sequence number of the last update that affected that page.
A page cannot be written to disk until the log record with that page's LSN has been written to the log (and is on stable storage).

A log sequence number is a 64-bit structure, with a 32-bit `lsn_t::hi`() and a 32-bit `lsn_t::lo()`.

7.8.2 Log partitions

The log is partitioned to simplify archiving to tape (future work).The log comprises 8 partitions, where each partition's size is limited to approximately 1/8 the maximum log size given in the run-time configuration option `sm_logsize`.

In the Unix-files case, the configuration option `sm_log` names a directory (which must exist before the storage manager is started) in which the storage manager may create and destroy files. In the raw case, the option `sm_log` names a raw disk partition, which the storage manager further splits into 8 log partitions (these "partitions" are not to be confused with disk partitions). The storage manger may have at most 8 active partitions at any one time. An active partition is one that is needed because it contains log records for running transactions. Such partitions could (if it were supported) be streamed to tape and their disk space reclaimed. Space is reclaimed when the oldest transaction ends and the new oldest transaction's first log record is in a newer partition (call this Pnew) than that in which the old oldest transaction's first log record resided. When this happens, all partitions created before Pnew are superfluous and their space is reclaimed. Until tape archiving is implemented, the storage manager issues an error (`eOUTOFLOGSPACE`) if it consumes sufficient log space to be unable to abort running transactions and perform all resulting necessary logging within the 8 partitions available. *Determining the point at which there is insufficient space to abort all running transactions is a heuristic matter and it is not reliable*. Ultimately, archiving to tape is necessary. The storage manager does not perform write-aside or any other work in support of long-running transactions.

In the Unix-file case, a partition is a file called

```
log.N
```

where `N` is the `N`th partition created.

The high 32 bits of an LSN identifies the partition in which the log record with that LSN resides.
The low 32 bits is a byte-offset in that partition.

7.8.3 Checkpoints

The class

```
chkpt_m
```

sleeps until kicked by the log manager, and when it is kicked, takes a checkpoint, then sleeps again. Taking a checkpoint amounts to these steps:

1. Write a `chkpt_begin` log record.
2. Write a series of `chkpt_dev_tab` log records to log the device table (list of volumes and devices mounted).
3. Write one or more `chkpt_bf_tab` log records to record the buffer pool's dirty-page information. For each dirty page in the buffer pool, the page id and its *recovery LSN* (`rec_lsn`) is logged. A page's recovery LSN is metadata stored in the buffer manager's control block, but is not written on the page. It represents an LSN prior to or equal to the log's current LSN at the time the page was first marked dirty. Hence, it is less than or equal to the LSN of the log record for the first update to that page after the page was read into the buffer pool (and remained there until this checkpoint). The minimum of all the *recovery LSNs* written in this checkpoint will be a starting point for crash-recovery, if this is the last checkpoint completed before a crash.
4. Write one or more `chkpt_xct_tab` log records to record the states of the transactions.
5. Log the prepared transactions, using the same log records used for a normal transaction-prepare operation.
6. Write a `chkpt_end` record.
7. Tell the log manager where this checkpoint is: the LSN of the `chkpt_begin` record becomes the new *master LSN* for the log. The master LSN is written in a special place in the log so that it can always be discovered when the log manager is constructed (on restart).

Certain things cannot occur during a checkpoint, but otherwise, the checkpoint records can be interspersed with other log records. What cannot occur are:

- Certain logging during a transaction-prepare operation.
- Certain logging during a transaction-commit operation.
- Mounting a volume.
- Dismounting a volume.

7.8.4 Recovery

When the storage manager object is constructed, the logs are inspected, the start of the last completed checkpoint is located, and its LSN is remembered as the `master_LSN.` Recovery is performed in these phases:

1. ANALYSIS: The analysis pass begins analyzing the log at the `master_LSN`. It reads the log records of the last completed checkpoint and reconstructs the transaction table, an in-memory dirty page table, and the device table, mounting devices and volumes as needed. From the dirty page table, it computes the LSN where recovery must begin, `redo_lsn,` which is the lowest of all the recovery LSNs in the dirty page table.

2. REDO: The redo pass scans the log, starting at `redo_lsn`, and for each log record it encounters, decides whether or not that log record's action must be redone. The general rule for redoing a log record's operation is:

   If the log record is not *redoable*, the log record is ignored. A redoable record might contain a page ID. If it does not, the redo method for the log record is called. If the record does contain a page ID, the page is inspected, and if the page's LSN is older than the LSN of the log record, the record's undo method is called.

