CEDA Log Structured Store

David Barrett-Lennard
Cedanet Pty Ltd
Perth, Western Australia
david.barrettlennard@cedanet.com.au

March 24, 2010

Abstract

This article describes the CEDA Log Structured Store which provides persistence for arbitrary sized binary objects. It supports frequent and fine grained transactions, recovery, backup, hot standby and is self cleaning to avoid fragmentation. It provides excellent control over clustering to optimise read performance, and the ingestion rate closely matches the sustained write rate of a hard-disk (e.g. 100 MB/sec), and yet provides proper journaling for atomicity in the face of failures.

1 Introduction

The *CEDA Log Structured Store* (LSS) is a persistent store for arbitrary sized binary objects, referred to as *serial elements*. It has all the features required for an industrial strength database storage layer.

Serial elements are read or written as a byte stream, in a manner similar to the C functions `fread` and `fwrite` used to read and write a file. The store can deal efficiently with very small and very large serial elements. The overhead on secondary storage is of the order 20 bytes per serial element.

Serial elements are identified by a 64 bit *Serial Element Identifier* (Seid). The LSS provides a mechanism for allocating new, unused Seids as required. Seids are logical (not physical), meaning that a Seid doesn't represent a physical location on disk. Therefore the LSS is able to move serial elements in order to avoid fragmentation and maximise clustering for read performance.

Serial elements are created, modified or deleted within the scope of a transaction, and the LSS ensures atomicity of each transaction – i.e. all changes made by a transaction are applied or else none are applied. For example, a transaction could fail to commit because of a power failure or a system crash. The next time the store is opened, any uncommitted transactions are automatically rolled back. CRC checks and other measures are taken to carefully validate the recovered data. The time for a recovery scan is bounded, and on typical hardware will never take longer than a few seconds.

Changes to serial elements are never performed in-place on disk. Instead new versions of serial elements are always rewritten at the end of a single growing *log* comprised of *segments* that are typically 512k or 1M byte in size. This practically eliminates disk seek overheads during writing, and data can typically be written close to the maximum sustained transfer rate of the hard-disk, which could translate into millions of serial elements per second.

Serial elements are written one after the other with no wasted space even though they may vary greatly in size. Serial elements never become heavily fragmented like files in a file system because serial elements are rewritten when they change size.

Multiplexing of the output allows for hot standby and backup consistent with 24 x 7 operation.

The LSS has been thoroughly tested. It has accumulated weeks of continuous stress tests on a range of different machines with different hardware, reading and writing many tera-bytes of data. This testing has included emulation of random system crashes and validation that the store recovers to a valid snapshot.

2 Serial Element Map (SEM)

At its essence the LSS is a persistent map called the *Serial Element Map* or SEM. This maps a Seid to an ordered list of bytes (i.e. a byte stream). When an LSS is first created the SEM is empty.

A serial element (identified by a Seid) is said to exist if there is an entry for that Seid in the SEM. This includes serial elements that have been written with zero size. There is an entry in the SEM if and only if the serial element has been created with a call to `WriteSerialElement` but has not subsequently been deleted with a call to `DeleteSerialElement`. A serial element can be recreated after it has been deleted (perhaps in the one transaction), simply by calling `WriteSerialElement` again.

2.1 Serial element Identifiers

Each serial element is uniquely identified by a 64 bit number called a Seid. The null Seid has the value 0 and cannot be used for any serial element. A Seid can be regarded as an array of 8 bytes. For reasons that are peculiar to the implementation of the LSS, a valid Seid requires that each byte be non-zero. A database typically bootstraps a root object with Seid `0x0101010101010101`. The LSS public headers define a constant named `ROOT_SEID` with this special Seid value.

Class `Seid` (defined in Seid.h) supports all six comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`). An ordering is imposed to allow the `Seid` datatype to be used for the key in `std::map` or a B-Tree.

An *application layer* using the LSS will typically record Seid values in the byte streams of serial elements, to allow serial elements to reference each other.

The number of serial elements is constrained by the maximum size of the store which is about 500 TB with the default settings, rather than the size of the 64 bit Seid space. This 500TB limit is simply a function of the size of the root block and can be increased easily if required.

3 Concurrency

The LSS implementation only takes partial measures to constrain concurrent access to serial elements. It is concerned with the integrity of its internal data structures. For example, the *LSS Segment Cache* is thread safe. However this is relatively low level, and insufficient to ensure serialisability of transactions defined in an *application layer* (which we use to refer collectively to the layers above the LSS). It is likely that an application layer provides its own concept of transactions, and only opens LSS-transactions for short periods in order to propagate changes made to transient objects resident in memory to the store.

LSS-transactions would be sufficient but for the fact that they are only intended for parenthesizing writes to the store. It is not necessary to open an LSS-transaction in order to read serial elements. Any number of threads can concurrently read serial elements (even the same serial element). This corresponds to the shared read access mode often provided by high performance database systems. Furthermore, multiple threads can be reading serial elements concurrently with an LSS-transaction that is creating, modifying or deleting (other) serial elements. In other words, the LSS provides no mutex to enforce mutual exclusion between the thread writing new data to the store, and the threads that are reading existing serial elements.

