Log Structured Store

Aug 2005 David Barrett-Lennard

1 Log structured store (LSS)

1.1 Summary

Ceda uses a *Log Structured Store* (LSS) as the underlying persistence mechanism. Advantages are

- The store is simpler that the classical page based systems, like ARIES.
- No particular hardware or OS support is required --> platform independence
- Write performance is optimised – because all objects are written to the end of the log
- Space utilisation is excellent because objects of all sizes are written "crunched up" to the end of the log
- Recovery is simple because the log is self describing. A simple scan of the log allows for recovery of indexing information.
- Because recovery is fast and efficient, check points can be done infrequently. This greatly reduces the amount of indexing information that is written to disk
- It is easy to see how compacting + cleaning algorithms can be devised that recover space and recluster the segments to optimise read performance.
- Cleaning eliminates fragmentation.
- The persistent store has been decoupled from concurrency, simplifying the design. Rather than all of ACID, it only promises atomicity.

Ceda uses a persistent history buffer to represent all changes to persistent objects – even individual character insertions and deletions in a text document, or resizing of a slider control. This reflects the capability for synchronous editing. Therefore, good write performance is important. A log structured store seems ideal for this purpose.

1.2 Overview

A *Log-Structured Store* (LSS) unifies the object database with the log. ie objects are written to the log, not to a separate area.

As new versions of objects are written to the end of the log, previous versions are rendered obsolete. Therefore the log requires periodic garbage collection (called *cleaning*). This occurs in parallel with the normal operation of the store.

The log is stored as a linked list of segments. A segment is fairly large to reduce the impact of disk seek operations. Over time, more and more objects within a previously written segment become obsolete. Cleaning involves the relocation of live objects, providing an opportunity to re-cluster objects. It also facilitates application level garbage collection.

The LSS is a persistent heap, and only manages opaque blobs of binary data. These are called *serial elements*. A layer above the LSS called the *Persistent Object Store* (POS) associates a serial element with the serialised state of a persistent object. There is no size limit on a serial element. A serial element is identified by a 64 bit *Object Identifier* (OID). This is assigned when the serial element is first created, and kept for life.

Internally the LSS breaks up large serial elements into smaller pieces called *packets*. Clients of the store are insulated from this implementation detail.

A client of the LSS reads or writes a serial element as a stream of bytes (through the `IFile` interface). Neither the LSS nor clients of the LSS need to store a serial element as a single contiguous block in main memory.

1.2.1 Object Identifiers (OIDs)

An OID is a 64 bit number that uniquely identifies a packet or serial element on a given LSS. It is composed from two 32 bit values called `OIDhigh` and `OIDlow`.

Note that an OID doesn't point directly at a serial element in the log. An OID is assigned to a serial element forever, even though the serial element may be written to the enf of the log many times. The OID is like a handle and the *Recoverable Packet Map* (RPM) maps the OID to the current position of the serial element in the log, providing the extra level of indirection to allow the serial element to move around and yet still be found using the same OID.

1.2.2 No steal policy

A steal policy is where dirty objects or pages are written to a data store even though they have not been committed yet. This could be important for long running transactions that modify many objects, or for page based systems where a page always contains uncommitted updates due to fine-granularity locking and overlapping transactions' updates to that page.

However, it is assumed that in Ceda transactions are short lived, and will only modify a small number of objects. Therefore we assume a no-steal policy, so objects will only be written to disk when a transaction commits. This implies that no undo work is required on recovery.

1.2.3 External interface provided by the LSS

The `ILSS` interface contains the following functions

Function	Description
`void Close()`	It is crucial that the LSS be explicitly closed - otherwise there will be data loss
`bool ObjectExists(OID oid) const`	Does a serial element with the given OID exist?
`IFile* Read(OID oid)`	Get a file stream for reading a serial element
`void Serialise(Archive& ar)`	Serialise the entire LSS
`void Flush()`	Flush all changes.
`OSID GetCurrentOsid() const`	Get the current OID high used for OID allocations
`OID AllocateOID()`	Allocate a new and unused OID for a new serial element
`OSID AllocateOsid()`	
`ILSSCommitPhase* CreateCommitPhase()`	Create a commit phase which is used to write serial elements to the store, or permanently delete serial elements. The returned commit phase has been allocated on the heap and *must be deleted*. Failing to do so prevents further commit phases from being opened. Multiple threads can call this function, but only one thread can open a commit phase at a time. A thread will block until a thread that has opened a commit phase deletes the commit phase. It is an error to close or delete the LSS while there is an open commit phase.

All changes (ie mutative work) done on an LSS must be done by a thread that has opened a commit phase. A commit phase is intended for a single thread. Only the thread that called `CreateCommitPhase()` on the LSS is permitted to call the functions in this interface, or delete the commit phase. A commit phase must be deleted after it has been used.

Function	Description
`void Delete(OID oid)`	Permanently delete a serial element
`IFile* Write(OID oid)`	Get a file stream for writing a serial element

1.2.4 The Persistent Object Store (POS)

The *Persstent Object Store* (POS) is the client of the LSS. The implementation of the POS is described in detail later in this documentation. To help understand the LSS it is useful to have an overview of the POS.

The LSS is responsible for managing persistent serial elements, keyed by OID. The POS is responsible for managing the deserialised form of those serial elements - ie the `IPersistable` objects in memory. For this purpose the POS makes use of the following data structures

- A *Resident Object Table* (ROT) which is a map from `OID` to pointer location in memory, to keep track of the set of the `IPersistable` objects that are reachable from the persistent store root and are currently resident in memory
- A *Dirty Object Set* (DOS) which keeps track of the persistent objects that have been marked as dirty and need to be written back to the LSS.

The POS assumes that `IPersistable` objects use `pref<T>` smart pointers to reference other persistable objects. A `pref<T>` stores an `OID` and this is used to "fault in" persistable objects on demand (ie when the `pref<T>` is first dereferenced). The POS first tests whether the object is already in memory by looking up the `OID` in the ROT. If not then the POS makes a request for the serial element in the LSS with the given OID, allowing the `IPersistable` object to be created and deserialised.

It is assumed that all access to persistable objects is done in the scope of a `CedaLock`. Therefore there can be at most one thread that makes a read request on the LSS at a given time.

The DOS isn't processed at the end of every `CedaLock` transaction. Rather, changes to the C++ objects in memory are allowed to "accumulate". This is very important for performance. For example, when a user makes hundreds of independent changes to an image (say over a period of 2 seconds) it is likely that the image is only marked as dirty and added to the DOS once. When the DOS is eventually processed, only the final rendition of the image will be written to disk.

A worker thread is responsible for processing the DOS. It is coded as follows

```
class DirtyObjectSetWriter
{
    void DirtyObjectSetWriter::SignalWriteDirtyObjectSet()
    {
        m_wakeThread.Signal();
    }

    void WorkerThreadFn()
    {
        const int PROCESS_DIRTY_OBJECT_SET_TIMEOUT_MSEC = 2000;

        while(!m_requestShutDown)
        {
            m_wakeThread.Wait(PROCESS_DIRTY_OBJECT_SET_TIMEOUT_MSEC);
            if (m_requestShutDown) return;

            // Get a CedaLock, in order to be able to serialise the objects in the DirtyObjectSet
            CedaLock lock(m_abortThread);
            if (m_requestShutDown) return;
            ASSERT(lock);
            m_ps.WriteDirtyObjectsToLSS();
            lock.Commit();
        }
    }
}
```

This ensures that when objects are added to the DOS, at most 2 seconds will elapse before a worker thread processes (or at least tries to process) the DOS - ie writes all the dirty objects to the LSS and clears the DOS. Note that the DOS is processed in the scope of a `CedaLock`. to ensure that no other thread is able to make changes to the persistent objects that are being serialised to the LSS. Therefore at most one thread has read or write access to the LSS at a given time.

When the DOS is processed, we say that a *commit phase* is performed on the LSS. This involves the following steps

1. `ILSS::CreateCommitPhase()` is called to obtain an `ILSSCommitPhase` interface pointer.
2. `ILSSCommitPhase::Write()` is called for all the dirty objects
3. `ILSSCommitPhase::Delete()` is called for all the objects to be permanently deleted
4. The `ILSSCommitPhase` is deleted.

The LSS uses a large segment cache, so often a commit phase will be able to write all its data to the LSS without blocking on I/O. It is important to note that a commit phase doesn't flush the log. A separate function `ILSS::Flush()` is provided for that purpose, but performance is degraded if it is called unnecessarily.

By parenthesising the changes made by a `CedaLock` transaction, the LSS is able to ensure atomicity, in case the system crashes part way through writing data. On recovery the LSS starts in a state where no transaction has only been partially applied.

Note that a transaction doesn't enter its commit phase unless all fallible work has been completed. The commit phase is meant to be infallible – in the sense that it can only fail due to a hardware or power failure. For that reason there is never a need to explicitly abort the commit phase in software.