   Formatting of pages presents a special case. Page formats can be handled in one of two ways:
   1. Trust the LSN on the page to be correct. This works only if every page on the volume is initialized when the volume is formatted. For large volumes, this can take a long time, but it means that during recovery, the `page_format` log records can be skipped if the LSN on the page is older than the LSN of the log record. This is the default.
   2. Don't initialize each page when formatting a volume. The LSN on the page cannot be trusted in this case, and every `page_format` log record is redone during recovery; this means that every subsequent log record for that page must be redone. The storage manager can be configured to work this way by `#define`ing `DONT_TRUST_PAGE_LSN` in the configuration file `shore.def.`

3. UNDO: After all log records have been inspected and applied if necessary, the state of the database matches that at the time of the crash. Now the storage manager rolls back the transactions that remain active. Care is taken to undo the log records in reverse chronological order, rather than allowing several transactions to roll back at their own paces. This is necessary because some operations use page-fixing to for concurrency-control (pages are protected only with latches if there is no page lock in the lock hierarchy -- this occurs when logical logging and high-concurrency locking are used, in the B-trees, for example. A crash in the middle of a compensated action such as a page split must result in the split being undone before any other operations on the tree are undone.). After the storage manager has recovered, there can be transactions left in *prepared* state. The server is now free to resolve these transactions by communicating with its coordinator. (See Distributed Transactions, Two-phase commit, Internally-coordinated transactions, below.)

Locks are not acquired during recovery.

```
Files: restart.cpp
```

7.8.5 Log Buffering and I/O

The log manager can be configured at compile-time to perform local I/O or use remote processes for non-blocking I/O. Log records are buffered until forced to stable storage to reduce I/O costs. The log manager keeps a buffer of a size that is determined by a run-time configuration option. The buffer is flushed to stable storage when necessary. The last log in the buffer is always a *skip* log record, which indicates the end of the log partition.

```
Files: log_buf.h, log_buf.cpp
```

7.8.6 Controlling logging overhead

Logging can be turned on and off on the basis of

- store property,
- a storage structure's protocol, or
- a system-wide configuration option, although this is of limited use.

A store's property greatly affects the amount of logging that is required for updates to the store or to the pages in a store. Temporary files, for example, do not need to have data updates logged; only metadata updates are logged.

7.8.7 Turning off logging

Each storage structure has its own protocol for controlling logging. The B-tree code, for example, turns off logging for short periods to avoid physical logging of updates, then back on while it explicitly logs updates logically. The section Transactions and logging describes the mechanism.

Updates to the user data in temporary files never get logged.

There is a run-time configuration option `sm_logging`, which was added long ago in the hope of making it possible to measure logging overhead by forcing all logging to be turned off for an experiment. This must be used with care, if at all, because it might break some assumptions in the code.

7.8.8 Measuring logging overhead

The best way to measure logging overhead is to look at the statistics kept by the log manager. The following statistics are apropos:

- log_sync_nrec_max: the maximum number of log records collected in the log buffer before the log was flushed to stable storage.
- log_sync_nbytes_max: the maximum number of bytes collected in the log buffer before the log was forced to stable storage. These two help measure the effectiveness of buffering the log.
- log_sync_cnt: the number of times the log was forced to stable storage.
- log_chkpt_cnt: the number of checkpoints taken
- log_switches: the number of times the log was turned off for a transaction
- await_1thread_log: the number of times a thread of a multi-threaded transaction had to wait for access to the transaction manager.
- acquire_1thread_log: the number of times the mutex protecting the transaction manager class was acquired.
- get_logbuf: the number of calls to `xct_t::get_logbuf()`
- anchors: number of log anchors grabbed
- log_records_generated: number of log records written

- log_bytes_generated: number of bytes written to the log
- await_log_monitor: times a thread had to wait for access to the log manager class for certain operations
- await_log_monitor_var: times a thread had to wait for access to the log manager class variables

The log manager is a performance bottleneck for loads that involve many updates.

7.8.9 Log record types

There are three styles of logging described in the literature: *physical* logging (the record contains before- and after-images of the data), *logical* logging (the record describes the operation rather than its effect, e.g., "add 3 to the value in the 4th slot), and *physiological* logging (a combination of the two, which has the space-savings and allows high concurrency locking protocols offered by logical logging wherever possible, and it has the simplicity of physical logging where logical logging is not used).