All writes (i.e. all API functions that modify the SEM) are fully serialised. Only a single thread can open an LSS-transaction at a time. This is enforced by a mutex within the implementation of the LSS. The mutex is locked by `OpenTransaction` and unlocked when `Close` is called on the transaction. Although LSS-transactions are mutually exclusive, an application layer may support concurrent changes to objects resident in memory, but a detailed discussion of that topic is outside the scope of this article.

It is therefore up to the layers above the LSS to make sure concurrent reading and writing is performed in a meaningful way (and as a minimum ensures that a serial element is not concurrently written and read by different threads). For example a layer on top of the LSS may implement a *lock manager* that ensures transactions follow a *Two-Phased Locking* (2PL) protocol.

Strict 2PL is where locks are taken as objects are accessed while the transaction proceeds, and all locks are released when the transaction ends. Unfortunately it allows for dead-lock scenarios and the normal approach is to pick one of the transactions (the "victim") to be aborted. However the LSS doesn't support aborting of an LSS-transaction (and roll back), so this concept would need to be implemented somehow within the application layer.

The LSS is effective for supporting *conservative 2PL* in the application layer. This is where all locks needed by a transaction are gained *before* the transaction begins. If locks are gained in a consistent total order, dead-locks are avoided. Conservative 2PL can be appropriate when relatively course level locking is used - such as by partitioning objects into large spaces that each employ a single mutex (which could optionally support a shared read access mode in addition to exclusive access).

4 Use of Abstract Base Classes

Most of the functionality in the API is provided using pure abstract base classes which serve as interfaces. In general these abstract base classes have a `Close` method, and this must always be called to destroy the object and free allocated resources. Once `Close` has been called it must be assumed the object has been destroyed so it is not permissible to issue any more method calls. It is very important not to use the standard C++ `delete` operator on the pointer.

The header file `Ceda/cxUtils/AutoCloser.h` defines a convenient template class `AutoCloser<T>` that can be declared on the frame and it calls `Close` in the destructor. This ensures the code is exception safe.

EOpenMode	Already exists	Doesn't already exist
-	fail	fail
OM_CREATE_NEW	fail	create+open
OM_DELETE_EXISTING	delete+create+open	fail
OM_CREATE_ALWAYS	delete+create+open	create+open
OM_OPEN_EXISTING	open	fail
OM_OPEN_ALWAYS	open	create+open

Table 3: EOpenMode

5 Creating or opening an LSS

All data in an LSS is stored in a single file. An LSS is created or opened with a call to `gfnCreateOrOpenLSS` which has the following signature:

```
ILogStructuredStore* gfnCreateOrOpenLSS(
    const char* lssPath,
    const char* deltasDirPath,
    bool& createdNew,
    EOpenMode openMode,
    const LssSettings& settings);
```

Table 1 describes these formal arguments. The file is opened with exclusive access so it is not possible for another process to open the same LSS file. `gfnCreateOrOpenLSS` never returns NULL. After using the store it must eventually be closed by calling the `Close` method. If an error occurs then `gfnCreateOrOpenLSS` throws a `FileException` (see `FileException.h`) or else `CorruptLSSException`.

6 Reading serial elements

As discussed in section 3, it is not necessary to open a transaction in order to read serial elements.

Reading a serial element may block on I/O if the required data is not already resident in memory in the *Segment Cache*. A very large serial element may take up many segments. Segments are loaded from disk *lazily* as the serial element is read as a byte stream.

The method `ILogStructuredStore::ReadSerialElement` is used to open a serial element with the given `Seid` for reading.

```
ICloseableInputStream* ReadSerialElement(
    Seid seid) const;
```

It is an error to call this function on a serial element that is currently opened for writing (within a transaction), or being deleted using a call to `DeleteSerialElement`.

Shared reading of serial elements is supported. i.e. any number of threads can independently (and concurrently) read the same serial element, assuming each such thread has made an independent call to `ReadSerialElement` - i.e. they don't try to share a returned `ICloseableInputStream`.

`ReadSerialElement` returns an input stream interface which can be used to read the bytes currently recorded in the serial element in sequence from start to finish. Seeking within the serial

Argument	Description
`lssPath`	A null terminated string representing the path to the LSS file to be opened or created. It may either use forward or back slashes in the path. If `lssPath = "memfile"`, then the LSS will be transient – i.e. it will reside entirely in memory.
`deltasDirPath`	`NULL` or else specifies the path to a directory in which to create delta files. See section 14
`createdNew`	Boolean out-parameter assigned the value `true` if and only if a new store was created.
`openMode`	An enumerated type that determines whether to allow an existing store to be opened or a new store to be created. The possible values are listed in Table 3. If the file already exists then there are three options: The function can either fail, open the existing file, or delete the old file and create a new empty one. Otherwise, if the file doesn't exist then the options are either to create an empty file or else to fail. This leads to $3 \times 2 = 6$ different modes, only 5 of which are actually useful. If the file exists when it's not expected or the file is absent when it's expected then a `FileException` exception is thrown. If `openMode` is either `OM_CREATE_ALWAYS`, `OM_DELETE_EXISTING` or `OM_CREATE_NEW` any existing file will be deleted and `createdNew` will be assigned the value `true`
`settings`	Defines various low level settings that influence the performance of the LSS. See Table 2 for a description of all the fields in the `LssSettings` struct.

