An ARIES Log Manager for *Minirel*

CS 764 – Project Report

Ajitkumar Natarajan (`ajitk`) Cjin Pheow Lee (`cjin`)

May 13, 1995

Abstract

Log management is an integral part of the recovery subsystem of any RDBMS. An ARIES based log manager for a simple, multi-user RDBMS – *Minirel* – has been designed and implemented. The interface consists of four classes: `lsn_t`, `logrecord`, `shm_log`, and `log`. The implementation is based on a one page log tail maintained in shared memory. Log reads and writes are completely mutually excluded from one another. Extensive testing of the implementation has been done. Limitations of the log manager include low concurrency, simplistic crash assumptions, and use of a single Unix file to maintain the log.

1 Introduction

The recovery subsystem is an essential part of any Relational Database Management System (RDBMS). Functions that are performed by the recovery subsystem include:

1. Logging of all updates done to the database. Either physical changes to the database may be logged (*physical logging*, e.g., the bytes in the offset range 20–24 in page 3 of the database changed from $\langle 20, 30, 40, 50, 60\rangle$ to $\langle 75, 32, 47, 95, 126\rangle$) or the operations performed on the data may be logged (*logical logging*, e.g., an insert of the $\langle$*key*, *record-id*$\rangle$ pair $\langle 25, 65\rangle$ was done on the B^+-tree index on employee ID of relation foo of the database).

2. Supporting the notion of transaction *aborts* or *rollbacks*. Every change that the transaction effected on the database must be undone. The log of changes made by the transaction to the database (maintained, as described in (1) above) is used for this purpose.

3. Crash recovery. The data in the database must be protected against crashes. Modern RDBMSs use a buffer pool to cache the most often/recently used pages of the database. If the system crashes before the changes made in main memory make it to disk, the recovery subsystem, on restart, will have to reconstruct the changes. Again, the log is consulted to determine the changes effected. Also, transactions that were in progress at the time of the crash need to be undone.

4. Checkpointing and Savepointing. Checkpointing takes a snapshot of certain key data structures and stores it in the log. This checkpoint is used during restart to minimize the amount of work the system needs to do before everything is back to normal. Savepointing allows transactions to snapshot themselves at various points within themselves, and rollback to any previously made savepoint.

It is clear that logging is an integral part of the recovery subsystem. This project report describes the design and implementation of a log manager for a recovery subsystem based on ARIES [Mohan et al. 92]. The recovery subsystem is a part of a simple, multi-user RDBMS called *Minirel*.

Section 2 briefly describes how ARIES recovery works. This section also introduces the features expected of a log manager based on ARIES. Section 3 describes *Minirel* and how an ARIES-based log manager fits into it. Section 4 defines the interface that our log manager provides to other parts of *Minirel*. Section 5 outlines how the log manager implements the interface and other implementation issues. Sections 7 and 8 enumerate some additional work that can be done to enhance the utility and performance of our log manager and *Minirel* as a whole, respectively. Finally, Section 9 concludes the report.

2 ARIES Recovery

ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) recovery is based on the Write Ahead Logging (WAL) protocol. Every update operation writes a log record which is one of

1. An undo-only log record: Only the before image is logged. Thus, an undo operation can be done to retrieve the old data.
2. A redo-only log record: Only the after image is logged. Thus, a redo operation can be attempted.
3. An undo-redo log record. Both before image and after images are logged.

Every log record is assigned a unique and monotonically increasing log sequence number (LSN). Every data page has a page LSN field that is set to the LSN of the log record corresponding to the last update on the page. WAL requires that the log record corresponding to an update make it to stable storage[1] before the data page corresponding to that update is written to disk. For performance reasons, each log write is not immediately forced to disk. A log tail is maintained in main memory to buffer log writes. The log tail is flushed to disk when it gets full. A transaction cannot be declared committed until the commit log record makes it to disk.

Once in a while the recovery subsystem writes a checkpoint record to the log. The checkpoint record contains the transaction table (which gives the list of active transactions) and the dirty page table (the list of data pages in the buffer pool that have not yet made it to disk). A master log record is maintained separately, in stable storage, to store the LSN of the latest checkpoint record that made it to disk. On restart, the recovery subsystem reads the master log record to find the checkpoint's LSN, reads the checkpoint record, and starts recovery from there on.

The actual recovery process consists of three passes:

1. Analysis. The recovery subsystem determines the earliest log record from which the next pass must start. It also scans the log forward from the checkpoint record to construct a snapshot of what the system looked like at the instant of the crash.
2. Redo. Starting at the earliest LSN determined in pass (1) above, the log is read forward and each update redone.
3. Undo. The log is scanned backward and updates corresponding to loser transactions are undone.

For further details of the recovery process, see [Mohan et al. 92, Ramamurthy & Tsoi 95].

It is clear from this description of ARIES that the following features are required for a log manager:

1. Ability to write log records. The log manager should maintain a log tail in main memory and write log records to it. The log tail should be written to stable storage on demand or when the log tail gets full. Implicit in this

[1] By stable storage, we mean a duplex-disk setup which is safe against single disk failures.

requirement is the fact that the log tail can become full halfway through the writing of a log record. It also means that a log record can be longer than a page[2].