The storage manager uses physiological logging, but the language used in the code and comments can be misleading, in that much of what is called "physical" logging is in fact logical by the commonly-accepted definitions above. Very few operations are, in fact, physically logged (`page_link, page_set_byte, page_splice, set_store_flags, set_deleting).` In the storage manager argot, a log record is "physical" if it is *page-oriented* (it applies to a page whose page ID is in the record), and "logical" it is *operation-oriented* (also called *functional* logging). For example, a B-tree insert results in a log record that is physically redone (insert this value on this page) and logically undone (remove this value from the B-tree; start by traversing the B-tree from the root).

Much of the log-record-specific code is generated by a Perl script. This makes it easy to change the logging scheme. The input file is `logdef.dat`, which contains a set of lines of the form

name mask args

for which the Perl script generates the following:

A class definition (in `logdef_gen.cpp`):

```
class name_log: public logrec_t {
      ...
public:
      name_log ( args );
      ...
      void redo ( page_p * ); // optional
      void undo ( page_p * ); // optional

};
```

A free function (in `logstub_gen.cpp`}:

```
rc_t log_name ( args ) { ... }
```

A case of a switch statement (in `redo_gen.cpp`):

```
case t_name:
      (name_log *) this) -> redo(page);
```

A case of a switch statement (in `undo_gen.cpp`):

```
case t_name:
      ((name_log *) this) -> undo(page);
```

A value for an enumeration (in `logtype_gen.h`):

```
t_name = ...,
```

The bodies of the methods of class `name_log` are hand-written and reside in `logrec.cpp`. The switch cases are included in general redo and undo function in `logrec.cpp.` Adding a new log record type consists of adding a line to `logdef.dat`, adding the method definitions to `logrec.cpp`, and adding the appropriate calls to the free function `log_name(args)` in the storage manager.

The base class for every log record is `logrec_t`, which is worth studying. It is in `logrec.h`. Every log record has one or more of the following characteristics (See the comments in `logdef.dat,` where some of these characteristics are set with the *mask*.)

- X: it is generated by a transaction (some log records are not, e.g., those used for checkpointing). If it is generated by a transaction, the log record will contain a transaction ID, and the `log_`*`name`*`()` function will be generated by the Perl script.
- R: it is redoable -- the general redo function will contain a switch case for this log record.
- U: it is undoable -- the general undo function will contain a switch case for this log record. An undoable log record is either page-oriented or logical (operation-oriented).
- A: it is a log record for space allocation, so its generation cannot be turned off by log switches or by store flags on a page.
- L: it is a logical log record -- it does not need a page pinned for undo.
- C: It is a compensation record. There is a log record whose sole purpose is to compensate around other log records, but compensations can be piggy-backed on other log records.

Certain methods of `logrec_t` refer to these characteristics:

- `bool logrec_t::is_redo() const` returns true if the log record is redoable
- `bool logrec_t::is_undo() const` returns true if the log record is undoable
- `bool logrec_t::is_cpsn() const` returns true for compensation records. These log records are not undoable. Instead of applying undo and then reading the previous log record for the transaction, the undo code skips this log record and simply goes on to look at the previous log record (`logrec_t::prev()`.
- `bool logrec_t::is_page_update() const` returns true if the log record is redoable, is not a compensation log record, and has a legitimate page ID to which to apply changes.
- `bool logrec_t::is_logical() const` returns true for logical log records.

```
Files: log.h, log.cpp, log_base.cpp, log_buf.h, log_buf.cpp, unix_log.h,
unix_log.cpp, srv_log.h, srv_log.cpp, raw_log.h, raw_log.cpp,
logdef.dat, logrec.h, logrec.cpp, logstub.cpp
```

7.9 External sort

The storage manager contains a set of sorting routines for use by the server. They are included in the storage manager for performance. File sort could be written at the server level, but it would require many record pins through the `pin_i` interface, which is costly. There are two sort implementations with two APIs for sorting files. One (the "old") sort implementation sorts on a fixed set of keys. The other (the "new") implementation is more general, and does much of its work through callbacks to the server-provided functions. This makes the storage manager prone to crashes if the callback functions contain any bugs. If the generality of the latter implementation is not needed, use the former.