Table 1: Arguments to gfnCreateOrOpenLSS

Field	Initial value	Description
`m_flushTimeMilliSec`	1000	Maximum time in milliseconds to flush the log after committing a transaction
`m_cleanerUtilisationPercent`	85.0	If segment utilisation (expressed as a percentage of the segment that contains useful data) falls below this threshold then a segment is cleaned
`m_enableFileBuffering`	false	If set then the Win32 file cache will be used. Typically not required because the LSS performs its own buffering, with its *Segment Cache*.
`m_maxNumSegmentsInCache`	32	Maximum number of segments in the segment cache. With default values segment cache is 32 x 512kB = 16 MB.
`m_numSegmentsPerCheckPoint`	128	Sets the rate at which the store is check pointed. With the default values a check point is performed after writing 128 x 512kB = 64 MB to the log. This controls the maximum time taken to perform a recovery scan. For a modern hard-disk, it only takes one or two seconds to read 64MB. Performing check points rarely has the advantage of writing less "meta data" to the log, and ensuring that the meta data is well clustered. It also means the root block is written less often.
`m_segmentSize`	524288	Size of each segment - the unit of download from disk. If too small, then performance becomes dominated by the head seek and rotational delay times of the hard-disk. If too large then performance becomes overly dependent on the clustering of related data. As a very rough guide, should equal the product of the maximum transfer rate of the hard-disk in bytes per second, times the seek time in seconds. E.g. for transfer rate = 50 MB/sec, seek = 10 msec then product = 500k. If an existing store is opened and the existing segment size doesn't match the segment size specified in the settings, then the requested segment size will be ignored.
`m_forceIncrementMSSN`	false	Force increment of the MSSN during start up

Table 2: LssSettings fields

element is not supported. `ReadSerialElement` returns `NULL` if no serial element exists with the given Seid.

Clients read from the stream with calls to `ReadStream`, which has the following signature:

```
int ReadStream(void* buffer, int
    numBytesRequested);
```

This blocks on I/O, trying to read `numBytesRequested` from the stream. It returns the actual number of bytes read which may be less than `numBytesRequested`, indicating the end of stream was reached.

`Close` must be called on the returned stream after it is used (including when exceptions are thrown by the LSS). It is allowable to close the stream before reading any or all of the data.

7 Seid Spaces

The high 32 bits of a Seid is called the *SeidHigh* and the low 32 bits is the *SeidLow*. For each SeidHigh there are about 4 billion available SeidLow values. This is referred to as a *Seid-Space*. Note therefore that a Seid-Space is identified by a particular SeidHigh value and there are about 4 billion Seid-Spaces available.

Within the application layer on top of the LSS, there may be multiple, independent purposes for storing serial elements. In that case it can be harmful to allocate Seids from a single global allocator because over time Seid values are interleaved across these unrelated purposes. The downside is that the meta-data (see section 17.1) used internally by the LSS to record the current physical location of serial elements cannot be so effectively clustered with the serial elements on disk.

To combat this problem, each Seid-Space is regarded as an independent space for *private* Seid allocations. This is easily achieved by persisting in the LSS the following *Seid Allocation Data* (SAD):

1. the *Seid Allocation Map* (SAM) which is a map from SeidHigh to the next available SeidLow for the purpose of allocating a new and unused Seid in the Seid-Space corresponding to that SeidHigh; and

2. the next available SeidHigh for the purpose of allocating a new and unused Seid-Space.

So with modest storage space overheads, the LSS provides multiple, independent spaces for Seid allocations.

There is a *consistency requirement* between the SEM and SAD. For any given Seid present in the SEM both the following must hold:

- its SeidHigh cannot be seen as unallocated according to the next available SeidHigh for the store; and

- its SeidLow cannot be seen an unallocated according to the next available SeidLow for that Seid Space.

To ensure this, calls to `WriteSerialElement` implicitly update the SAD as required. Therefore, strictly speaking, it can be optional for whether to explicitly call the Seid or Seid-Space allocation functions. Reasons why that may be useful are outside the scope of this article.

There is no converse requirement. i.e. it is allowable for large regions of Seid Spaces to have been reserved (i.e. marked as allocated in the SAD), and yet corresponding serial elements don't actually exist in the SEM. For example, when serial elements are deleted from the SEM, the Seids are never marked as available again in the SAD. Instead the Seid allocator will never recycle Seids. One good reason for this is to detect dangling Seid references reliably. It is also a pragmatic matter, because the allocator only records the next available value of a SeidHigh or SeidLow rather than sets or intervals that are free which would be much more expensive. It is also worth noting that a 64 bit address space for Seids is enormous and on hardware in the foreseeable future practically inexhaustible, so there is no justification to go to the trouble and expense of recycling Seid values.

The LSS implementation takes some corresponding liberties as far as ensuring allocation requests on the SAD are persisted. More specifically, the recovery scan after a system crash will ensure that the SAD and SEM are consistent in the sense defined above, but may have lost some of the allocation requests.