2. Ability to wraparound. The log is typically maintained on a separate disk. When the log reaches the end of the disk, it is wrapped around back to the beginning.

3. Ability to store and retrieve the master log record. The master log record is stored separately in stable storage, possibly on a different duplex-disk.

4. Ability to read log records given an LSN. Also, the ability to scan the log forward from a given LSN to the end of log. Implicit in this requirement is that the log manager should be able to detect the end of the log and distinguish the end of the log from a valid log record's beginning.

5. Ability to create a log. In actual practice, this will require setting up a duplex-disk for the log, a duplex-disk for the master log record, and a raw device interface to read and write the disks bypassing the Operating System.

6. Ability to maintain the log tail. This requires some sort of shared memory because the log tail is common to all transactions accessing the database the log corresponds to. Mutual exclusion of log writes and reads have to be taken care of.

The following sections describe some simplifying assumptions that we have made to fit the protocol into *Minirel* and the interface and implementation of our log manager.

3 The *Minirel* RDBMS

Minirel is a simple, multi-user RDBMS. It has all the features of a regular RDBMS, including concurrency control and locking, logging and recovery, buffer, heap file, and space management, access methods (B^+-trees, hashing, grid files, etc.), relational operators, query planning, and user interface components. The log manager fits into the system as follows:

- Every update operation will log its update via the recovery manager into the log tail. When the log tail is full or when there is a flush request, the log tail is flushed to disk.
- WAL is enforced by informing the buffer manager whenever the log tail is flushed. The buffer manager can also call the log manager to flush the log tail when it wants to flush a page whose page LSN is more than the last flushed LSN the buffer manager knows about.
- The recovery manager periodically writes checkpoint records into the log and calls the log manager to update the master log record.
- On restart, the recovery manager scans the log and performs restart recovery as described in Section 2.

Thus, the log manager mainly interacts with the recovery manager. Every other component (except the buffer manager) sees the log manager only through the recovery manager. The only piece of the log manager that the buffer manager sees is the enforcement of the WAL protocol.

Minirel is organized as follows: every *Minirel* process runs one transaction and accesses one database throughout its existence. There is one log corresponding to each database. (For simplicity, the log is maintained as a Unix file.

[2] The log consists of a set of pages. Reads and writes to the log are done in units of a page. The reason for this is that logs are typically maintained in a separate disk and this disk can be read/written only in units of a page, i.e., disk block.

Duplex-disks and log disks are not used.) Every *Minirel* process has the entire code for all components. However, the data portion (the buffer pool, the log tail, the lock and transaction tables, etc.) are kept in shared memory. The first transaction to use a database initializes all the shared data structures. Currently, a *Minirel* process uses only one database and its corresponding log. Thus, the shared memory data structures correspond to a common database.

We have made the following simplifications:

- No duplex-disks or raw device manipulation. We maintain the log as a single Unix file.
- A log tail of size one page. This considerably simplifies the implementation. Unfortunately, performance is not as good as it would have been with double buffering or multiple buffers.
- Complete ignorance of log record types. All that the log manager sees is a sequence of bytes. The recovery manager interprets the contents of a log record.
- No transaction "crashes." If regular crashes were allowed and a transaction was in a critical section when it crashed (e.g., updating the log tail), no other transaction can enter its critical section. Thus, every transaction exits cleanly. However, cleanliness is restricted to unlocking shared data structures and exiting critical sections.

The log management component of a real database system would relax these assumptions and be more rigorous about duplexing and crashes.

4 Log Manager Interface

This section describes the interface that the log manager provides to other parts of *Minirel*. Most of the interface is used by the recovery subsystem only.

4.1 Log Sequence Numbers

Every log record is assigned a unique Log Sequence Number (LSN). There must be a way to create unique LSNs. Also, the recovery subsystem will supply an LSN and expect the log manager to read the corresponding log record. Thus, LSNs must encode some notion of position within the log.

The functionality for creating and using LSNs is provided by `class lsn_t` (header file: `lsn.h`). The default constructor of this class initializes the LSN to an invalid LSN. An invalid LSN is one that can be uniquely detected and does not denote any valid position in the log. It is very useful for error returns. For ease of use, `lsn.h` declares a constant LSN called `invalid_lsn` that can be used wherever an invalid LSN is required.

`Class lsn_t` provides methods for setting and retrieving the individual parts of an LSN. The methods for setting the LSN are `private` so that unintentional modification of the LSN by external code is prevented. `Operator` << has been overloaded to recognize LSNs and output them in a human-readable form.

The recovery subsystem needs a way to compare LSNs for equality. It also needs to be able to decide whether the page LSN field of a page is less than the log record's LSN to determine whether to redo an update. `Class lsn_t` overloads the operators `==`, `!=`, <, >, <=, and >= to handle this. While invalid LSNs can be compared against other LSNs for equality and inequality, they cannot be meaningfully compared using any of the other comparison operators.

`Class log` is a `friend` of `class lsn_t` so that it can manipulate LSNs in anyway it wants. The log manager sets the LSN for every log record written and hence, needs to be able to access the methods that set the various parts of an LSN.