1.2.5 Durability

Durability relates to the 'D' in the so called ACID properties typically expected of OO and relational data base systems.

ACID properties are particularly relevant to the management of data that relates directly to real world processes such as aeroplane reservation systems, or financial systems. In these cases, a transaction on a computer is associated with events in the real world. For example, 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 practise this means that every transaction must be flushed to disk.

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 an IDE drive at 5400 to 7200 RPM, there can be at most 50 to 70 disk flushes (ie transactions) per second. In many applications this is inadequate.

By contrast Ceda focuses on the management of data that doesn't (in a transactional sense) relate to 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 constrainst can be relaxed a little - 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 is fine for the type of data managed by Ceda.

This allows for excellent performance on hard-disks with no write cache. For example, Ceda currently achieves about 20000 (simple) transactions per second.

TODO for doco

- Provide a simple diagram that compares central server versus peer-peer with data replication. State the basic differences
- Explain that data replication and operational transform immediately leads to the idea that data is for only one process - dramatically easing the need for concurrency control using pessimisitic locking.

1.2.6 Assumptions on the hard-disk

The LSS implementation goes to a number of lengths to avoid certain assumptions about the hard-disk. Many data bases assume one or more of the following

- Disk sectors (typically 512 bytes) are written atomically
- Clusters (often 4k or 8k) are written atomically
- Data is written to the platter in the same order that the write commands are issued in software
- When a file is created with the FILE_FLAG_WRITE_THROUGH option, WriteFile() doesn't return until the data has gone to non-volatile storage.

Ceda avoids making these assumptions.

1.2.7 OID allocations

As a transaction runs, new objects may become reachable and therefore need to be assigned an OID. This happens when container objects are marked as dirty. The POS needs to assign OIDs to the objects that become reachable for the first time. Therefore it is necessary to support OID allocation in `ILSS` rather than `ILSSCommitPhase`. An insignificant downside is that when a transaction aborts we have some wasted OIDs.

1.2.8 Assumed access pattern for reading serial elements

When the Object Store reads a serial element, the stream of bytes is deserialised into a C++ object (and typically won't be contiguous in memory). A *Resident Object Table* (ROT) keeps track of objects that are resident in memory, so there will rarely be a need to read the same serial element twice.

Note that the LSS typically reads a large number of serial elements at a time – because of the course granularity of segments. The LSS assumes that locality within a segment is good and therefore it buffers segments that have been read from disk.

1.2.9 Localisation

When objects are written to disk at around the same time, it is likely that they will be written to the same segment. Therefore the LSS tends to localise objects in a segment according to localisation of mutative changes in time. Objects that were previously together will be drawn apart if one object is modified but not the other.

When a user creates large sub-trees of objects in short periods of time there will be good localisation. If a user tends to make changes all over the place there will be poor localisation.

Cleaning techniques that improve localisation are discussed in the chapter on contexts.

1.2.10 State variables in memory to manage the LSS

The following variables are defined in main memory to manage the LSS.

`SC`	The *Segment Cache*. Provides a cache of a subset of the segments that have been loaded from disk. Also used for segments that are prepared in memory ready to be written to disk. A maximum number of segments in the cache can be specified. A typicaly value would be 32 to provide a 16Mbyte cache. An LRU policy is used to choose segments to be evicted from the cache
`SW`	The *Segment Writer*. Implements the logic for a background thread called the lazy writer that is used to write segments in the SC to disk.
`SUT`	The *Segment Utilisation Table* indicates which segments are free, and the utilisations of segments.
`RPM`	The *Recoverable Packet Map*. Maps from OID to `{SegId,offset}` for every packet in the log. This data structure is only partially loaded into memory.
`lastCheckPoint`	A `LogRecordPosition` that points at the last valid check point position in the log.

1.2.11 Efficient serialisation of the entire LSS

It is possible to serialise the entire LSS to a single stream that is suitable for transmission between computers without the overhead of meta-data like the SUT and the RPM. This simply involves serialising all serial elements. Each serial element is serialised by writing the following

- The OID assigned to the serial element
- The size of the serial element
- The serial data of the serial element

The order in which they are sent doesn't matter apart from maintaining locality.

De-serialisation at the destination involves starting with an empty store and writing the serial elements to segments (breaking them up into packets) and recreating the SUT and RPM at the destination. This can be achieved using one big transaction followed by a check point.

Better performance is gained if the stream is also passed through a compression algorithm.

1.2.12 Excellent transaction throughput

We call information in the SUT and RPM *meta-data* to distinguish it from the "real" data stored in the serial elements.

Meta-data is redundant and this allows the LSS to only write meta-data during check points, and yet still recover information written to the log after the last check point. By performing fewer check points, the overhead of writing the meta-data can be reduced so that efficiency of the LSS is very high – because most of the time it is writing real data. Furthermore, data is always written to a growing log without delays from disk head seeking. This allows the LSS to support a very high transaction throughput.

1.3 Challis' algorithm

Consider that we need to *atomically* write some data to a fixed check point location on disk. One approach is to duplicate the data in two different places and write them in strict alternation. Each mirrored copy (or *dataset*) begins and ends with a *modification sequence number* (MSN).

On recovery, if the MSNs of a dataset are different then it indicates that a crash occurred in the middle of writing that dataset, so it is corrupt and must be ignored. For extra confidence, it is also possible to compute a CRC and store this in each dataset. If both datasets are valid, then we pick the one with the largest MSN.

When writing a dataset it is vital that error codes are checked to be confident that the dataset is written correctly. Otherwise it is possible that both datasets become invalid, and that may be disastrous.

1.4 Storage medium

A *segment* is a 512K byte contiguous block on disk

The *storage medium* contains a set of segments. A segment can be uniquely identified with a 32 bit `SegId`. The `SegId` can be used to compute the seek position of the segment. The underlying I/O system allows for reading and writing a segment (or part of a segment) given its `SegId`. Segments are large so that they can be read/written in any order without I/O performance being dominated by seek times.

No segment will be assigned a `SegId` of zero. Therefore zero can be used to represent a null reference to a segment.

The LSS is stored in a single large file. The file begins with the *root block*. After that are the segments. Initially a single segment is allocated.

The cleaner thread may choose to clean segments at the end of the file if required to allow the file to shrink.

In order to support files larger than 2GB we use the WIN32 file handling routines - ie

- `CreateFile()`
- `SetFilePointer()`
- `SetEndOfFile()`
- `ReadFile()`
- `WriteFile()`
- `CloseHandle()`

These functions support a 64 bit seek position. The LSS manages its own read and write caching strategy, and therefore it is undesirable for the Windows OS to store the LSS file in the system file cache. Therefore the file is created with the WIN32 option `FILE_FLAG_NO_BUFFERING`. This means that all I/O must be at the granularity of disk sectors (typically 512 bytes). Furthermore buffers in memory must be aligned on disk sector boundaries. The latter is achieved by allocating memory buffers using the WIN32 functions `VirtualAlloc()` and `VirtualFree()` because these provide buffers aligned on 4k page boundaries, which is a multiple of the disk sector size.

1.5 The root block

The root block and segments need to support non-buffered I/O - which means that read and write calls on the RAS must be aligned on disk sector boundaries. Usually a disk sector is 512 bytes. Some Asian disks have a sector size of 1024 bytes. Therefore we write at the granularity of 1k byte boundaries.

The total size of the root block is 64k bytes. It consists of the following sections

Description	Size
Root block header - static part	1k
Root block header - dynamic part	1k
1st root block division	31k
2nd root block division	31k
Total	64k

The static part of the root block stores the following information

- The magic string "CEDANET-LSS" used to verify that the file is in fact a Ceda LSS
- The total size of the root block header (currently 2k = 2048 bytes)
- The total size of the root block (currently 64k = 65536 bytes)
- The segment size (currently 512k = 524288 bytes)
- The disk sector size (typically 512 bytes)

The remainder is padded with zeros.

The dynamic part of the header starts on a new 1k boundary so it can be written independently of the static part. It contains a single boolean flag - the Graceful Shutdown Flag (GSF) used during startup to test whether the LSS was shut down gracefully the last time it was opened.

The root block divisions contain the check point information and are written in strict alternation using Challis' algorithm and validated with a 32bit CRC. The divisions start and end on 1k boundaries to ensure they can be written independently.