7.1 Allocation of Seids

Allocation requests are not regarded as part of a transaction on the LSS and this is reflected in the API where the Seid-Space and Seid allocation functions are available on the LSS without opening a transaction.

`CreateSeidSpace` returns a new and unused `SeidHigh` that represents a Seid-Space which can be used to allocate about 4 billion Seids.

```
SeidHigh CreateSeidSpace();
```

When a new serial element is to be written to the LSS for the first time, it is typically necessary to allocate a fresh Seid for it. This can be achieved with a call to `AllocateSeid`, providing a suitable SeidHigh value.

```
Seid AllocateSeid(SeidHigh seidHigh);
```

Any number of threads can concurrently allocate Seid-Spaces and Seids because the above allocation functions are thread-safe.

7.1.1 Allocation of affiliate Seids

The following is an alternative Seid allocation function that is passed a Seid as an in-out parameter. The input value is an existing Seid in the LSS and the output value is a new, unused seid that is deemed to be affiliated with the input Seid. Typically the new Seid will share a large portion of the prefix of the Seid considered as an array of 8 bytes, and therefore the Seids will tend to be localised with respect to the 8-level hierarchical map used to index the physical locations of the serial elements.

```
bool AllocateAffiliateSeid(Seid& seid);
```

This is useful for very large sets of serial elements where it is difficult to know a-priori how to partition the serial elements into separate Seid-Spaces.

Before this function can be used to allocate Seids, it is first necessary to "bootstrap" by calling `CreateSeidSpace` to allocate a Seid space, then `AllocateSeid` to allocate a root serial element in the space. Then, instead of calling `AllocateSeid` to allocate additional serial elements, it can be preferable to call `AllocateAffiliateSeid`. The Seid passed into the function represents an "affiliate" Seid to which the new Seid returned by the function will be clustered.

E.g. the affiliate may be a parent node in a tree of nodes. `AllocateAffiliateSeid` does a good job of allocating Seids for trees of nodes which grow over time from any position by adding child nodes.

8 Transactions

A transaction is associated with *mutative* work on the LSS. i.e. for creating, rewriting or deleting serial elements. As discussed in section 3, transactions on the LSS are serialised using a mutex.

During a transaction it is possible to delete or write any number of serial elements. Writing a serial element encompasses creation of a new serial element as well as rewriting an existing serial element.

8.1 Opening a transaction

A client opens a transaction by calling `OpenTransaction` on the `ILogStructuredStore`. This returns a pointer to a `ILssTransaction` which is defined as follows:

```
struct ILssTransaction
{
  virtual void Close() = 0;
  virtual void FlushWhenClose() = 0;
  virtual ICloseableOutputStream*
      WriteSerialElement(Seid seid) = 0;
  virtual void DeleteSerialElement(Seid seid) =
      0;
  virtual void DeleteSeidSpace(SeidHigh
      seidHigh) = 0;
};
```

Note well that only the thread that called `OpenTransaction` is allowed to call any of the methods on the returned `ILssTransaction`.

8.2 Closing a transaction

There is no concept of clients aborting a transaction. If a thread begins a transaction then that thread (and only that thread) must eventually commit the transaction with a call to `Close`. As such there is no concept of roll-back during the normal operation of the LSS. Roll-back only occurs during recovery (i.e. when the store is opened when it was not previously closed gracefully).

An exception may occur in the middle of a transaction. For example, the thread that opened the transaction makes a call to `ReadSerialElement` and this fails because of a low level I/O error, throwing a `FileException`. It is vital that the client still calls `Close` even though an exception occurred. Otherwise the mutex will not be closed, and there could be a subsequent dead-lock - such as when the client tries to close the LSS.

The best way to ensure correctness in the face of exceptions is to declare an instance of an `AutoCloser<ILssTransaction>` on the frame in order to perform a transaction on the LSS within a lexical scope.

Note that when such an internal I/O error occurs, the LSS will enter an “error” state, preventing any transactions from being propagated to disk, even though the transaction is explicitly closed.

It is an error to close a transaction that is in the middle of writing a serial element. In other words, after calling `WriteSerialElement` to write a serial element, the returned `ICloseableOutputStream` must be closed before it is allowable to close the transaction.

If `FlushWhenClose` has previously been called on the transaction then `Close` will *synchronously* flush this and all previous transactions on the LSS.

Closing a transaction destroys the transaction so it is not permissible to call any of the methods again.

8.3 Flushing transactions for Durability

Database systems conventionally provide the ACID properties. The ’D’ stands for *durability* which means that when a transaction is committed it is made durable as part of the call to commit by synchronously flushing the transaction to disk.

`FlushWhenClose` can be called on an opened transaction to put it into a mode where it will synchronously flush the transaction to secondary storage when the transaction is closed. It must only be called by the thread that originally opened the transaction. The subsequent `Close` will only return after the transaction (and all previous transactions) have been written to disk - at least according to the Win32 calls. Note that the LSS file is opened with “no write through cache” so in theory all flushed transactions will be durable. Note well that this may not actually happen in practice for hard-disks that have their local cache enabled.