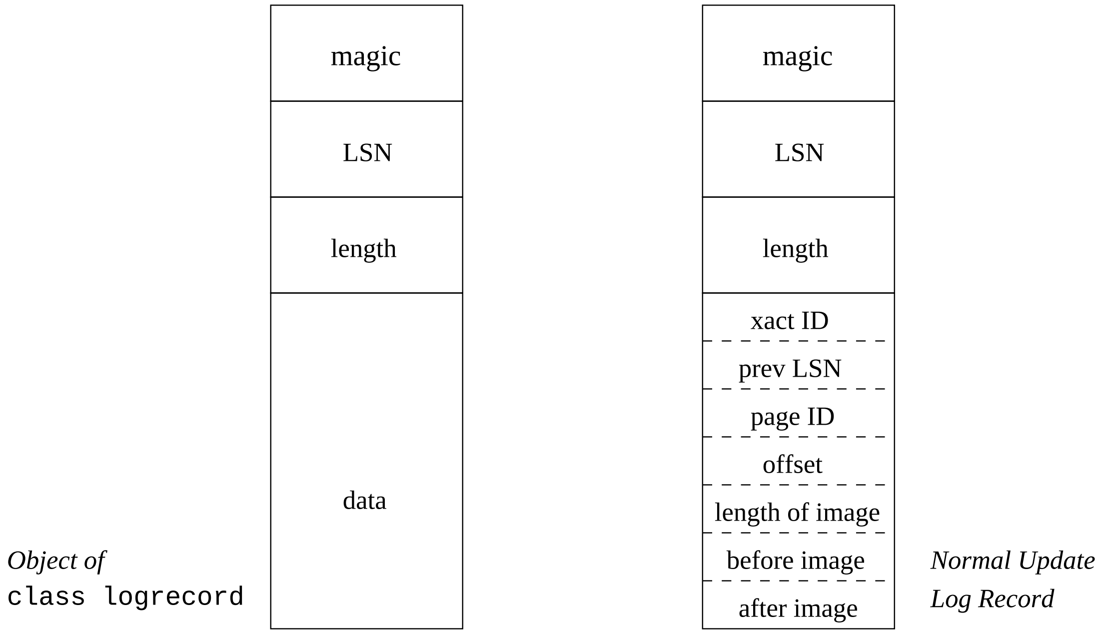

Figure 1: Log records. The figure at the left shows an object of the base `class logrecord`. The one at the right shows an object of a class derived from `class logrecord`. The derived class object shown corresponds to a normal update log record. For exact representation and details, see [Ramamurthy & Tsoi 95].

4.2 Log records

The log consists of a sequence of log records. Each log record has a fixed header portion and a variable data portion. `Class logrecord` (header file: `logrecord.h`) declares the basic log record. The fixed portion of the log record contains the following fields:

1. A magic number. This is used (as described in Section 5) to distinguish between the end of the log and the beginning of a new log record.

2. The LSN of the log record. This need not necessarily be stored in the log record. However, we store it in the log record to make our implementation easier. Also, whenever the log record is output during debugging, the LSN can be output.

3. The length of the log record. This is the length in bytes from the magic number field to the end of the data portion of the log record. However, users of `class logrecord` will only see (and set) the length of the data portion. The implementation will take care of setting this field correctly. This has been done because users of the class do not set or retrieve the fixed header portion of the log record.

The data portion of the log record is a sequence of bytes. `Class logrecord` does not interpret this data in any way. However, the recovery subsystem derives from this class to define several different kinds of log records [normal updates, checkpoint records, commit log records, Compensation Log Records (CLRs), etc.]. Thus, it is the recovery manager that interprets the contents of the data portion of each log record. Derivation ensures that the fixed portion of the log record (which is interpreted by `class logrecord`) is available in any kind of log record. Figure 1 shows an object of `class logrecord` and a normal update log record.

`Class logrecord` provides methods to get and set the data length. It has a method to return a pointer to the data portion of the log record. It provides read access to the LSN of the log record and write access to `friends`. Finally, it overloads `operator <<` to output the log record in a human-readable format. The data portion of the log record is dumped in hex and ASCII.

4.3 Shared Memory Structure

The *Minirel* system consists of several processes, each trying to access and update a single database. Thus, some form of sharing is required between processes to access common data structures. One example of a shared data structure is the log tail. This section describes the data structure maintained by the log manager in shared memory.

The data structure maintained in shared memory is an object of `class shm_log` (header file: `logshm.h`). It contains the following fields:

1. The log tail. This is used to write log records into the log. When the log tail is full, it is flushed to disk.
2. The first LSN. This is the LSN of the earliest log record that the recovery subsystem is interested in. Log records with LSNs before this will never be used for a future restart.
3. The next LSN[3]. This is the LSN that will be assigned to the next log record that is written.
4. The log file header. This contains the master log record, the maximum size of the log (in pages), and the log tail size (in pages). Currently, the log tail size is fixed at 1 page.
5. The number of transactions that are currently sharing this shared memory structure.
6. A semaphore to mutually exclude log opens from one another.

Only `class log` accesses the shared memory structure. However, its definition must be made public so that the OS infrastructure group can declare a shared memory segment of suitable size and assign the global pointer `global_shm_log_ptr` to the correct address.

