Why is CEDA LSS such a great storage engine?

Introduction

A CEDA LSS database is a persistent map from 64 bit keys to serialised byte streams called *serial elements*. The keys are called *seids* (meaning serial element identifiers).

This directly combines two fundamental notions in computing

1. Serial byte streams
2. A heap, involving addressable elements which can be allocated/freed

Serial elements are always read or written as a byte stream from start to finish. They can be any length (from 0 up to many gigabytes). The API for reading/writing serial elements is analogous to the functions fread/fwrite used to read into a buffer or write from a buffer for files.

Seids can be regarded as addresses or pointers to serial elements. There are functions to allocate and free seids which are analogous to the malloc/free functions of a heap.

The serialised bytes within a serial element often contain seids of other serial elements in order to reference them. In this manner a collection of serial elements can form structures like linked lists, trees or graphs. Therefore data structures like B-Trees and hash tables are easily implemented in layers on top of the LSS.

An important difference between the LSS and a key-value store is that both the type and value of the serial element identifiers are defined by the LSS, since they are allocated by the LSS, whereas in a key-value store both the data type of the key and the values of the key are defined by the application. In this sense the LSS is lower level than a key-value store, and indeed a general purpose key-value store is implemented as a layer on top of the LSS.

Since the type and values of seids are defined by the LSS, it is able to employ specialised indexing data structures. The implementation makes use of array indexing within the index nodes, so achieves better performance than a B-Tree. Also seids are allocated sequentially, so under workloads involving creation of many elements, the indexing structures are updated efficiently.

Memory mapping versus serialisation

Some page-based, memory mapped storage engines such as LMDB assume pages on disk are copied verbatim into main memory. There is no need to write any code to serialise the data. On a machine with a 64 bit virtual address space it is reasonable not to employ pointer swizzling. An advantage is the very low processing overhead and elimination of the need for a separate cache.

Unfortunately there is a major disadvantage. Representations which tend to be efficient in-memory do not tend to be efficient on disk. The latency involved with random disk access is often the bottleneck. Even latency to a SSD (about 0.1 milliseconds) represents a few hundred thousand clock

cycles. A program that random accesses main memory 10 million times in one second might take 15 minutes on an SSD and over 24 hours on a HDD.

For a given storage device the product of latency and bandwidth provides a ballpark figure for the amount of useful data which needs to be transferred serially for each random access, in order for the device to spend half it's time "seeking" and half its time actually reading or writing useful data. We call this product the *nominal transfer unit*.

Very roughly we get the following:

Storage device	Access time	Transfer Rate	nominal transfer unit
Desktop HDD	10 ms	100 MB/s	1MB
SSD	0.1 ms	500 MB/s	50 kB
Main memory	100 ns	10 GB/s	1 kB

This huge variation in the nominal transfer unit means that a program can have close to optimal performance for its access patterns for main memory, and yet be woefully suboptimal for its access patterns to disk.

Most software programs tend to allocate large numbers of relatively small memory blocks from the heap. They tend to be far too small for efficient disk access. For example, consider a C++ program making heavy use of std::string whose average size is 100 bytes. Every string object contains a pointer to a heap allocated buffer for the actual characters. Mutations to string objects lead to reallocations from the heap, and over time locality tends to decrease. Typically such programs will behave extremely poorly when the representation is memory mapped to disk.

Serialisation of objects provides a means for multiple objects in main memory which have poor locality, to be serialised to disk as a contiguous block. For example on average an object of type Person might easily involve a hundred heap allocations:

```
struct Person
{
    std::string name;
    std::string address;
    Date dateOfBirth;
    std::set<std::string> favoriteFoods;
};
```

Serialising all the state of a Person object to a contiguous block on disk can make an enormous difference to the performance.

Measured performance

Write performance to RAM disk

Measured ingestion rate of the LSS using a 512 MB ramdisk, in order to measure CPU bottlenecks. In all cases a total of 256 MB of data is written to the store, using a given number of transactions, a

given constant number of serial elements per transaction and a given constant serial element size. Performance drops as the serial element size decreases, because of the overheads of indexing a large number of serial elements.