Durability is particularly relevant to the management of data that relate directly to real world processes such as airplane reservation systems, or financial systems. It is also important for distributed transactions – typically involving a multi-phase commit protocol. In these cases, a transaction on a computer is associated with the state of objects or events in the real world such as when a client withdraws cash from an ATM. Clearly it is necessary for the database to correctly record all such withdrawals. This leads to the durability requirement. In practice this means that every transaction must be flushed to disk as part of the commit.

Dedicated database servers may employ disk write caching that promises to (eventually) write all data in the cache to disk. This requires a battery backed up cache, and other facilities, such as intercepting the RST signal to avoid clearing the cache, and use of ECM (Error Correcting Memory). Unfortunately “normal” hard-disks provide a disk write cache that is unsuitable for database servers, and this can’t merely be fixed by using a UPS. Therefore it is necessary to disable the disk write cache. This may require changing jumpers on the hard-disk, or running special control software provided by the manufacturer.

Unfortunately, with no write cache, disk flush operations become very expensive. With a stock hard-drive at 5400 to 7200 RPM, there can be at most 50 to 70 disk flushes (i.e. transactions) per second. In many applications this is inadequate.

By making durability optional, the CEDA LSS is also applicable to the management of data that doesn’t need to be synchronised in real time with real world processes. Examples are editing of text documents, spreadsheets, statistical analysis, web browsing, GIS, multimedia databases, source code repositories and CAD. In these cases the durability constraints can be relaxed a little - such as by only flushing transactions to disk every few seconds. Atomicity is still required to protect the integrity of the data. However, a transaction only “commits” in the sense of defining an atomic unit of work, rather than demanding it go to non-volatile storage as part of the commit. This of course means that a user may lose some edits on system failure, but losing at most a few seconds of work may be acceptable in certain applications.

8.4 Deleting serial elements

An opened transaction can delete a serial element with a given Seid with a call to

```
void DeleteSerialElement(Seid seid)
```

`DeleteSerialElement` must only be called by the same thread that originally opened the transaction.

It is an error to delete a serial element that is currently opened for reading or writing.

8.5 Deleting a Seid space

An opened transaction can delete the Seid space associated with the given `SeidHigh` with a call to

```
void DeleteSeidSpace(SeidHigh seidHigh)
```

The Seid space must be empty — i.e. by calling `DeleteSerialElement` as required to delete all serial elements in the Seid space.

`DeleteSeidSpace` must only be called by the same thread that originally opened the transaction.

8.6 Writing serial elements

Each time a serial element is written to the LSS, it must be rewritten in its entirety - even if only a small part of its content changes. It will in fact be written to a completely new location within the store - i.e. at the "end of the log".

If that seems particularly wasteful then the serial elements should be smaller (i.e. finer grained). There is actually a trade off here. Small serial elements reduce the number of bytes to be written to disk when changes are made. However larger serial elements may improve read performance because related data tends to remain clustered together on disk. Also larger serial elements reduce the various overall space and time overheads that are associated with each serial element, such as the need to index its physical location. Note finally that the product of transfer rate and seek time for a modern hard-disk is quite large - of the order 512k, so it is not efficient to write lots of small objects to disk if they can become scattered over time.

To write a serial element with a given Seid, call the method `WriteSerialElement` on a `ILssTransaction`

```
ICloseableOutputStream* WriteSerialElement(Seid
    seid);
```

This function is used to write new serial elements and also to re-write existing serial elements. i.e. if the serial element already exists then the previous rendition will be replaced by a new one.

The returned `ICloseableOutputStream` (which is never `NULL`) can be used to write the content as a byte stream. Note that the entire serial element must always be written with calls to `WriteStream` which has the following signature:

```
void WriteStream(const void* buffer, int
    numBytes);
```

Note that if no data is written to the serial element then it is still deemed to exist.

After writing the data, the `ICloseableOutputStream` must be closed with a call to `Close`. Furthermore, it must be closed before the next call to `WriteSerialElement`, `DeleteSerialElement`, `DeleteSeidSpace` or `Close` on the transaction.

It is an error to call `WriteSerialElement` for a serial element that is currently opened for reading (perhaps by a different thread).

`WriteSerialElement` must only be called by the same thread that originally opened the transaction.

9 Boot strapping a store

It is common for a serial element to store (within its byte stream content) the Seids of other serial elements. For example these could represent the "children" in a whole-part hierarchy of objects.

Typically an application using the LSS will need to write some sort of "root" registry or directory object to the store with a known Seid. This is the starting point for accessing all other objects in the store. `ROOT_SEID` is the Seid for this root serial element.

Just after creating a new LSS, it is guaranteed that the first call to `CreateSeidSpace` will return the high 32 bits of `ROOT_SEID`. It is then guaranteed that the first call to `AllocateSeid` (passing in the high 32 bits of `ROOT_SEID`) will return `ROOT_SEID`.

10 Clustering

A developer using the LSS needs to be concerned with clustering related data together, in order to maximise read performance. This is achieved by writing related data close together in time (so the related serial elements tend to be written to the same segments). Note that rewriting individual serial elements over time can have the effect of upsetting the clustering (because charges to serial elements are never made in-place). Reclustering simply involves rewriting a collection of related serial elements to the end of the log. The background cleaner thread will automatically defragment the store.