4.4 The Log Manager

The log manager is defined in `class log`. Every *Minirel* process will instantiate a global object of this class. `Class log` includes methods to

1. Read a log record given its LSN.
2. Read the log record that immediately follows the log record with the supplied LSN.
3. Read the log record that immediately follows the supplied log record.
4. Append a log record to the end of the log.
5. Flush the log tail to disk. Anyone who wants the log tail flushed (except the buffer manager) will call `log::flush`. The buffer manager [Qiu & Su 95] will call `log::bufmgr_flush`. When `log::flush` is invoked, it will flush the log tail and inform the buffer manager of the last LSN that made it to disk. The buffer manager can use this information to enforce WAL efficiently. If the buffer manager wishes the log tail to be flushed, it calls `log::bufmgr_flush` which will flush the log tail and return the last LSN that made it.

[3] Throughout this report, the terms next LSN and last LSN are used interchangeably.

wrap count	page ID	offset

Figure 2: Layout of an LSN. In the current implementation, each field is four bytes long.

6. Return the LSN that will be assigned to the next log record written.
7. Read/write access to the master log record.

`Class log` is also used to create and initialize new log files.

5 Implementation

The following sections describe the implementation details of the classes described in the previous section.

5.1 `class lsn_t`

Every LSN is a three-tuple: $\langle$wrap count, page ID, offset$\rangle$. Figure 2 shows the layout.

The wrap count is needed because the log is maintained in a bounded space and wraps around when it reaches the physical end of the space. The wrap count field gives the number of times the log has wrapped around. The current implementation stores the wrap count as an `unsigned int`. It is an error for the wrap count to exceed `UINT_MAX` -1.

The page ID and offset fields are required because the log can be read and written only in units of a page. The page ID and offset give the location of the start of the log record within the log.

Invalid LSNs are given the value: $\langle$`UINT_MAX` -1, `INT_MAX` -1, `UINT_MAX` $-1\rangle$. The implementation ensures that LSNs never reach this value.

Implementation of the methods of `class lsn_t` (see `lsn.h` and `lsn.C`) is straightforward and will not be described.

5.2 `class logrecord`

`Class logrecord` declares a buffer which is an array of characters of size $3*$`PAGESIZE`. `PAGESIZE`, the size of a page, is declared in the header file `page.h`. A normal update can log a modification to an entire page of the database. Thus, the sizes of the before and after images can each be one page. Header information in each log record will add another fraction of a page. Thus, a maximum size of 3 pages suffices. Checkpoint records, however, could potentially be very large. In such situations, the log record can be broken into several pieces, each piece of size less than 3 pages.

The first part of the buffer contains the log record header (described in Section 4.2). The remaining portion is the data. Thus, the entire log record will be stored physically contiguously in the buffer. The `logrechead_ptr` and `data_ptr` members point to the start of the buffer and start + `sizeof(logrecheader)` respectively.

Implementation of the methods of this class also (see `logrecord.h` and `logrecord.C`) is straightforward and will not be described.

5.3 `class shm_log`

The shared memory structure `class shm_log` has a constructor that will be called by the OS infrastructure code for the first *Minirel* process. This constructor must initialize the shared memory structure. In our case, it has to set the number of transactions field to zero. It must also call the constructor for the semaphore member `sl_sem`.

The class has no other methods because manipulation of the members is done by `class log`.

5.4 `class log`

Most of the functionality of the log manager is implemented in `class log` (file: `log.C`). The following sections describe various aspects of the implementation of `class log`.

5.4.1 `class Rawfile`

At the lowest level, the log is maintained in a Unix file. However, to emulate the way a real log is read/written, access to this file should be in units of a page. We also need the forced write semantics – when a write succeeds, it should be guaranteed that the changes have made it to disk. Further, changing over to a duplex-disk setup or to a raw disk interface should change the least amount of code.

`Class DB` [Murthy & Narasimhan 95] was a candidate for this low level file maintenance. It provided pagefile semantics and provided a forced-write semantics. However, it also managed a space map and allowed multiple page files to reside within the `DB`. This functionality was needed because `DB` was used to store the database itself. We did not need this extra functionality. Besides, it was not natural to treat a log as a `DB`. Therefore, we implemented `Rawfile` (see `rawfile.h` and `rawfile.C`), which provides a page file view of a Unix file.

`Rawfile` implements bounded file semantics as follows: when the log is created, the number of pages it contains is specified. `Rawfile::Rawfile` (the create constructor) allocates the required space by `lseek`ing [see `lseek`(2)] to the end of the file and `write`-ing [see `write`(2)] a single byte there. This has the advantage that the log file does not occupy disk space until actually written to. In addition, we need not keep track of the size of the file. We can find it any time by using the `stat`(2) or `fstat`(2) system calls. `Rawfile::write_page` never allows a write to go beyond the current size of the file.

`Rawfile::read_page` and `Rawfile::write_page` are straightforward. They use the `read`(2) and `write`(2) system calls to read and write a page at a time. `Rawfile::write_page` also forces the write to disk by calling the `fsync`(2) system call.

5.4.2 `log::log` (The Create Constructor)

The create constructor is invoked by a special utility which creates a database. This utility does not run transactions. Also, transactions do not use the database being created until the create utility successfully completes. Therefore, the create constructor does not need to provide the full functionality of an open (see Section 5.4.3). It does not create a shared memory structure or attach to it, and does not allow any of the `log` methods to be invoked. It simply creates a log file (a `Rawfile`) of the appropriate size and initializes it. Initialization involves writing the header page (in page 0) which contains a master log record, the size of the log file, and the size of the log tail. This is used in future opens to allocate and initialize the appropriate data structures. The size of the log tail is one page in the current implementation. The master log record contains the LSN of the last checkpoint record that made it to disk. For a new log, there is no checkpoint record. This field, therefore, is set to `invalid_lsn`.