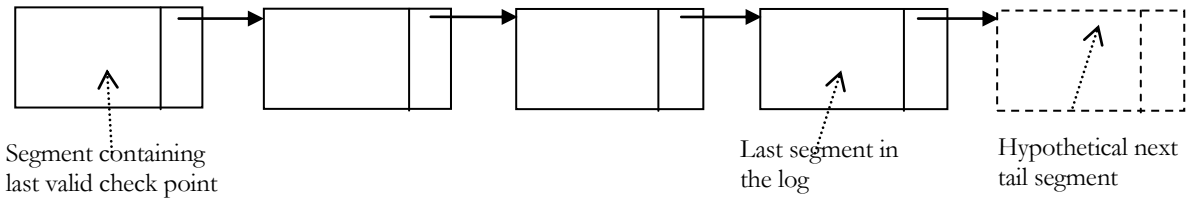
A root block division contains the following fields

Data type	Description
`int m_crc`	32 bit CRC to validate the division
`int m_msn1`	For Challis' algorithm
`SUT m_sut`	A snapshot of the SUT at the time of the last check point.
`RPM m_rpm`	A snapshot of the root RPM-node at the time of the last check point
`LogRecordPosition m_positionOfLastCheckPoint`	Identifies the location of the last check point record in the log
`int m_currentOSID`	
`GUID m_checkPointId`	Each check point is given a unique identity by creating a guid at the time of the check point. This is used to validate flush units during a recovery scan.
`int m_msn2`	For Challis' algorithm

1.5.1 Daisy chaining of segments

A *check point* involves writing data to the log to reduce the time taken for a *recovery*. A recovery involves a scan through the log records in sequence starting from the last valid check point until reaching the end of the log. There is never a need to scan log records in sequence before the last valid check point. Therefore, only the segments on or after the last check point need to be daisy chained. These segments are never freed by the cleaner (because the cleaner is only allowed to work on segments before the last check point).

We regard the segments after the last check point as forming a log in the normal sense, whereas the segments before the last check point are no longer ordered, and behave more like a read only random access data store. This simplifies the cleaner because it doesn't need to worry about a linked list as it frees segments.



Segments are only ever added at the tail of the log.

1.5.2 Log Flush Units

Data is always written to the end of the log using a Log Flush Unit (LFU). An LFU always begins and ends on disk sector boundaries within a single segment. LFUs are written one after the other until the segment is full.

A flush of the log involves

1. Preparing the relevant sectors in memory (within the segment in memory to be flushed)
2. Writing all the sectors to disk.

An LFU begins with a *Flush Unit Header*. This takes up the first 32 bytes of the first sector in the LFU. The header stores the CRC calculated on the header + payload, allowing for validity testing of the LFU. In particular it tests whether the LFU was written in its entirety to disk. This avoids many assumptions about the hard-disk such as atomicity of writing disk sectors, or the order in which sectors are written.

A *Flush Unit Header* containing the following fields

Field	Description
`CRC32 m_crc32`	32 bit CRC calculated on all remaining bytes in the flush unit starting with the checkPointId and ending with the payload
`Guid m_checkPointId`	The number of bytes that have been written to the segment
`FlushSeqNumber m_fsn`	A 32 bit *Flush Sequence Number* assigned when the flush unit was written to the tail of the log.
`int m_numBytesInPayload`	32 bit size of the binary payload in bytes associated with this flush unit
`SegId m_nextSegid`	Identifies the next segment in the forward linked list. Never zero – even for the last segment in the log!

After the header is the binary payload. The payload is zero padded to fill up the last sector in the LFU.

The checkPointId is a guid generated each time a check point is performed, and stored in the root block. This avoids one LSS file accidently recovering LFUs that don't belong to the check point.

The flush unit header contains the identity of the next segment in the sequence. To avoid needing to write this value later, we instead make an early decision on what segment will be used next! This choice is marked in the SUT in memory. Since this value is never zero some other means is required to identify the end of the log. This is achieved using the a strong validation test on flush units during the recovery scan.

During recovery, the system will find the last valid LFU. All the following must be true for a valid LFU.

- The checkPointId is correct
- The FSN corresponds to the next assigned segience number to the LFU
- The size of the payload is reasonable (eg it must not overflow the the segment)
- The NextSegId is a valid segment number
- The last sector is correctly padded with zeros
- The CRC is correct

1.5.3 Recovery

Recovery begins by reading the root block and finding the last valid check point. The check point has an associated check point Id - a 128 bit guid. This is used to validate flush units in the recovery scan.

Now $2^{128} = 3.4 \times 10^{38}$ is a very large number, so the probablity of incorrectly validating a flush unit is very small. In fact, recovering flush units at the rate of 1GHz would take 10^{22} years to give a reasonable chance of seeing randomly generates bytes look like the check point Id.

Note that segments can be recycled without clearing away old data. The system must robustly identify the end of the log. It is assumed that the validity test defined above will have extremely low probablity of making a mistake.

Between check points, it is not possible for segments to be recycled - because of the use of the delta-FSS. Therefore, all segments written with the checkPointId will comprise a single linear list of segments.

After recovery is completed, the store is immediately check pointed. This generates a new check point id, eliminating the chance that previously written flush units will be accidently validated on a subsequent recovery.

Example scenario :

- Store is gracefully shut down. This always involves a check point, and therefore a new check point id will be generated. On startup no recovery scan of the store will be attempted. Also, there is no need to check point the store

on startup. Flush units will be written using the last valid check point. [It will be safer to check point the store anyway - because when the file for the store is copied, we always continue writing flush units with independent guids, avoiding any risk of one store recovering the other's store's flush units]

- Store is not gracefully shut down, a large number of segments have been written since the last valid check point. In the worse case, the sectors may have been written to disk out of order. On startup a recovery scan will be performed. This will validate flush units in turn. The recovery scan ends at the first flush unit with an invalid CRC or checkPointId. This may happen "early" if sectors have been written out of order. If the power fails then recovery will be repeated the next time. After recovery a check point is performed. Only when this is completed successfully will the next recovery start the scan from a different position and use a different check point Id. At that point we don't care about all the old flush units because they will have an out of date check point Id in the flush unit headers.

Between check points segments may be cleaned. However they are only marked as free in the delta-FSS, so there is no chance they will be recycled. Therefore we will never write a segment that already contains flush units with the existing check point id.

1.5.4 Flushing the log

For maximum transaction throughput, we normally only flush a segment when it is completely full. ie we buffer it in memory. Then the entire segment, including the segment trailer is written in one go – with a single call to `WriteFile()`.

Without battery protected RAM to hold the buffered segments, there is a risk of losing data if the system crashes. Therefore, occasionally there may be a need to *flush the log*. This will typically flush a partially full segment to the end of the log. For space efficiency, we support appending additional log records to the same segment at a later time.

The application programmer should be aware that committing a transaction doesn't imply that the log is flushed. The Ceda framework will automatically flush the log so that data never remains in the cache for longer than a few seconds (this is configurable) to reduce data loss on power failure.

The lazy writer is signalled when a full segment is available to be written to disk.

1.5.5 Concurrency notes

Between check points, only free segments can be written to. Therefore we can regard segments as supporting shared read access. As such there is no need to lock segments (say to limit concurrency between a `CedaLock` thread and the cleaner). However we need to be careful between the cleaner and the check pointer. Perhaps the same thread should be used for both. It seems nice for the LSS to hide both of these concepts. This makes the `ILSS` interface small and simple.

1.5.6 Segment Cache (SC)

The *Segment Cache* (SC) is used to cache segments loaded from disk. The RAS doesn't support concurrent loading of segments. Therefore, there is no loss in also limiting the SC to loading one segment at a time.

The SC is also used to hold segments that are being prepared in memory by the *Segment Writer* (SW) before being written to disk. The SW is a client of the SC. When a segment is written by the SW it remains in the cache until it is evicted as the LRU segment.

Clients of the SC need to release the segment after that have finished with it, to parenthesise access to the buffer. This allows the SC to only unload segments that are not being accessed.

Note that the cleaner will have access to an entire segment in the SC. The cleaner needs to be able to iterate through the (live) packets within a segment.

```
class SegmentCache
{
public:
    void Clear();

    // Allocate a free segment initialised with the given SSN.  It is vital that the segment be
    // released with a call to ReleaseSegment() after it is used.
    // It is assumed that this function is called by an LRSWriter - because this function calls
    // AllocateSegment() on the SUT, and the SUT is not threadsafe.
    Segment& AllocateFreeSegment(int ssn);

    // Gain access to the segment with the given SegId.  The segment must be released with a call
    // to ReleaseSegment() after it is used
    Segment& GetSegment(SegId segid);

    void ReleaseSegment(SegId segid);

    // Called by the SUT each time the utilisation of the given segment is changed
    void SetUtilisation(SegId segid, int utilisation);

private:
```

```
    // The segments that are resident in memory, indexed by SegId.
    // NULL indicates that segment is not resident in memory.
    std::vector<Segment*> m_segments;
};
```

The implementation of `LSS::Read(OID oid)` will simply use the RPM to look up the `SegId` and offset of the head packet in the chain, and then use the SC to gain access to the segment with the given `SegId`. It can then directly offset to the packet. A similar approach can be used to find the remaining packets in the chain.

1.5.7 Segment Eviction Queue

The *Segment Eviction Queue* (SEQ) is a doubly linked list of segments, used to queue segments for eviction. The SC has a constraint on the maximum number of segments that may be resident in memory. Once the SC is full, each time we load or write a segment we must first free the segment at the front of the SEQ.