The easiest way to achieve good read performance is to partition a very large database into mutually exclusive groups of serial elements, where each group is characterised as follows:

- The serial elements in a group are all closely related, meaning that when one serial element is read from disk, it is likely that other serial elements in the group will also be read in the near future;
- There is a tendency for the serial elements in a group to be written to the same segment. This is achieved by writing batches of related serial elements to the LSS at a time. As a corollary to this requirement, a group of related serial elements can't be so large that it defeats the whole idea of them being "clustered together"; and
- Every serial element in the group shares the same high 32 bits of the Seid. This allows the LSS to achieve clustering in its internal hierarchical map used to track the physical locations of serial elements.

Over time there is an "increasing entropy" effect where related serial elements become spread around the disk. It can be very beneficial to recluster serial elements, particularly serial elements used to implement directory structures. This is achieved by occasionally rewriting all the relevant serial elements to the LSS in a single "batch".

It is actually the excellent write performance of the LSS that makes it economical to recluster related data together. Therefore the LSS can provide very competitive read performance.

11 Supported Platforms

The CEDA LSS is currently supported on all flavors of 32 bit Windows (i.e. Win95, Win98, WinXP etc) and will run as a 32 bit application under all x64 versions of Windows. The store is written to the hard-disk (which could be FAT32 or NTFS) as a single file. This file grows as required to accommodate new data written to the store. It can also be used with a raw partition.

12 MSSN

MSSN stands for *Missing Shutdown Sequence Number*. This is a value stored in the root block of the LSS. It equals the number of times that the store has not been gracefully shutdown over its life.

This is zero if the store has always been properly closed (so that the store is check pointed and flushed correctly). A large value indicates that there have been many power failures or the client software is not closing the LSS correctly.

The MSSN is useful for correctly mapping Seids to a larger global address space that encompasses objects stored on many computers. Single user applications of the LSS have no need for the MSSN other than a simple diagnostic.

When the store is first created the MSSN is initialised to zero. The value persists and is only incremented during recovery if it is found that the store wasn't previously shut down gracefully.

13 Throttle control

Consider a thread that quickly generates large amounts of data that needs to be written to disk, and the thread writes these changes using a large number of fine grained transactions. This thread is considered to be a "producer" and the LSS is the "consumer".

Initially all 32 segments in the LSS segment cache will be available to store the data written to the LSS. Therefore calls to write data to the `ICloseableOutputStream` (returned by a call to `WriteSerialElement`) will simply store the data in memory (in the segment cache) and return quickly. However, once the segment cache is full of "dirty" segments (ie segments that need to be written to disk), calls to write data to an `ICloseableOutputStream` will block on disk I/O.

Consider further that in the layers above the LSS, there are mutexes (ie locks) used for concurrency control of higher level data structures or objects. The thread that generates large amounts of data may acquire these locks. The problem then is that these locks could be held while the thread blocks on I/O. This could make the system unresponsive, or reduce concurrency. It would be better if the thread avoided tying up system resources while blocking on I/O.

To solve this problem the LSS provides a facility for flow control between "producer" and "consumer". The thread has the option of calling the following functions defined in `ILogStructuredStore`:

```
// Blocks until it is appropriate for the
// producer to begin writing changes to
// the LSS again (because the LSS has
// written enough segments out to disk).
void BlockUntilLowWaterMark() const;

// Returns immediately, and indicates whether
// the producer has written enough changes
// to the LSS such that it should call
// BlockUntilLowWaterMark() in order to wait
// for the LSS lazy writer to "catch up"
bool ReachedHighWaterMark() const;
```

The "producer" thread should call `BlockUntilLowWaterMark` before it acquires locks on valuable system resources. Then during the period it is waiting for the LSS to catch up (by writing dirty segments in the segment cache to disk), it avoids holding locks.

14 Backup and hot standby for the LSS

The LSS optionally supports hot standby and incremental backup.

Note however that at present, hot standby has fairly relaxed assumptions about how up to date the standby store must be.

The backup / hot-standby system is compatible with 24x7 operation of a store which continuously reads/writes large amounts of data. The LSS properly supports applications that are write bound for prolonged periods.

When the LSS is created or opened, a path to a directory for the delta files can optionally be provided. LSS delta-files will automatically be written to this directory. These files are named nnnnnn.lssdelta where nnnnnn is a sequence number, called a *Check Point Sequence Number* (CPSN).

Note that the path to the main LSS file is independent of the path to the directory of delta-files. They could easily be on different hard-disks.

To avoid limiting the write performance of the system, it is recommended that a separate local hard-disk be used for storing the deltas. This will allow the deltas and the LSS file to be written concurrently. It also means either hard-disk can fail without losing data.

Note that with virtual file systems it is easy to have delta files written directly to a remote site. However that may expose the LSS to network outages. A better strategy may be to write deltas to a local hard-drive, and a separate process is responsible for copying these files to a remote site. During network outages the application is able to continue running.