The create constructor then closes the `Rawfile`. The log is now ready for future use.

5.4.3 `log::log` (The Open Constructor)

The open constructor will open the log file, attach to the shared memory structure, and initialize data members so that the log is operational. Each *Minirel* process will have an object of `class log` in private memory. The shared memory is accessed through `global_shm_log_ptr`. The following paragraphs describe the sequence of operations that the open constructor goes through.

The open constructor must mutually exclude other opens from occurring concurrently. If the opens of the first two transactions access the shared memory structures without mutual exclusion, each open may think that it was the first one and a race condition will occur. To prevent this, the open constructor `downs` the `sl_sem` semaphore before attempting anything else. Thus, `sl_sem` is used to mutually exclude opens from each other.

However, it may be argued that the internal semaphore that each shared memory segment has (by virtue of deriving from `class SharedRegion`) can be used for mutual exclusion. This semaphore is used by all methods to mutually exclude access to the shared memory structure. Unfortunately, this will not work for the open constructors of the first set of transactions if they all start their opens more or less simultaneously. Consider the following code fragment:

```
global_shm_log_ptr->lock();           //  down the semaphore
if (global_shm_log_ptr->sl_n_xact > 0) {
    global_shm_log_ptr->sl_n_xact++;
    global_shm_log_ptr->unlock();
    return;
}
global_shm_log_ptr->sl_n_xact = 1;

//
//  We are the first transaction.  Initialize shared memory
//  structures.  Fill in the information by reading the header
//  page from the log file.  Call recovery manager to perform
//  restart.
//
...
```

This segment will not work because the recovery manager will attempt to read the log. `Log::read` will try to down the semaphore again and will block everyone forever. We have to unlock before calling recovery.

However, if we unlock before recovery, it could happen that some other transaction enters its open, detects that it is not the first, increments `sl_n_xact`, does not perform recovery, and starts reading and writing the log. Unfortunately, since recovery has not run yet, the log will be corrupted if another transaction attempts to read and write it.

It may appear that the second transaction went along corrupting the log because it saw that `sl_n_xact` was 1. However, setting `sl_n_xact` to 1 finally (instead of at the start) is not a correct solution. If this were done, the second transaction will also think that it is the first one in the system and perform restart recovery. There will be two restarts running concurrently. Since the recovery manager assumes that at most one copy of it is running in restart recovery mode at any time, this solution is incorrect.

The correct solution is that we must mutually exclude opens from each other. In that case, no other transaction can open the log while another transaction is attempting an open. Also, the first transaction gets to run restart recovery without the danger of another transaction doing the same. Further, since we do a `global_shm_log_ptr->unlock()` before restart recovery, the recovery manager can safely issue `log::read` and `log::write` calls. No other transaction will simultaneously be trying to execute `global_shm_log_ptr->lock()` because it is guaranteed that only the first transaction will run recovery. `Downing` `sl_sem` prevents other transactions from entering the system.

The following argument demonstrates that there is no deadlock:

- Every open will start by `down`ing the `sl_sem` semaphore. If at any time, it accesses a shared memory structure, it will also perform a `global_shm_log_ptr->lock()`. Thus, shared data structures are not corrupted.
- Other `log` methods never use `sl_sem`.
- Every open requests a `down` on `sl_sem` before a `global_shm_log_ptr->lock()`.

Thus, every open requests the `down`s in the same order and other methods never `down sl_sem`. Therefore, there can never be a deadlock.

Since every method `up`s semaphores that it `down`s and there is no deadlock, there is no possibility of starvation.

Initialization of shared memory structures, etc., is straightforward. It will not be described here.

There is one further problem that must be resolved. The `main()` function of a *Minirel* process instantiates a global object of `class log` by executing:

```
extern log *logMgr;
char       log_file_pathname[MAXPATHNAME];
Status     error_status;

logMgr = new log(log_file_pathname, error_status);
```

Therefore, `logMgr` has not been initialized when the open constructor is in progress. However, the open constructor, on detecting that it is the first transaction to use the database, will start restart recovery. Restart recovery will then read log records by invoking:

```
logMgr->read(...);
```

This will fail because `logMgr` has not been initialized as yet. The solution we adopted was to execute

```
logMgr = this;
```

in the open constructor code just before calling restart recovery.

5.4.4 `log::~log`

The destructor behaves differently depending on whether the open constructor or the create constructor was used. If the create constructor was used, nothing need be done. If the open constructor was used, the destructor checks to see if this transaction is the last one using the database. If so, it flushes the log tail. In any case, the log file is closed. The destructor determines the constructor that was used by checking the `open_called` flag. The open constructor sets this flag to `true`, the create constructor sets it to `false`. The destructor determines if this is the last transaction by examining `sl_n_xact`. This is admittedly a naive approach. If a transaction exited uncleanly without invoking its `log` destructor, this value will be wrong. This required us to assume that every transaction cleanly exits by releasing its locks and semaphores, and calling appropriate destructors.