A doubly linked list is a suitable data structure for the SEQ because it supports fast insertion and removal from any position. Each segment in memory has `m_prev` and `m_next` segment pointers to support placement in the SEQ. Both of these pointers are `NULL` if the segment doesn't currently reside in the SEQ. The SC has segment pointers `m_first` and `m_last` to point at the first and last segments in the SEQ. These are `NULL` if the SEQ is empty. `m_first` points at the next segment to be evicted.

When a segment is accessed by a client, it is first removed from the SEQ. This eliminates any chance that it will be evicted while it is used by the client. When the client has finished with the segment it must be released. To allow a segment to be accessed by multiple clients, we have the concept of an access count. When the access count falls back to zero the segment is returned to the back of the SEQ where it is queued for eviction. Note that this has the effect of implementing a Least Recently Used (LRU) segment eviction policy.

Note that an `std::list` is not suitable for the SEQ, because given a segment pointer we need to be able to quickly remove the segment from the SEQ, and that it fast with an std::list when one has an iterator that points at the element to be removed.

The SEQ supports the following functions

```
class SEQ
{
public:
    // Clear the SEQ
    void Clear();

    // These are threadsafe methods that clients can call to gain access then subsequently release
    // access to a segment
    // Each segment stores a private access count that it managed by the SEQ.  When the access count
    // of a segment falls to zero the segment is placed in the eviction queue (where it may be
    // evicted).  When the access count increases from zero, the segment is removed from the
    // eviction queue (to stop it from being evicted)
    void AccessSegment(Segment* s);
    void ReleaseSegment(Segment* s);

    // It is asserted that the given segment is not currently accessed.  Remove it from the queue
    // on the assumption that it is to be deleted from the segment cache
    void EvictSegmentAssumingNotAccessed(Segment* s);

    // Will block if necessary.  Never returns NULL
    Segment* EvictSegment();
};
```

1.5.8 LazyWriterQueue

We simplify the discussion initially by only considering segments that are full.

As segments are filled with data in memory they are passed onto the *Lazy Writer Queue* (LWQ). More specifically, when a segment becomes full the Segment Writer calls

```
LazyWriterQueue::OnFilledSegment(Segment* s)
```

This pushes the given segment onto the back of a FIFO queue. The lazy writer pops (full) segments from the front of the queue by calling

```
Segment* LazyWriterQueue::GetNextFullSegment()
```

and writes them to disk.

The LWQ employs a private critical section in order to be threadsafe - ie to allow one thread (associated with the SegmentWriter) to push segments while another thread (the lazy writer) pops segments. The LWQ can't be involved in a deadlock scenario because all methods return without needing to block on external resources or events.

Segments in the LWQ are assumed to have a positive access count to protect them from eviction by the SEQ. The lazy writer will only release a segment after it is finished with it.

Segments that are full should be written to disk as fast as the lazy writer can go - ie there is no concept of timeout before directing the lazy writer to write segments that are full.

1.5.8.1 Allowing for partially full segments and flushing

The last segment in the LWQ may be partially full. In that case the lazy writer will only write it to disk when the log is flushed. After a Log Record Set (LRS) is written by the Segment Writer, it calls

```
void LazyWriterQueue::SetFlushPosition(Segment* s, int size)
```

This allows the LWQ to remember the partially full segment at the end of the log, and its "size". The size relates to the amount of data in the segment that should be flushed to disk. This may differ from the current size of the segment stored in the segment trailer, because that is updated continually as new log records are written to the end of the log. There is no point flushing data between snapshot records, so the flush position only needs to advance each time a snap shot record is written to the log.

The lazy writer doesn't access the segment trailer of a given segment - ie only the "payload" of the segment. In particular, for a partially full segment the lazy writer will only access the data up to the last flush position set by the Segment Writer. This range of bytes in the segment is immutable - because new log records written by the Segment Writer never overwrite existing log records in a segment. Therefore a flush of the log can proceed in parallel with the Segment Writer.

A single Log Record Set can involve many segments being written to disk. In that case there will be repeated calls to `OnFilledSegment()`, for each segment that is full, followed by a call to `SetFlushPosition()` for the partially full segment at the end of the log. This segment is not added to the queue (yet). However the partially full segment is available in the LWQ for the purpose of flushing the log.

1.5.9 Lazy Writer

The lazy writer employs a low priority worker thread to write filled segments to disk. This allows other threads (that write to the log in memory) to avoid blocking on I/O, unless the segment cache becomes full.

The lazy writer has a Lazy Writer Queue (LWQ) to manage a FIFO queue of filled segments that need to be written to disk. The lazy writer thread pops segments from the front of this queue and writes them to disk. When there are no filled segments, the thread blocks in an efficient wait state on an event. This event is signalled each time a segment is pushed onto the back of the queue.

After a filled segment is written to disk, it is released. This decrements the access count. Typically the access count will fall to zero, allowing the segment to be pushed onto the back of the Segment Eviction Queue (SEQ), in turn allowing it to be evicted from memory.

The worker thread is only interested in writing full segments to disk. It will never itself flush the log (and write a partially filled segment). However the LazyWriter class, does provide a threadsafe Flush() method that a client can use to flush the log. This uses a critical section to avoid contention with other threads trying to flush the log, and the lazy writer worker thread that writes filled segments to disk.

Note that the lazy writer must be explicitly started and stopped. It is an error to start it while it has already been started, and to stop it while it has already been stopped. It is an error to allow the LazyWriter to destruct before it has been stopped.

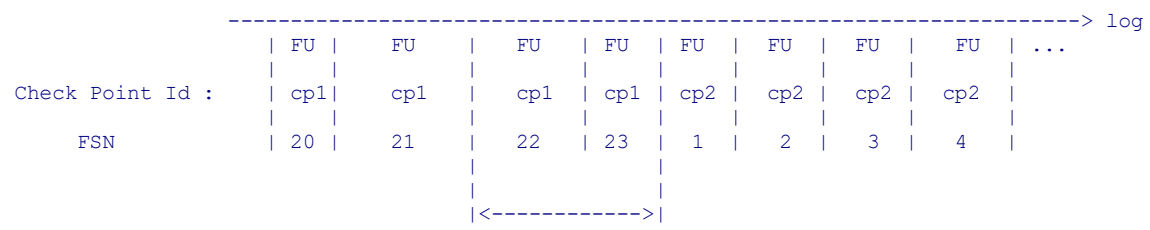
1.5.10 Segment Writer (SW)

The Segment Writer is concerned with tracking the write position of the current segment being prepared in memory. It also is responsible for writing the flush unit headers.

There is a concept of opening a flush unit (which means that some members of the header are initialised, and the write position is offset past the header, ready to write log records for the flush unit payload.

A flush unit must be closed before the next one is opened. When a flush unit is closed the size of the payload can be written to the flush unit header.

Each flush unit stores a check point id and a 32 bit Flush Sequence Number (FSN). The check point is a guid that is generated each time a check point is performed. Consider the following depiction of the log broken up into flush units (FU)



```
Let these be
the FUs written
by a check point
with id = cp2
```

Note the following

- A flush unit stores the check point id corresponding to the last completed check point. The check point id won't change until *after* a check point has been completed.
- The FSN increments with each flush unit written to the log. It is reset back to one for the first flush unit written immediately after a check point.

Note that the flush units written during a check point use the check point id of the previous check point!

1.5.10.1 When to start a new segment

The Segment Writer is responsible for breaking up serial elements into packets.

Serial elements that overflow without good reason will reduce performance. A small serial element that spans two segments reduces localisation which degrades read performance.

Criteria...

Let nFree be the number of bytes remaining in the segment.

Let nSize be the remaining size of the serial element.

If nSize ≤ nFree then we should write the remainder of the serial element as a single packet. If nSize > nFree then the object will overflow. We choose to write part of the serial element to the current segment if nFree > T1 and nSize > T2 where T1 and T2 are suitable fixed thresholds

1.6 Log records

Log records are written to flush units within segments. It is not permissible for a log record to overflow a flush unit or segment.

The physical position of a log record can be specified with the following structure

```
struct LogRecordPosition
{
    SegId segId;       // Identifies the segment
    int offset; // The offset in bytes within the segment
};
```

Give the physical position, it is possible to efficiently seek directly to that location and read the log record.

The following three types of log records are supported

Record	Structure	Description

LR_PACKET	struct PacketHeader
	{
	// Low two bits equals LR_PACKET
	BYTE m_type;
	
	// OID associated with this packet
	OID m_oid;
	
	// Number of bytes in packet
	int m_bufferSize;
	
	// Then follows buffer of serialised
	// data containing
	// exactly m_bufferSize bytes
	
	// If bit 7 of m_type is set then
	// packet finishes with the OID of
	// the next packet in the chain
	};
		A packet contains a contiguous and opaque blob of binary data and is uniquely identified by an OID.
		Packets form forward linked chains.
		Packets are used to build serial elements and RPM packets.