The new sort implementation produces output directly usable for bulk-loading R-tree and B-tree indexes. It performs a polyphase mege sort with callbacks for key comparisons. If the type of the key is one of the fundamental types supported by the old sort implementation, the internal key comparison functions from the old sort implementation are used. For performance comparisons, the old sort can be made to invoke the new sort implementation. A Boolean argument to the old sort API controls this option.:

There are some significant differences that are worth mentioning here. The B-tree implementation sorts on key-value pairs, so in the event that a B-tree holds <key,OID> pairs, the records with duplicate keys must be sorted on the OIDs. The B-tree bulk-load method contains a hack to sort duplicates on their OIDs, so that bulk-loading can work with the old sort code. The new sort can be made to guarantee a stable sort, but in the event that the sort is producing records for bulk-loading a B-tree, this can be counter-productive. The new sort code handles duplicates as follows:

```
if (duplicates are being eliminated) {
  remove the one with the larger OID
} else if (this is a stable sort) {
  sort by input order
} else if (this is output for an index) {
   sort by OID
} else {
   don't guarantee any thing about their relative order
}
```

The old sort eliminates duplicates as follows:

```
if (duplicates are being eliminated) {
   if (entire object is a duplicate) {
       remove one of the objects
   }
}
```

7.10 Statistics

The storage manager keeps server-wide statistics. It also keeps some per-transaction- and per-thread statistics. Per-thread statistics can be aggregated into the per-transaction statistics, and per-transaction statistics can be reaped by the server at the end of a transaction. Some of the data structures and code for managing counter-type statistics is generated from Perl scripts and `.dat` files. To add a counter to the storage manager, simply add a line to the (mostly self-explanatory) `sm_stats_info.dat` and add code to update the counter in the appropriate places in the storage manager. The macros INC_TSTAT and INC_STAT update statistics. Grep for the definitions of these macros in `sm/*.{h,cpp}` to see how they are used.

7.11 Distributed Transactions

The support for distributed transactions is described separately from the transaction manager because it is separable from the rest of the storage manager, for single-server systems (single-process servers).

The server library contains a set of classes that support two-phase commit and centralized global deadlock detection. These classes use the communications package for inter-process communication. Because the communications package is not included with the storage manager, the details for these two modules are omitted.

7.12 Two-phase commit

The storage manager contains hooks for two different flavors of distributed transactions.

7.12.1 Internally-coordinated transactions

When two or more SHORE storage manager-based server processes cooperate to implement distributed transactions, the coordinator in the storage manager library can be used. All over-the-wire communication is performed with the communications system described above. The coordinator implements the presumed-abort commit protocol. The coordinator's log is a B+-tree.

```
Files: coord.cpp, coord.h, coord_log.cpp, coord_log.h, coord_thread.cpp,
subord.cpp, participant.cpp, participant2.cpp
```

7.12.2 Externally-coordinated transactions

When a (one or more) server(s) cooperates with one or more external entities to implement distributed transactions, the server must use an external coordinator, and all interaction with the storage manager is through the storage manager's API (no over-the-wire interaction with external coordinators). The storage manager library contains a coordinator that uses the communications package (not included). The coordinator is a class that is meant to be instantiated by a server after (local) recovery is finished. If recovery yields any prepared transactions that the server considers to be in-doubt, the server is expected to initiate global recovery before accepting any more work from clients. The tester shell (`ssh`) contains code that handles recovery for testing purposes; it can be examined to see how recovery is done.

7.13 Centralized global deadlock detection

The storage manager library contains a module that performs deadlock detection on global and local transaction. This module interacts directly with the lock manager.

The lock manager, after waiting a (configurable) period of time for a lock, invokes the global deadlock detection class (which may be replaced, for example, with timeout). The deadlock detector provided with the library collects the waits-for edges, filtering out unnecessary local-only edges to reduce traffic, and detects cycles. The server is invoked through a callback to choose a victim if a cycle is detected. The policy regarding choice of a victim is thereby left with the server.

Inter-server communication uses the communication module described above.

The implementation of global deadlock detection separates policy from mechanism.
The policy of selecting a victim in the event of a deadlock is implemented by the server in a callback function. Certain events can be monitored by the server through callbacks:

1. a global deadlock is detected
2. a local deadlock is detected
3. a global victim was selected
4. the storage manager is about to kill the victim selected

The cooperating servers use global deadlock detection through a few methods in the storage manager and by instantiating several classes. Each server's lock manager, upon finding that a transaction (thread) must block awaiting a lock, performs local deadlock detection, then, if necessary, global deadlock detection through a hook to a `class GlobalDeadlockClient`. In this context, all cooperating servers are clients of a deadlock detection service. The only such service implemented in the storage manager is a centralized service (`class CentralizedGlobalDeadlockClient: public GlobalDeadlockClient`, along with `class CentralizedGlobalDeadlockServer`), although one could extend the storage manager, implementing a distributed service by writing a `class DistributedGlobalDeadlockClient.` Any one of the cooperating processes must be running an instance of the `CentralizedGlobalDeadlockServer.` This class has a thread that listens for requests on a well-known `Endpoint.` It is a well-known endpoint in the sense that the cooperating servers, when they start up, create instances of `CentralizedGlobalDeadlockClient`, giving these classes the well-known endpoint. When these instances are initialized, they contact the deadlock server and, with a handshake, identify themselves as deadlock clients, in return getting identifiers assigned by the deadlock server.