5.4.5 `log::write`

The `write` method first checks if the log record supplied is beyond the maximum size allowed. If not, it executes `global_shm_log_ptr->lock()` because it extensively accesses and modifies the shared memory structure. It then computes the total number of pages that the log record would need[4] and the number of pages available in the log. The

[4] Whenever a log record is written, a `logrecord::LOG_END` is appended to it. This allows detection of end of log as described in Section 5.4.6. Therefore, the length of the log record used in this computation is one more than the length of the supplied log record.

current size of the log is defined as the number of pages between the first LSN (`global_shm_log_ptr->sl_firstlan`) and the current last LSN (`global_shm_log_ptr->sl_nextlsn`) inclusive. For normal log writes [those writes that do not correspond to transaction aborts (CLRs)] a check is made to verify that writing the log record will not cause the log size to exceed half its maximum size. For CLRs, a check if made against the maximum size of the log. That way, if a transaction decides to abort, there will be enough log space to write out all the CLRs.

If the check for log size fails, the recovery manager is called to compute a more optimistic value of first LSN and the check is repeated. If it fails this time, the log write is failed.

It is now known that the log record will fit into the log. Therefore, it is assigned an LSN. The LSN of the log record is the current value of `global_shm_log_ptr->sl_nextlsn`.

Normally, a small log record will fit into the log tail. This is the simplest case. The log record is copied into the log tail and the last LSN is updated. If the log record is too big to fit a page, then whatever fits is copied into the log tail, the log tail is flushed, and the remaining portion is written in a similar manner. Thus, the `write` method will work for log records that span many pages.

Unfortunately, the procedure outlined above is not foolproof. Suppose a log record spans more than a page. Assume that the first write and flush succeeded. Suppose that during the second write, a power failure occurred. Now the log is left inconsistent. During restart, the end of log will not be detected properly (see Section 5.4.6) and the `read` may return incorrect data. One way of detecting corruption is to end every log record with a magic number. If we do not detect the magic number at the end of the record read, we know for sure that the corresponding `write` failed midway. However, if we do detect the magic number, we still cannot be sure that there was no corruption. Hence, using this method of detecting corruption is not guaranteed to work.

When the entire log record is written, `write` executes `global_shm_log_ptr->unlock()` and returns. The log record that was supplied as a parameter to `write` will now contain the LSN field filled in.

5.4.6 `log::read`

`Read` is symmetric to `write`. It gets the page in which the start of the log record resides. It checks whether the log record starts with the magic number. If not, it checks the byte against `logrecord::LOG_END`. If this matches, an EOF is signaled and `global_shm_log_ptr->sl_nextlsn` is set to the LSN supplied. (Thus, a side effect of the analysis pass of restart recovery would be to determine the end of the log.) If this test also failed, the LSN does not point to the start of a log record. An error indication is returned.

The log record header is then examined. It may happen that the log record header is split across pages. In this case, the next page is read to get the remaining portion of the header. The header contains the length of the entire log record. This is used to determine the number of pages that must be read to get the whole log record. The appropriate number of pages are read to assemble the entire log record. If the physical end of the log is reached, `read` wraps back to the physical beginning of the log to continue the read.

Note that `write` always goes through the log tail to the log file whereas read never touches the log tail. This is required so that an arbitrary read does not corrupt the log tail. However, this poses a problem during restart recovery. The recovery manager will read log records till the end of the log is reached. New log records are then written. Unfortunately, if the end of the log occurred in the middle of a page, the log tail is not initialized with the contents of the page and, hence, the next flush of the log tail will corrupt the log. The open constructor handles this by reading the page of the log into the log tail when it detects end of log.

5.4.7 `log::read_next_logrec`

These methods use the `read` method to accomplish the actual read. The method which takes the LSN of the current record reads the log record at that LSN and supplies it to the other `read_next_logrec`. This method uses the length of the log record to compute the LSN of the next log record and reads that in.

It is obvious that it would be more efficient to call the second method if the current log record is available rather than getting its LSN and calling the first method.

5.4.8 `log::flush` and `log::bufmgr_flush`

The flush methods write the log tail to the corresponding page on disk. An issue is to compute the last LSN that made it to disk when this write succeeds. Since LSNs determine positions and LSNs can be compared using $<$, $>$, etc., we return the last LSN flushed as one less than `global_shm_log_ptr->sl_nextlsn`. This works because we are not interested in the LSN of the last log record that made it to disk but an LSN that can be used to enforce WAL. The `private` method `next_lsn_less_1` computes this.

The flush methods do nothing if the log is empty.

Much of the functionality in these two methods are common. The common part is in `internal_flush`. This is called by both methods to do all the work.

This concludes our discussion of the implementation details of the log manager. Additional details can be found in the documentation in the code itself.

6 Testing

As and when a class was implemented, unit testing was done on it to determine whether its methods worked in isolation. Then, the classes implemented so far were integrated and integration testing was performed to ensure that the classes worked together. Each time a bug was found and a fix applied, regression testing was done with previous test cases to ensure that the bug fix did not break any other part of the code. Finally, after the whole log manager was put together, we wrote a test driver to test the entire code. The recovery manager and the buffer manager calls were faked so that a problem could not be the result of code that we did not develop. We had the additional benefit that we started from scratch and did not have to use any existing code base. The test cases developed for this driver exercised most paths of the log manager implementation including several special cases and boundary conditions. Quite a few problems were exposed and fixed.