LR_DELETE_PACKETS	struct DeletePacketsHeader
	{
	// Equals LR_DELETE_PACKETS
	BYTE m_type;
	
	// Number of packet chains to be deleted
	int m_count;
		Provides a list of OIDs of packets to be permanently deleted

	`// Then follows array of OIDs. Each`
	`// OID is the head of a packet chain`
	`};`	

`LR_SNAPSHOT`	`struct SnapShotHeader`
	`{`
	`BYTE m_type; // Equals LR_SNAPSHOT`
	`};`
		Commits previously written packet records.

All packet records begin with a byte called `m_type` that indicates the type of record. The low two bits are used to indicate the type

```
LR_PACKET = 0x00
LR_DELETE_PACKETS = 0x01
LR_SNAPSHOT = 0x02
```

When the log record is of type `LR_PACKET`, the following additional bits are used

Bit 7 (mask = 0x80)	If bit set then the packet finishes with the OID of the next packet in the chain
Bit 6 (mask = 0x40)	If bit set then this is the head packet in a packet chain. For robustness, it is preferable to distinguish between packets at the head of their chain from the remaining *overflow packets*. This allows the LSS to detect when an attempt is made to read/write or delete an invalid OID – ie that doesn't correspond to the head packet of a serial element.
Bit 5 (mask = 0x20)	If bit set then this is an RPM packet (as distinct from a data packet)

1.6.1 Log Record Set (LRS)

A *Log Record Set* (LRS) is a set of log records that are written to the log as a unit during an *LRS-op*. An LRS consists of zero or more packet records followed by a snap shot record.

There are three types of LRS-op

- When the POS commits a transaction
- When one or more segments are cleaned
- When dirty RPM-sections are written to the log, prior to a check point

In all cases, an exclusive write lock is gained on the SegmentWriter, so that the LRS can be written without overlapping with another LRS.

1.6.2 Concurrency issues

An LRS-op gets an exclusive lock on the SegmentWriter for the life of the operation. Therefore it is not possible for these operations to overlap.

The SUT and RPM are only modified by LRS-ops. Therefore an LRS-op can't see intermediate states of the SUT or RPM. In particular, a check point will see a consistent SUT and RPM without locking them for an extended period. This ensures that a check point writes a coherent and up to date version of the SUT and RPM to the root block and log. Also, the cleaner will get a coherent and up to date picture of what packets are live and obsolete as it cleans a segment.

The root block and the SUT are only read or written by LRS-ops, so therefore there is no need to protect them with critical sections.

The RPM is only written by LRS-ops, but it can be interrogated outside of an LRS-op. Therefore, it uses a critical section in order to be thread-safe. Locks on the RPM are only granted for short periods.

1.7 Packets

1.7.1 Support for large serial elements using packet chains

The LSS supports arbitrarily large serial elements. Large serial elements are split into smaller variable size pieces called *packets*. A packet is written as a *packet log record* to a segment. A packet can't overflow a segment.

The packets of a given serial element are daisy chained. Each packet is assigned its own OID. A packet stores the OID of the next packet in the chain, or a null OID to indicate the tail packet. No! There is a bit in the first byte of a packet log record that indicates whether there is a next packet in the chain.

The OID of a serial element matches the OID of the head packet in the chain for that serial element.

The RPM and the cleaner work at the level of packets without any regard for how they daisy chain into serial elements.

1.7.2 OID management

1.7.2.1 Detecting dangling references

OIDs are never reused. ie if a serial element is destroyed then its OID may never be assigned to another serial element. This allows the LSS to reliably identify dangling references to persistent objects.

1.7.2.2 Allocating OIDs to new objects

A client that is assigned a value of `OIDhigh` can efficiently allocate OIDs in its private address space by simply incrementing `OIDlow` for each serial element that is created. The RPM stores the next available `OIDlow` for each value of `OIDhigh`

1.8 Snapshots

The LSS moves atomically from one valid snapshot to the next. This is achieved by writing snapshot records to the log. Packet records are only committed when the subsequent snapshot record is found. If no snapshot record is found on crash recovery, the log is truncated.

1.8.1 A transaction gains exclusive access to the log in its commit phase

Given the no-steal policy, we may as well assume that each transaction gains exclusive access to the log in its *commit phase* so it can write all of its modified objects, and the commit record in one go without interleaving log records with other transactions or a check point. With sufficient write caching, a transaction will be able to quickly write data to the cache and return, and avoid being blocked by I/O. There doesn't seem to be any benefit in having multiple transactions all writing to the log at the same time.

Advantages

- There is less risk of data loss, because a transaction is given the freedom to write all its data in consecutive log records. As an extreme example, if lots of transactions write their data in parallel and a crash occurs, it is likely that none will have succeeded in writing all the changes to the log.
- We avoid mixing the data of unrelated transactions, giving us better data localisation – which will improve read performance.
- It simplifies crash recovery. It can be assumed that log records written by a transaction never span a check point, so there is no need to scan log records before the last valid check point during recovery.

1.8.2 Performing a commit phase

When a transaction of the POS enters the *commit phase*, there is every intention of writing the snapshot record. In fact only hardware or power failure can prevent this from happening. Therefore the in-memory RPM and SUT can be updated as packets are written by the SW to the SC.

The commit phase involves the following steps

1. Get an exclusive lock on the log
2. Initialise a temporary vector X of OIDs to empty, used to keep track of the set of packets to be deleted
3. For each serial element to be written (with given OID)
    a. Use the SC to find the current chain C_{prev} of packets used by the serial element (if any). For each packet in C_{prev}
        i. Add the OID to X
        ii. Subtract the size of the packet from the SUT utilisation for the segment containing the packet
        iii. Remove the entry for the packet from the RPM.
    b. Write a chain C_{new} of packets to the log. For each packet in C_{new}.
        i. Add the size of the packet to the SUT utilisation for the segment containing the packet
        ii. Add an entry to the RPM
4. For each serial element to be deleted (with given OID)

a. Use the SC to find the current chain C_{prev} of packets used by the serial element (if any). For each packet in C_{prev}

i. Add the OID to X

ii. Subtract the size of the packet from the SUT utilisation for the segment containing the packet

iii. Remove the entry for the packet from the RPM

5. Write a snapshot log record. This record contains the list X of OIDs of packets to be permanently deleted as part of the snapshot.
6. Release lock on the log

Note that the log is not flushed.

1.9 Segment Utilisation Table (SUT)

The *Segment Utilisation Table* (SUT) is a look up table maintained in main memory and check pointed on disk. It is indexed by `SegId` and yields the following fields

Field	Description
Utilisation size	The total size of all the live packets in the segment
Modification timestamp of youngest live object	The time when the youngest live packet remaining in the segment was last modified (by committing a transaction). This corresponds to the time when it was added to the segment. This is stored in memory but not on disk

The SUT keeps track of free segments. Segments that are free have a utilisation of zero.

The SUT is not particularly large. For example, a 40G store with 512k byte segments has only 80000 entries and takes up 320k on disk (but twice that in memory). Indexing into the map is extremely fast – which is important for maximising transaction throughput.

It is assumed that snapshot log records are obsolete and don't contribute to the utilisation size.

There is no need to log the identity of cleaned segments because a segment is automatically cleaned when its utilisation falls to zero. Recovery involves *replaying history*, so the segments that were originally cleaned by an LRS-op will seen to be cleaned again during recovery.

1.9.1 Should we support very large stores?

With 32 bit SegIds we can support 4 billion segments x 512kbytes per segment = 2000 Terrabytes. However, at that point the SUT will be 4 billion segments x 4 bytes per segment = 16 Gigabytes in size! This is far too much to write to the root block. It also isn't feasible to load the entire SUT into system memory.

It is not clear whether we need to support stores this large. For a start hard-disks aren't this large (today at least!), although it is possible to use a number of physical hard-disks to make a single massive logical hard-disk. However this raises atomicity questions. The very idea to combine multiple hard-disks into a single big hard-disk has other theoretical problems - for example, how do we isolate (ie localise) the effect of data corruption? It would obviously be bad if one failed hard-disk invalidated all the data accross all hard-disks. This suggests that centralising data into a massive persistent store may not be a good idea.

A better, and more scalable approach seems to be to break up data into partitions, and assume that a given process only needs to access a small proportion of the total amount of data. It is only accross all processes and hard-disks that we have a massive data store. This approach has a number of advantages

- We improve parallelism. A given Ceda process is single threaded (at least as far as reading and writing persistent objects goes). Clearly a massive persistent store would become a significant system bottleneck.
- We don't need massive hard-disks, or sets of hard-disks that behave like a single massive hard-disk. This avoids a number of questions about fault tolerance.

It is therefore proposed that for the time being we assume that the SUT is written to the root block. This is a valid approach up to about 100G stores which may be as large as we need anyway.

1.9.2 Allowing for very large stores

In the future we should investigate the idea to break up the SUT into a hierarchical map (in a similar fashion to the RPM). For example, a 4 level map may be suitable, with a maximum fan-out of 256 at each node. The root SUT-node is written to the root block, while the other SUT-nodes are written to the log, in a similar fashion to the RPM. SUT-nodes are associated with serial elements in the LSS and therefore have an OID and need to be indexed by the RPM.