A single delta-file is written for each check point on the LSS. The LSS stores a CPSN in the main LSS file. This helps ensure that deltas are applied in the right sequence. The CPSN directly corresponds to the sequence number used for naming the delta files.

With the default settings the LSS performs a check point after writing 128 segments (or 64 MB). A check point is also performed whenever the store is closed.

Delta files respect check point boundaries. Note in turn that check point boundaries respect both flush unit and transaction boundaries.

15 Hot standby configuration

As long as the main application is not running, (and the main LSS file is not opened) it can be copied using the file system. This creates a "level 0 backup". The copy will of course have the same CPID and CPSN, recorded in the root block. However, there are two significant problems with making a complete copy of the store

- If the store is large then it can take a long time;
- The application that reads/writes the store can't be running while the copy is made.

Once a copy has been made the delta-files can be used to very efficiently and safely bring the copy into sync with the main store.

Consider that we have previously created a "standby" store (by making a file system copy). Let the 24x7 application be configured to automatically create the delta-files in the normal way. Let LssApplyDeltas.exe be run repeatedly so it applies deltas to the "standby" as soon as they become available. At quiescence the standby will match the main store.

Note that this process is compatible with 24x7 operation of the main store (because it never needs to be shut down).

It is easy to create any number of "standby" stores in various stages of how up to date they are, because deltas are not consumed when they are applied to a store. Furthermore the standby stores and the deltas can be backed up to tape etc. Therefore this approach provides a great deal of flexibility.

Argument	Description
`level0path`	The path to an existing LSS store, called the "level 0"
`deltasDirPath`	the path to the directory containing the delta-files
`cpsn2`	[Optional] A "one past end" value of the cpsn, to specify what delta files should be applied to the level 0.

Table 4: LssApplyDeltas command line args

16 Utility Console Applications

16.1 LssApplyDeltas.exe

A console application called LssApplyDeltas.exe is able to apply deltas to an existing LSS store, called a "level 0", to bring it more up to date.

On the command line two or three arguments may be specified:

```
LssApplyDeltas level0path deltasDirPath [cpsn2]
```

The arguments are described in Table 4. The half open interval [cpsn1, cpcn2) is applied. cpsn1 is determined automatically from the level 0 file. Note that delta files are applied up but not including cpsn2.

LssApplyDeltas can optionally be passed the cpsn2 parameter to limit the number of deltas to be applied. Currently this is only at the course granularity of check-point boundaries. [In the future it is expected that it will also be possible to specify a date/-time stamp for more precisely controlling what transactions are applied]

If the delta files directory contains the delta files from 0 onwards, and there is no level 0 LSS file, then LssApplyDeltas.exe will actually create a level 0 from the delta files

The LSS always uses the extension "partial" for the current delta file being written. This is renamed with the extension "lss-delta" after the delta file has been completed. It is assumed that this approach is sufficient to ensure that LssApplyDeltas won't apply a partially written delta file.

LssApplyDeltas is idiot proof in that it will never apply an inappropriate delta.

16.2 LssCompare.exe

This console application can be used to compare two LSS stores to see if they are (logically) equivalent - i.e. they represent the same SEM. This is useful for validating the backup system. On the command line, path1 and path2 are paths to two different LSS files to be compared:

```
LssCompare path1 path2
```

17 LSS implementation

A *Log Structured Store* ([1] and [3]) stores all data in a single ever-growing linear sequence of records (called the *log*). Records in the log are never overwritten, making it straightforward to support transaction atomicity by simply writing special snapshot marker records to the log. Since data items are never modified in-place, the system must maintain an index structure that records the current physical location of a given data item. A LSS records the index structure itself in the log; typically this is written during regular check points. In [3] the index forms the majority of what is referred to as meta-data, and it is pointed out that a LSS is very space efficient in the sense that the amount of meta-data written to disk can be comparatively small compared to other approaches. Recovery to a valid snapshot position simply requires a forward scan through the log from the last valid check point up to the last snapshot record to bring the index up to date.

Most DB systems employ data that is read or written in pages, and atomicity of transactions is achieved by the technique of *Write Ahead Logging* (WAL) of changes to the data pages to a separate log file, which can be scanned during recovery as required to undo/redo partially completed transactions.

The LSS achieves excellent write performance by treating the log itself as the data! Therefore all writing occurs at the end of the log, allowing for continuous writing with minimal movements of the disk head.

The log is divided up into relatively large 512k pieces (called segments). Reading and writing at this coarse granularity minimises disk head seeking overheads.

There are four background threads that take on responsibility for writing segments, flushing the log, check pointing the store (setting the point from which a recovery scan is required) and cleaning partially fill segments to avoid fragmentation.

17.1 Recoverable Packet Map

The *Recoverable Packet Map* (RPM) is an 8 level hierarchical map keyed by 64 bit Seid used to record the locations of serial elements in the store. The *Segment Utilisation Table* (SUT) records the utilisation of every segment in the LSS. Both of these data structures are only updated on disk during a check point. Check points are normally performed after writing 64MB to the store. This places an upper bound on the time required for a recovery scan.

17.2 Assumptions on the hard-disk

Most databases providing ACID properties assume that the hard-disk promises atomicity at the granularity of disk sectors (which are typically 512 bytes), and will always write disk sectors on the platter in the same order they were written to the OS.