Unfortunately, space constraints prevent us from describing the various test cases.

7 Future Work

This section describes improvements that can be effected, both in terms of functionality and performance, to the current implementation of the log manager.

7.1 Duplex-Disks and Raw Disk Interfaces

The current implementation of the log manager uses a Unix file to store the log. Real database systems use a duplex-disk and directly access the disk to read and write log records, bypassing the OS. The current log manager implementation can be changed to handle this. The only changes required are in the implementation of `class Rawfile`.

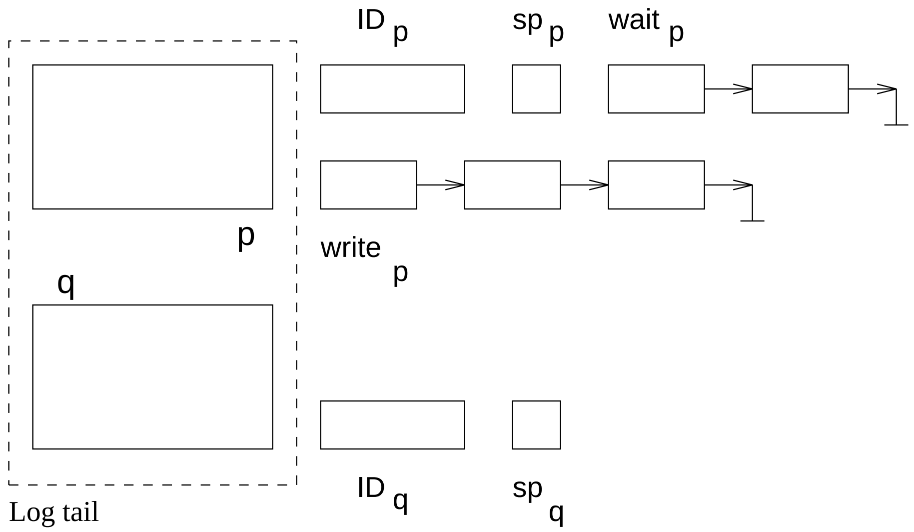

Figure 3: The design for enhancing concurrency. Each page in the log tail has the page ID of the log file that it corresponds to. Each page ID is protected by a semaphore. wait_p is the list of processes (transactions) waiting for the page ID, ID_p, to change. write_p is the list of transactions that have been assigned LSNs for page p but not yet written their records.

7.2 Allowing More Concurrency

The current implementation uses a single page for the log tail and locks the entire shared memory structure while performing any operation. This considerably reduces concurrency. Since log record writes are in the critical path of every update (i.e., every update operation has to log its update), maximizing concurrency is crucial.

This section outlines an alternate, more complex design for the log manager that considerably increases the concurrency.

Concurrency can be enhanced if we reduce the size of the critical section to encompass just the assigning of LSNs. Writing on the log tail page can proceed concurrently because the writes are to non-overlapping portions of the page (the mutual exclusion of assignment of LSNs guarantees this). The scheme is more complicated because we no longer know when a page can be flushed. Also, since LSN assignments proceed at a faster rate, we have to ensure that the page for a log write is in the log tail before allowing the write to proceed. This raises the question of what should be done if the page is not in the log tail.

We propose the following solution to tackle these problems. We assume double buffering, i.e., two log tail pages such that when one is being flushed to disk, the other can be filled by log writes. Associated with each log tail page is the page ID in the log file it corresponds to. Each page ID is protected by a semaphore. We denote the page ID corresponding to log tail page p as ID_p and the semaphore as sp_p. Figure 3 illustrates the configuration.

Let us walk through a typical set of log writes. Initially, both buffers are empty. ID_p is 0, ID_q is 1. Every log write goes through a critical section for assignment of LSN. Let the first five log records get LSNs assigned in page 0. All five of these can now concurrently write to page 0. Every time an LSN is assigned for a log write to page ID_p, an entry is made in write_p. When the corresponding `memcpy` to the log tail succeeds, the entry is removed.

When the sixth log write request arrives, the LSN assignment code detects that a page change has occurred[5].

[5] We are assuming that the first five log records exactly fit a page. This need not be the case. If a log record is split across pages, we can treat it as though it were two log records, one fitting into the current page and the other fitting into a new page. The only difference is that the second part does not get an LSN.

Thus, no more log writes to page ID_p occurs. This is made note of. Now, when write_p becomes empty, the flusher can be invoked.

The flusher will check whether another flusher is active for page q. If so, it waits until that flush completes. It will then start writing the page to disk. Since write_p is empty, it is guaranteed that no further updates to page ID_p will be made. (Flushers can check whether other flushers are active by having semaphores per log tail page that are downed and uped by the flushers only.)

Meanwhile, LSN assignments and log writes will continue in the page ID_q. Suppose that log writes to page ID_q are completed. No further log writes can occur because the page ID_p is still being flushed. This is detected by writers because LSN.pageID $\neq$ ID_p[6]. When a transaction detects this, it adds itself to wait_p and goes to sleep. (Because of double buffering, each transaction will know which buffer it will write to eventually.)