Consider that a check point is performed. Firstly the DOS is processed (if non-empty). This means that the RPM and SUT in memory are up to date with respect to all serial elements. However, we now want to write the dirty parts of the SUT. The problem is that the very writing of the SUT affects the SUT, because previous SUT-sections will be rendered obsolete.

We know up front which SUT sections have been marked as dirty. Consider that we pre-apply the negative offsets to the SUT. The difficulty here is that even this may cause more SUT-sections to become dirty. It is not clear whether the entire SUT will end up being marked as dirty!

During a check point consider that we do the following

1. The DOS is processed (if non-empty), and all the data packets are written to the log being prepared in memory. As this is done the SUT and RPM in memory are changed. Level 0 nodes in both of these hierarchical maps will be marked as dirty.
2. The SUT is put into a mode where further changes (ie instructions to change the utilisation of a given SegId) are stored in a delta SUT but not applied. This means that the SUT will no longer change!
3. The SUT is written to disk, starting at level 0 and working upwards. Level 0 nodes store the actual utilisations, whereas higher level nodes store the OIDs of the nodes on the next lower level. Note that the act of writing SUT nodes will render previous versions of the SUT nodes as obsolete, and this will cause changes to the delta SUT.
4. The RPM is written to disk. This will render previous RPM-nodes as obsolete, and this will cause changes to the delta SUT.
5. The log is flushed
6. The root SUT-node, root RPM-node and delta SUT are written to the root block as part of the check point

When the store is next started, it is necessary to apply the delta-SUT to the SUT to bring it up to date.

1.9.3 The Free Segment Stack (FSS)

The *Free Segment Stack* (FSS) keeps track of the free segments.

The delta-FSS keeps track of the segments that have been freed since the last valid check point. The delta-FSS is only transferred to the FSS at the end of a check point. This avoids the chance of overwriting data on a segment before we are allowed to.

Each time a segment is freed, its `SegId` is pushed onto the delta-FSS. When the SW needs to write a new segment to the end of the log, it pops a free segment from the FSS. If the FSS is empty then an unused `SegId` is "allocated" from past the end of the file.

A suitable data structure for the FSS is a std::deque<> (which is allocated in pages and is ideal for push and pop operations).

It is not clear whether the FSS should be written to disk. It could be created at start-up by scanning the entire SUT.

[TODO: The FSS is not implemented yet]

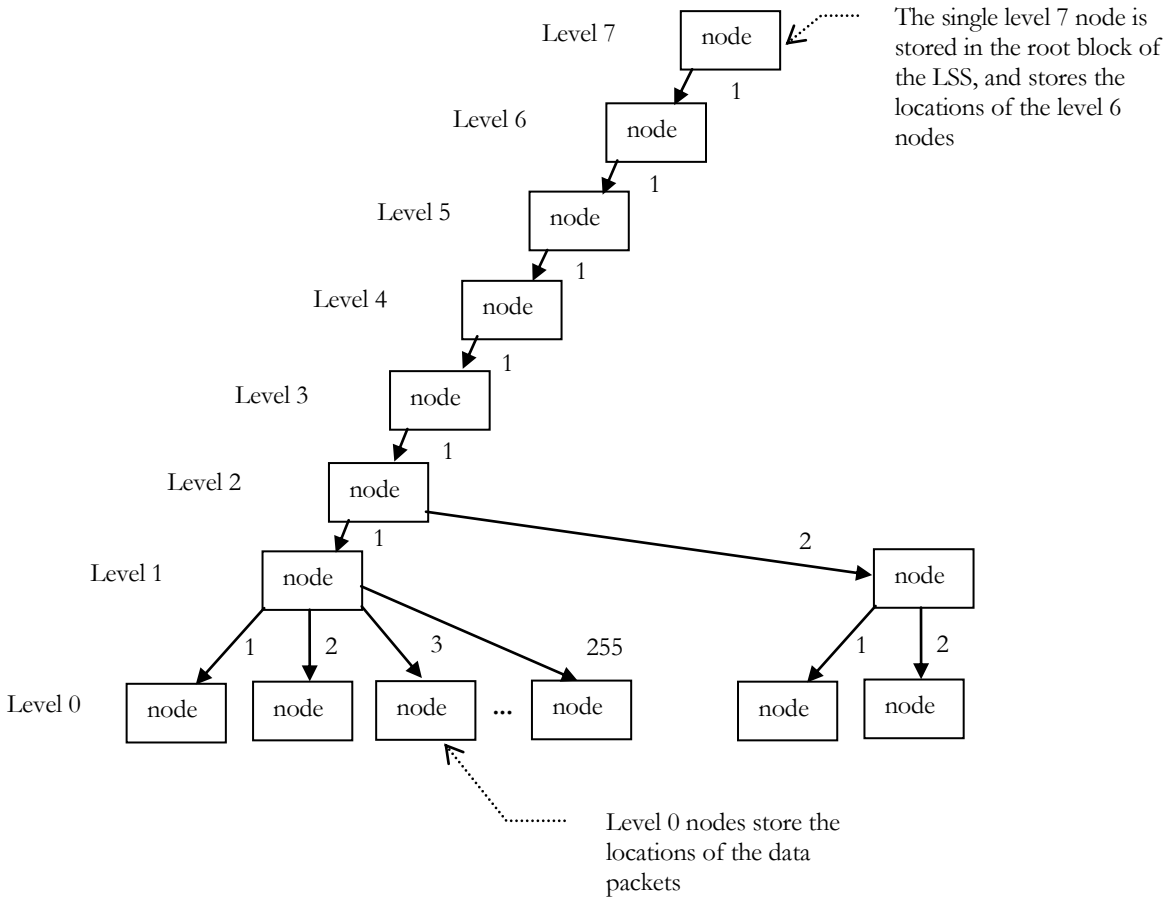
1.10 Supporting overflow of the root block

Given the idea of the FSS, we can "pinch" some segments for the purpose of supporting overflow of the root block. It is important to write and flush the free segments first, then write the root block. In general we must always write the root block last of all because this is what takes us atomically from one check point to the next.

1.10.1 Recoverable Packet Map (RPM)

The *Recoverable Packet Map* (RPM) is a persistent 8 level hierarchical map used to locate the latest versions of recoverable packets given the OID. These 8 levels correspond to the 8 bytes in a 64 bit OID. There is one RPM for the LSS.

Each RPM-node in the search tree stores an array of 255 `LogRecordPosition`., using an index in the range [1,255]. The storage space required is at most 255 x 8 ~ 2K bytes. The levels are labelled from 0 to 7 where 0 is at the "bottom" and 7 is at the "top" of the search tree. There is exactly one RPM-node at level 7. This node is stored in the root block of the LSS. All the other RPM-nodes are stored in the log as indivisible packet log records (indivisible means that the packet can't be broken up into a chain of packets, as done for data packets).



The level 7 RPM-node in the root block stores the current locations of the RPM-nodes at level 6. The RPM-nodes at level 6 store the current locations of the RPM-nodes at level 5. This continues until we get to the RPM-nodes at level 0. These store the current locations of the packets that contain actual data.

OID allocations involve incrementing the next `OIDlow` or `OIDhigh`, so that RPM-nodes are filled in an ordered manner at level 0 (for `OIDlow`) or level 4 (for `OIDhigh`). When a level 0 RPM-node is filled, we create a new level 0 RPM-node under the parent (level 1) RPM-node. If this level 1 RPM-node is full then we create a new level 1 RPM-node under the parent (level 2) RPM-node. This "create RPM-node on overflow" concept continues up through the levels, and relates to counting in base 255. Therefore the fan-out at a given node will often be much less that 255. When an RPM-node is serialised to disk, only the portion of the array of `LogRecordPosition` that are in use is written to the archive.

RPM-nodes are faulted in on demand, because it would not be feasible to load the entire RPM into memory for an LSS with millions or even billions of objects.

RPM-nodes have a concept of being marked as dirty. During the normal operation of the LSS, when data packets are written to the log (followed by snap shot log records), the RPM-nodes at level 0 are updated to track the new locations of the data packets. As this is done, RPM-nodes at level 0 are marked as dirty.

When a check point is performed, the dirty data RPM-nodes at level 0 are written to the log (as packet log records). As this is done, we can update the positions of the level 0 nodes that are stored in the level 1 nodes. This marks level 1 nodes as dirty as required. Then we write the dirty RPM-nodes at level 1. This continues all the way up to level 6. So we see that the RPM must be written to the log in bottom up order (ie level 0 first, ending in level 6). Finally the level 7 RPM-node (which will obviously have been marked as dirty) is written to the root block.

TODO: It will be more efficient to propagate dirtyness up the levels eagerly to avoid the need to recurse through all the RPM nodes in memory when a check point is performed.