The deadlock client and deadlock server classes use helper classes that implement the protocol among them. These helper classes allow the mechanism of communication to be separated from the deadlock detection algorithm (centralized or distributed) to some extent. The helper classes are `DeadlockServerCommunicator` and `DeadlockClientCommunicator`. These two classes implement the following protocol elements:

- `msgRequestClientId`: a client sends this message to the server, which response with a

- `msgAssignClientId`: the server assigns an identifier (a bit in a bit mask) to each active client, and responds with the identifier.
- `msgVictimizerEndpoint`: sent by one client or server to a server to indicate a remote entity that will receive `msgSelectVictim` requests.
- `msgRequestDeadlockCheck`: a client sends this to a server when the client's lock manager causes a transaction thread to block.
- `msgRequestWaitFors`: sent by a server to a client whose waits-for graph the server wants
- `msgWaitForList`: the client's response to the above request
- `msgSelectVictim`: sent by a server to whatever cooperating process is responsible for selecting a victim once a deadlock is detected. This message contains a list of global transaction IDs representing the transactions deadlocked.
- `msgVictimSelected`: response to the above request
- `msgKillGtid`: once a victim is selected, the server sends a message to the client that is running the victim transaction thread
- `msgQuit`: sent from any cooperating process to another to cause the entire set of classes to shut down.
- `msgClientEndpointDied`, `msgVictimizerEndpointDied`, `msgServerEndpointDied`: send by the communications package when any of the endpoints detects a network or server failure.

The centralized server does not use the `msgVictimizerEndpoint`, `msgSelectVictim` and `msgVictimSelected` messages. Instead, it asks its own server to select victims through the callback function mentioned above. If the centralized server is not given a callback function in its constructor, a default policy is used. The default policy is to choose the first transaction in the list of transactions involved in a deadlock.

The centralized server awaits `msgRequestDeadlockCheck` requests, and responds by broadcasting requests for waits-for graphs. It uses its bit mask of clients to determine when it has received a response (waits-for graph) for each of the active clients; at that time it checks for deadlocks, and, if so, selects a victim and issues a `msgKillGtid` to the right client.
The cycles of broadcasting and collecting waits-for graphs are serialized. Several requests are satisfied with a single cycle.

```
Files: sm_global_deadlock.h, sm_global_deadlock.cpp, deadlock_events.h,
deadlock_events.cpp
```

7.14 Callback Classes

In order to simplify testing and to allow servers to manage policy regarding distributed transactions, the storage manager defines interfaces for two simple mapping classes, which the server is expected to implement:

- Global transaction ids ---> local transaction ids
- Server name <---> communication endpoint

Global transaction IDs and server names are treated as opaque data by the storage manager.

8 Testing the storage manager - sm/ssh

Unit-testing of the storage manager is performed with a server that interprets Tcl scripts. The interpreter contains functions for each element of the storage manager's API.
In addition, there are some shell scripts that run and restart the Tcl-based server, invoking hooks in the server to cause controlled crashes.

Altogether, these tests include single- and multi- transaction tests, multi-threaded-transaction tests, distributed transaction tests, and a few tests with pseudo-randomness.

9 References

[1] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publishers, Inc., 1993

[2] C. Mohan, D. Haderle, Bruce Lindsay, H. Pirahesh, P. Schwarz, *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Wriate-Ahead Logging*, , ACM Transactions on Database Systems, Vol 17, Number 1, March 1992

[3] C. Mohan, *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*, 1989, Research Report RJ 7008, Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120; also published in abbreviated form in VLDB 1990, Brisbane, Queensland, AU, pp 392-405

[4] C. Mohan, Frank Levine, ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging, 1989, Research Report RJ 6846, Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120

[5] A. Silberschatz, P. Galvin, Operating Systems Concepts, Addison-Wesley, Addison-Wesley Longman, Inc., 1998

[6] A. Silberschatz, J. Peterson, Operating Systems Concetps, Addison-Wesley Publishing Company, 1988

[7] C. Mohan, D. Haderle, *Algorithms for Flexible Space Management in Transaction Systems: Supporting Fine-Granularity Locking*, International Conference on Extending Database Technology (EDBT) 1994, Cambridge, UK

[8] N. Beckmann, H-P Kiegel, *The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles*, Proceedings of the 1990 SIGMOD International Conference on Management of Data, Atlantic City, NJ, Mar 23-25, 1990, pp 322-331