When the flusher is done with page ID_p, it will down(sp_p), update ID_p and up(sp_p). Finally, it will wake up all processes in wait_p. These transactions will again check LSN.pageID against ID_p before proceeding with the log write.

Not much can be gained by increasing the number of buffers, because, finally, a new buffer can be written to only after it has been flushed. Double buffering with the above strategy is likely to give good concurrency.

8 Enhancements to *Minirel*

This section outlines some enhancements that could be done to *Minirel* as a whole.

8.1 Revised Error Protocol

The current error protocol within *Minirel*, using which each layer informs the layer above about problems it detected is very simplistic. Higher layers can only display all the errors that occurred and not deal with them in any meaningful way. This section proposes an alternate error protocol.

The entire error protocol is based on a base `class Error`. Every layer derives a `class` from `class Error`. `Class Error` provides virtual functions to display itself, set its error level and error code, and return the level and code. Each layer will overload the methods to provide meaningful information about itself. In this context, an error level indicates the seriousness of the error: the error is fatal, the error will cause the current operation to fail, the error is a warning of impending problems, or the "error" is simply a message to the higher layer. Higher layers can inspect the errors returned by lower layers, display them if necessary, and take appropriate action. Error codes are defined by each layer and understood by the layer immediately above.

Globally, there will exist an object of `class Errorlist` which will store error objects of each layer (there will be a fixed number of layers – an array of pointers to `Error` objects would suffice). This object can be asked to display the whole hierarchy of errors or it can be asked to return the error code that a particular layer set, etc. `Class Errorlist` will also provide a method to clear out the errors in case a fresh start is to be made. Figure 4 illustrates an example object of `class Errorlist`.

This enhancement will complicate the code of the layers slightly because they have to set the appropriate object and indicate the error level and the error code. However, the overall design will be much cleaner if a revised error protocol like the one outlined above is used.

8.2 Crash Assumptions

The current implementation of *Minirel* requires a "crashing" transaction to exit its critical sections, release its locks, up its semaphores, and call appropriate destructors. This is hardly realistic. These restrictions must be relaxed if

[6]Each writer will down(sp_p) before checking ID_p and up it just before writing or sleeping.

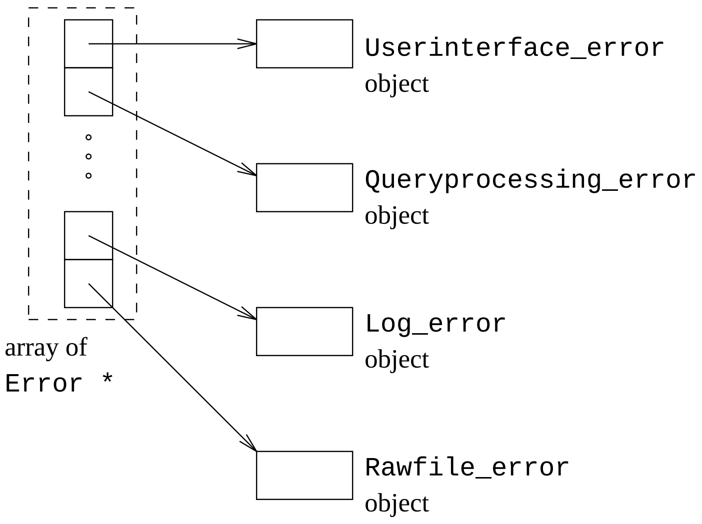

Figure 4: The `Errorlist` object. It maintains pointers to `Error` derived objects for each layer. If this object is asked to display the entire hierarchy of errors, for example, it will walk through the array and call the display method of each object.

Minirel is to function in a real environment where the system has little control over exceptional conditions.

8.3 Multiple Databases

Currently each *Minirel* process can run only one transaction and can access only one database throughout its existence. The implementation can be enhanced to allow a process to run multiple transactions. Also, each *Minirel* process must be allowed to access multiple databases.

8.4 A Networked RDBMS

Minirel currently runs on one machine only. It could be extended to run on multiple machines using a client-server architecture. The page server which serves pages from the database could be located one on server, the log database could be located on another server. Clients run queries and communicate with the servers through reliable messages to fetch pages, log updates, etc. Security and speed are two problems that must be tackled before this scheme can be put to effective use.

9 Summary and Conclusions

This report described the features required of an ARIES style log manager in an RDBMS environment. It then went on to describe the interface and implementation of various parts of the log manager and the simplifying assumptions made in the beginning and along the way. After outlining the test procedure, the report concluded with future work that would enhance the log manager and suggested a few enhancements to *Minirel* itself.

References

[Mohan et al. 92] C Mohan, D Haderle, B Lindsay, H Pirahesh, P Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*. 17(1):94-162. 1992.

[Murthy & Narasimhan 95] R Murthy, S Narasimhan. Heap File and Space Management in *Minirel*. CS 764 Project Report. 1995.

[Qiu & Su 95] K Qiu, F Su. *Minirel* Buffer Manager Report. CS 764 Project Report. 1995.

[Ramamurthy & Tsoi 95] R Ramamurthy, K Tsoi. An ARIES Recovery Manager for *Minirel*. CS 764 Project Report. 1995.