Recovery of the RPM is achieved as follows. The level 7 RPM-node is loaded from the root block. This represents a snapshot of the RPM at the time of the checkpoint. We then scan the log to apply further changes to bring it up to date. This works because the log records are self describing. This allows for high transaction throughput because check pointing is performed infrequently and transactions don't need to explicitly log changes to the RPM.

RPM-nodes that are written to the log have an OID. [TODO : Is this actually required?] Like any other packets they can be copied to the end of the log as a segment is cleaned.

During a check point, dirty RPM-nodes are written to the log. This renders previous versions of the RPM-nodes obsolete - in fact an obsolete packet will no longer be referenced by the RPM. It is necessary to subtract the total size of the obsolete RPM packet from the utilisation in the SUT.

TODO: It may make sense for the level 0 RPM nodes to store the size of each data packet. This allows the RPM to be used to make all required changes to the SUT, without needing to go back to the SC. However, that may not be worthwhile because we generally need to process packet chains, so that would need to be cached in the RPM as well.

There are two types of packets

- Data packets
- RPM packets

Let an OID have oidhigh = (c3.c4.c5.c6) and oidlow = (p.c0.c1.c2). The OID space is reserved as follows

	OID pattern mask
Null oid	`0.0.0.0 0.0.0.0`
Level 6 RPM packet	`0.0.0.c6 0.0.0.0`
Level 5 RPM packet	`0.0.c5.c6 0.0.0.0`
Level 4 RPM packet	`0.c4.c5.c6 0.0.0.0`
Level 3 RPM packet	`c3.c4.c5.c6 0.0.0.0`
Level 2 RPM packet	`c3.c4.c5.c6 0.0.0.c2`
Level 1 RPM packet	`c3.c4.c5.c6 0.0.c1.c2`
Level 0 RPM packet	`c3.c4.c5.c6 0.c0.c1.c2`
Data packet	`c3.c4.c5.c6 p.c0.c1.c2`

1.10.1.1 Bad idea : Using the RPM to track which data packets have been loaded

RPM-nodes for levels 1 to 7 keep track of what RPM-nodes at the next (lower) level have been brought into memory. In fact they directly cache an array of 255 memory pointers to RPM-nodes. By analogy, it would seem reasonable for the level 0 RPM-nodes to keep track of what data packets have been brought into memory.

Instead, the *Segment Read Cache* (SRC) keeps track of the segments that have been faulted into memory, and there is no concept of eagerly breaking up a segment into independent packets. For that reason we don't have a C++ class that represents a single packet, and the level 0 RPM-nodes have no idea of what data packets have already been loaded into memory by the SRC.

Interestingly, if packets were indeed represented as C++ objects, and the RPM was used to track which packets have been brought into memory, we would have a significant problem : Segment loading would be unbounded. When a segment is loaded we may read a packet with an OID that requires other segments to be loaded in order to retrieve the relevant RPM nodes (in order to indicate the fact that the packet has been loaded in the RPM in memory). These segments can in turn read packets that require yet more packets to be loaded. In the worse case, a chain reaction could end up loading the entire store into memory!

As a general rule, we favour a design that keeps the (complicated) RPM as simple as possible. This will hopefully reduce the amount of meta-data that needs to be written to the store. For example, the RPM won't be able to tell us how big an existing packet is for the purposes of updating the segment utilisation in the SUT. Only the SRC can give us access to information stored in the packet, such as the total packet size or the next OID in the chain.

1.10.1.2 Extracting bytes from a 32 bit integer

Let a 32 bit integer be written as a.b.c.d where a is the most significant byte and d is the least significant byte. The following masking and shifting code can be used to efficiently extract the bytes from a given 32 bit integer v

```
d = v & 0xFF;
v >>= 8;
c = v & 0xFF;
v >>= 8;
```

```
b = v & 0xFF;
v >>= 8;
a = v & 0xFF;
```

Note that the bytes are obtained in the order from least significant to most significant.

1.10.1.3 Algorithm to index into the RPM

Given an OID the following algorithm is used to index into the RPM

```
Let's put   oidhigh = c3.c4.c5.c6   and oidlow  = p.c0.c1.c2

1.  Extract bytes in the order :  c6,c5,c4,c3,c2,c1,c0,p   (using the above masking and shifting
    algorithm on oidhigh then oidlow)

2.  If c6 = 0 then it is assumed that we have a null oid
    Otherwise c6 identifies a node at level 6 in the form of an index into the children under the
    root node at level 7.

3.  If c5 = 0 then it is assumed that the OID points at the RPM node at level 6
    Otherwise c5 identifies the node at level 5 in the form of an index into the children under the
    node at level 6

4.  If c4 = 0 then it is assumed that the OID points at the RPM node at level 5
    Otherwise c4 identifies the node at level 4 in the form of an index into the children under the
    node at level 5

5.  If c3 = 0 then it is assumed that the OID points at the RPM node at level 4
    Otherwise c3 identifies the node at level 3 in the form of an index into the children under the
    node at level 4

6.  If c2 = 0 then it is assumed that the OID points at the RPM node at level 3
    Otherwise c2 identifies the node at level 2 in the form of an index into the children under the
    node at level 3

7.  If c1 = 0 then it is assumed that the OID points at the RPM node at level 2
    Otherwise c1 identifies the node at level 1 in the form of an index into the children under the
    node at level 2

8.  If c0 = 0 then it is assumed that the OID points at the RPM node at level 1
    Otherwise c0 identifies the node at level 0 in the form of an index into the children under the
    node at level 1

9.  If p = 0 then it is assumed that the OID points at the RPM node at level 0
    Otherwise p identifies a data packet in the form of an index into the children under the
    node at level 0
```

1.11 Check pointing

A check point is performed at regular intervals to reduce the time required for crash recovery. The last valid check point divides the log into two sections

- The records before the last valid check point are referred to as the *check pointed records*.
- The records on or after the last valid check point are referred to as the *recovery records*. These are the records that that are scanned to perform a recovery

The *recovery segments* are the segments that contain recovery records.

A checkpoint writes a coherent, up to date version of the RPM and SUT to the log and root block. Locking of the log ensures that no other thread is making changes to the RPM and SUT as they are check pointed.

Check pointing involves the following steps

1. Begin an LRS-op
2. Write any dirty RPM-nodes to the log
3. Write any dirty SUT-nodes to the log
4. Set `lastCheckPoint` to point at the end of the log
5. End the LRS-op
6. Flush the log
7. Write the SUT, root RPM-node and the position of the check point record to the root block.

After the check point is completed, the cleaner may be allowed to process more segments because the location of the last check point has advanced.

It is vital that the log be flushed to perform a check point. It is not allowable for the root block to reference log records that haven't been flushed yet.

A check point is not performed if the last LRS-op was a check point.

Check pointing is performed by a thread that sits in an infinite loop, performing check point operations then sleeping for a configurable time (say 3 minutes). The check point reduces the amount of recovery work after a crash.

We should flush the log more regularly than we check point to reduce the potential for significant data loss. Flushing every 20 seconds may be suitable.

The following code might be suitable for the check pointing thread

```
while(1)
{
    for (int i=0 ; i < numFlushesPerCheckPoint ; ++i)
    {
        FlushLog();
        Sleep(tmFlushInterval);
    }
    CheckPoint();
    Sleep(tmFlushInterval);
}
```

By sleeping for long intervals, check pointing represents a relatively small I/O and CPU load on the system.

TODO: It would be better to perform a check point after a configurable number of bytes has been written to the log since the last check point – because this relates to the time taken for recovery. During quiet periods check points will be far apart.

1.12 Recovery

An initial implementation of recovery can simply truncate the log at the position of the last valid check point. However this may cause excessive data loss on failure.

A more advanced recovery algorithm scans the log from the last valid check point, because a number of valid snapshots of committed transactions may be present. These must be applied to the RPM and SUT.

It is assumed that the recovery effort is not great, and there is no need to log the recovery process itself. If the system crashes before recovery is complete, then the same work must be repeated the next time the recovery is performed.

It is not allowable for transactions to run while recovery is in progress – because the LSS can't reliably return the most up to date version of a serial element until recovery has brought the RPM up to date. Therefore recovery is completed before the LSS returns the `ILSS` interface pointer, preventing clients from calling any `ILSS` methods while recovery is in progress.

1.12.1 Truncation of the log

When the recovery scan reaches the end of the log, it is possible that a number of packet records have been scanned without the subsequent snapshot record (perhaps because of a power failure before all data was flushed to the log). Before the log can be used, it is necessary to *truncate the log* – or otherwise these packet records may wrongly be realised as part of a subsequent snap shot in another recovery.

If truncation is required, it is performed synchronously at the end of recovery by

1. Rewrite the segment trailer for each segment being removed from the log. Set the `nextSegId`, `size`, and `SSN` to zero. This must be done in reverse order – ie start at the last segment in the log and work backwards.
2. Rewrite the size in the segment trailer of the segment containing the last snapshot or check point record.