Unfortunately many hard-disks on the market fail to meet these requirements.

The LSS goes to a lot of trouble to avoid data loss without making such strong assumptions on the hard-disk. It is permissible for disk sectors to only be partially written, and also for writing to the platter to be out of order. This is achievable because the LSS doesn't use Write Ahead Logging (WAL), and it employs 128 bit check point identity testing as well as a 32 bit CRC check on "flush units" that are read during the recovery scan.

The root block contains two independent copies of the root block data, written in strict alternation, and CRC checked.

17.3 Comparison to conventional database systems

Most databases use the ARIES algorithm (or similar). This allows for in-place changes to be made to binary objects on disk. However it must use *Write Ahead Logging* (WAL) to support atomicity. This has a number of disadvantages:

- it is vital that changes be written (and flushed) to the log before the corresponding changes are made to the "real" data. Unfortunately most off the shelf hard-disks need to have

their local cache disabled to ensure that data is not written in an order that conflicts with the WAL assumption;

- the write performance is essentially halved because all changes must be logged - i.e. written to disk twice;
- writing changes in-place means the disk head needs to seek around a lot. Sophisticated techniques are required to reduce these problems. For example dirty pages are typically ordered so they can be written to disk by a lazy writer thread to minimise disk-head seek times (this is called elevator seeking). In practice, the product of transfer rate and seek time for a modern hard-disk is quite high - e.g. 512 kByte. Therefore it is very inefficient for the disk head to seek around only writing a small number of bytes at a time; and
- writing changes in-place usually means that objects can't vary in size over time. Therefore objects that contain strings and other variable sized data either need to reserve a fixed size buffer, and impose a limit on the size, or else store the string separately. Storing the string separately hurts clustering.

A log structured store eliminates these problems. Write performance is typically limited only by the maximum transfer rate of the hard-disk.

17.4 Check pointing

The LSS internally uses a hierarchical map to track the physical locations of all serial elements in the store. It also maintains information about the current utilisation of all the segments.

All this information is itself written to the root block or the log, but only during a "check point". The root block is updated using Challis' algorithm ([2]).

A check point is performed at the following times:

- after the store is opened and a recovery scan is required;
- after writing 128 segments (i.e. 64MB) of data to the end of the log; and
- when the store is closed;

A recovery scan is performed when the store is opened and it was found that it had not previously been closed gracefully. In that case it has to scan all segments in the end of the log since the last check point. This allows it to recover all committed transactions and roll back any uncommitted transactions.

The maximum time for the recovery scan is bounded by the time taken to read the segments at the end of the log since the last check point. The time to read 128 segments only takes a few seconds on a modern hard-disk.

17.4.1 Check point identifiers

The LSS generates a 128 bit GUID called a Check Point Identifier (CPID) for each check point. The current CPID is stored in the root block of the LSS. Each delta-file stores an input CPID and output CPID. A delta file may only be applied if its input CPID matches the current CPID of the store. The store's CPID is then set to the output CPID defined by the delta-file.

Both the CPID and CPSN are used to validate a delta-file (i.e. to see whether it is allowed to be applied to the store). Note that two stores can share a common ancestry, then diverge. The use of CPIDs ensures that delta-files are never applied incorrectly.

Note that the first CPSN to be applied isn't specified on the command line to LssApplyDeltas.exe. Instead the LSS can work this out itself (because it stores the current CPSN in its root block). This, together with the CPID validation makes LssApplyDeltas.exe "idiot proof".

17.5 Lazy Writer

The LSS uses a background thread called the "Lazy writer" to write dirty segments in the segment cache to disk. Therefore the thread that opens a transaction and writes some serial elements will typically only write the byte stream to a buffer in memory, allowing it to complete the transaction very quickly without blocking on I/O.

It is important to understand that ending a transaction implicitly commits it, but only in the sense of defining an atomic unit of work. The actual data is written to disk in the background.

When writing larges amounts of data, the segment cache can become full of dirty segments, and in that case it is possible for the thread performing a transaction to block on I/O.

17.6 Cleaning

When the LSS is opened, a background thread is automatically started that cleans segments with a poor utilisation (i.e. below a preset threshold). The data on a segment to be cleaned is written to the end of the log, allowing the segment to be returned to an internal free segment pool. Because of this, users of the LSS never need to concern themselves with fragmentation of the store.

When the store is created or opened, the cleaner threshold can be specified. This defaults to 85% meaning that all segments that are less than 85% utilised will be cleaned.

The cleaner is provided an ordered list of segments to clean after each check point.

References

[1] Mendel Rosenblum and John K. Ousterhout, *The Design and Implementation of a Log-Structured File System*, ACM Transactions on Computer Systems, 1992, Volume 10, pages 1–15.

[2] Challis, M. F., *Database Consistency and Integrity in a Multi-User Environment*, Databases: Improving Usability and Responsiveness, Academic Press, pages 245-270, 1978.

[3] David Hulse and Alan Dearle, *A Log-Structured Persistent Store*, Proceedings of the 19th Australasian Computer Science Conference, 1996, pages 563–572.