1.12.2 Recovery algorithm

Rcovery involves a scan of the log records beginning from the last valid check point.

Let the following structure be used to store information about a data packet

```
struct PacketInfo
{
    LogRecordPosition m_position;  // The position of this data packet (ie SegId and offset)
    OID m_oid;                     // The OID assigned to the packet
    int m_size;                    // The size of the packet in bytes
}
```

Let the PacketList (PL) be a `std::deque<PacketInfo>`, used to store information about all the data packets that have been encountered during the recovery scan.

Let the DeletionList (DL) be a `std::deque<OID>` used to accumulate all the `LR_DELETE_PACKET` records during the recovery scan.

Steps

1. Read the root block, according to Challis' algorithm. Load the SUT, root RPM-node and the segment and offset of the last valid check point
2. Initialise the truncate-log-position to null
3. Initialise the PL and DL to empty
4. Initialise a vector S `SegId` to empty
5. Let `SW.SSN` equal the `SSN` of the segment containing the last valid check point.
6. Scan the log records from just after the last valid check point. For each encountered segment add its `SegId` to S.
    a. If come accross an RPM-packet record then goto 7.
    b. Store each data packet record in the PL
    c. If reach a snapshot record

```
SW.SSN += S.size()
```

        i. Clear S
        ii. For each packet in the PL, add entry to the RPM and apply change to the SUT (ie account for obsoleted packet, if any and the new packet).
        iii. For each OID in the DL, remove the entry from the RPM. Decrease the utilisation of the segment containing the packet by the size of the packet with the given OID.
        iv. Clear the PL and DL
        v. Set the truncate-log-position to just after the snapshot record
7. If the truncate-log-position is not null and it doesn't match the end of the log then truncate the log to that position. S contains the list of segments that need to be removed from the log.

1.13 Cleaning

The cleaner is able to relocate objects without loading them into C++ objects (ie it simply moves the bytes around). It works at the level of the packets. Therefore it doesn't need a `CedaLock`, improving concurrency.

The cleaner uses a LRS-op. It transfers live packets within one or more segments to the end of the log, to allow the segments to be freed (ie returned to the FSS). There is no need to flush the log after cleaning a segment.

It is important to understand that cleaning a segment doesn't write to the segment - either in memory or on disk. On the contrary, it is important that it be left alone! This is because the segment is still needed in case of power failure because it is reachable from the last valid check point. In fact it is not permissible to return a cleaned segment back to the FSS, because that may mean that the segment gets reallocated (saying for writing more data to the end of the log). Instead, it is necessary to wait for the end of the next check point. Only then is the segment ready to be re-used. This gives us the concept of the delta-FSS to track the segments that have been freed since the last valid check point. The delta-FSS is only transferred to the FSS at the end of a check point.

Steps to clean a segment

1. Choose the next segment to be cleaned
2. Access the segment in the SC. This will load the segment if required
3. Open a commit phase on the LSS. This will gain a write lock on the log
4. Scan the packets in the segment, and interrogate the RPM to find out which packets are live. Write the live packets to the end of the log
5. Close the commit phase. This will write a snapshot record, update the SUT, RPM and release the write lock on the log
6. Add the `SegId` of the cleaned segment to the delta-FSS. (Actually this may be implicit - because the utilisation of the segment should have fallen to zero and the SUT detects this transition)

Note that the cleaner only interrogates the RPM during the commit phase. The cleaner sees a stable version of an up to date RPM at this time, so it can't make mistakes about what packets are live.

The cleaner is only permitted to clean segments that come before the last check point. In these segments, only live packet records are useful. Snapshot records have already been taken into account by the last valid check point and are obsolete.

The cleaner can't work on segments that haven't been flushed to disk because the cleaner may only work on segments that come before the last valid check point, and a check point flushes the log.

1.13.1 Choosing the segment to clean

There are significant advantages in prioritising segments to clean, according to the utilisation and the assumed likelihood that the segment will undergo further changes. Cleaning is performed by a low priority background thread that spends most of its time asleep. It wakes up occasionally and checks whether the utilisation of any segments has fallen below a threshold. If so those segments are cleaned.

In practise some packets are "hot" and are changed repeatedly, while other packets are "cold" and are only rarely changed.

A segment is as hot as the hottest live packet remaining within it. It is assumed that coldness of a live packets is proportional to its age. The age of a segment is defined to be the age of the youngest live packet remaining within it (ie the minimum of all the ages).

A hot segment should be cleaned at a lower utilisation threshold than a cold segment because it is likely that the utilisation of a hot segment will continue to fall – and the longer we wait the less unnecessary work that is performed by the cleaner. A particularly hot segment should not be cleaned because it's utilisation may fall rapidly to zero, so the cleaner only needs to remove it from the log.

The Sprite log structured storage uses the following measure to prioritise segment cleaning

quality = benefit/cost = free space generated * age / cost = (1-u) * age / (1+u)

where u is the utilisation (from 0 to 1) of the segment. The "benefit" is proportional to the age because a long age suggests that the cleaning will be long lasting.

When a transaction commits, all packets marked as dirty as part of that transaction are stamped with the current system clock. We record the timestamp of the youngest live object within each segment in the SUT. When a transaction commits, for each dirty object (ie live object that becomes obsolete in a segment), we compare its modification time stamp in the PBT to the time stamp stored in the SUT. The SUT is adjusted if necessary so it continues to provide the modification timestamp of the youngest live object remaining in the segment.

When the SUT is check pointed (ie written to the root block) there is probably no advantage in recording the timestamps. This will help reduce the size of the root block.

When segments are first loaded into memory, the modification timestamp is initialised to the current time. This represents very hot segments that are less likely to be cleaned.

1.14 Compacting the store

The user may want to compact the store to make it as small as possible. This may be useful before the store is sent over the wire or copied to a floppy disk. In-place compaction is desirable.

1.14.1 Iterative method

Steps

1. Disable the cleaning and check pointing background threads
2. Clean all segments with a utilisation below 100%
3. If no segments were cleaned then stop
4. Perform a check point and go to step 2

Segments that are cleaned become available so compaction can be done efficiently in-place.

In the first round of cleaning, all the obsolete packets for RPM-sections and serial elements will be removed. However, the subsequent check point may create new obsolete packets (some of the RPM-sections). These could be anywhere so a second round of cleaning may be required.

In the second round, only segments containing obsolete RPM-sections will be cleaned. This may still cause many objects to be moved to the end of the log. Then another check point is performed. Now there will be obsolete segments in the previous check point. However, we are better off because we have localised RPM-sections to a smaller number of segments.

```
Legend        -   Obsolete packet
              D   Live data packet
              P   Live RPM-section
              C   Check point

                 Segment 1  Segment 2  Segment 3  Segment 4  Segment 5
Initially         ---D---    -DDP---    DDDDDDD    -------    PPD----

Clean 1           DDDPDDD    DDDDPPD
Check point 1     DDD-DDD    DDDD--D    PPPC

Clean 2           DDDDDDD    DDDDPPP
Check point 2     DDDDDDD    DDDD---    PPPC

Clean 3           DDDDDDD    DDDDPPP
Check point 3     DDDDDDD    DDDDPPP    C

Clean 4 – nothing to do so stop
```

1.14.2 Size reduction of the store

If many objects are deleted from the store, it may be useful if the file size drops back to a reasonable level.. This can easily be achieved by repeatedly removing the last segment from the file (cleaning it first if necessary).

1.15 Implementation notes

The entire functionality of the LSS is implemented in the folder Ceda/Sys/LSS.

Currently we don't support recovery scanning of the log. Instead, recovery simply involves going back to the last valid check point.

We cater for Challis algorithm in a generic way, so we get code reuse for the root block and the segment trailer.

A lower layer (RAS) represents the underlying file, and the ability to read and write parts of it. This abstraction is useful because it could be replaced by a raw disk partition. It supports a 64 bit address space to allow for more than 2Gig of data.

The next layer breaks this address space up into the two root blocks and the segments. The ability to add new segments will be provided. Read/write functions to the root block and to a given segment will be provided. Also the ability to allocate new segments by growing the underlying file.

A class to encapsulate the SUT has been written. This is implemented as a vector of records. The SUT expands when extra segments are added to the file. Note that the system allows for the SUT and the size of the file to be out of sync – typically the file could have grown then the system crashed before the SUT has grown and been written to a valid check point. Growing of the file is not logged.

The class LSS encapsulates all functionslity.

1.15.1 IDEA: Atomic append using checksums

Rather than store a size field in the segment trailer, and need to rewrite the segment trailer every time data is flushed to the log, consider that a *flush log record* is always written to the log after each flush operation. When scanning the log on recovery, we only trust data that is bounded by a valid flush log record. A flush log record can't simply store some fixed "magic" values, because segments that are reused already contain previously written flush log records and so there is a risk that data will be trusted by mistake. A flush log record can contain a sequence number and/or checksum over the data in the segment. Note that this approach requires that the recovery scan be resilient to corrupt log records.