Test machine: Intel Core i5-3570K @ 3.40-3.80 GHz. 8GB RAM. Windows 7 64 bit. Dataram 512MB RAM disk, format NTFS.

LSS configuration: Segment cache: 32 x 512kB segments. Windows OS file buffering disabled. Write through disabled.

It is concluded that there are three constraints on performance:

1. Max data rate is about 3.1 GB/s
2. Max serial element rate is about 2.8 MHz.
3. Max txn rate is about 2.2 MHz.

Space efficiency relates to the total size of the serial elements divided by the size of the file on disk. As expected space efficiency decreases as the serial element size decreases because of the overheads of the packet log record headers and the RPM.

Num Txns	Serial elts per txn	Serial elt size	Total time (s)	Txn rate (kHz)	Num serial elts	Serial elt rate (kHz)	Disk write rate (MB/s)	Data write rate (MB/s)	Space efficiency (%)
1	1	256 MB	0.1	0	1	-	3064	3058	99.8
1	32 k	8 kB	0.1	0	32 k	395	3100	3088	99.6
32	1 k	8 kB	0.1	0.3	32 k	342	2684	2673	99.6
32 k	1	8 kB	0.1	359.1	32 k	359	2822	2805	99.4
1	64 k	4 kB	0.1	0	64 k	803	3155	3137	99.4
64	1 k	4 kB	0.1	0.7	64 k	710	2789	2773	99.4
64 k	1	4 kB	0.1	699	64 k	699	2757	2730	99.0
1	256 k	1 kB	0.1	0	256 k	1969	1965	1923	97.9
256	1 k	1 kB	0.1	1.8	256 k	1890	1885	1845	97.9
256 k	1	1 kB	0.1	1805.1	256 k	1805	1832	1763	96.2
1	512 k	512 B	0.2	0	512 k	2491	1269	1216	95.9
512	1 k	512 B	0.2	2.3	512 k	2342	1193	1144	95.9
512 k	1	512 B	0.3	1953.9	512 k	1954	1027	954	92.9
1	1 M	256 B	0.4	0	1 M	2545	673	621	92.3
1 k	1 k	256 B	0.4	2.5	1 M	2560	678	625	92.3
1 M	1	256 B	0.5	2190.6	1 M	2191	615	535	86.9
1	2 M	128 B	0.8	0	2 M	2705	385	330	85.8
2 k	1 k	128 B	0.8	2.7	2 M	2721	387	332	85.8
2 M	1	128 B	0.9	2222	2 M	2222	352	271	77.0
1	4 M	64 B	1.5	0	4 M	2817	229	172	75.1
4 k	1 k	64 B	1.5	2.8	4 M	2827	230	173	75.1
4 M	1	64 B	2.0	2068.4	4 M	2068	195	126	64.8
1	8 M	32 B	2.9	0	8 M	2857	145	87	60.2
8 k	1 k	32 B	3	2.8	8 M	2840	144	87	60.1

Write performance to HDD

In this test serial elements between 32 bytes and 16GB are written to a HDD. Performance is typically only constrained by the transfer rate of the HDD (it drops a little when it becomes CPU bound when writing millions of very small serial elements using millions of individual transactions).

For small serial elements the space efficiency begins to fall. This is easily determined by the fact that each serial element has about 21 bytes overhead.

```
Intel Core i5-3570K @ 3.40-3.80 GHz.  8GB RAM.  Windows 7 64 bit.
2TB Western Digitial Black HDD WDC WD2002FAEX-007BA0 format NTFS
32 x 512kB segments. No windows OS file buffering.  No write through.
```

Num Txns	Serial elts per txn	Serial elt size	Total time (s)	Txn rate (kHz)	Num serial elts	Serial elt rate (kHz)	Total bytes	Disk write rate (MB/s)	Effective Data write rate (MB/s)	Space efficiency (%)
1	1	256 MB	2.2	0	1	0	256 MB	117	117	99.8
1	1	1 GB	8.8	0	1	0	1 GB	117	117	100.0
1	1	4 GB	34.1	0	1	0	4 GB	120	120	100.0
1	1	16 GB	136.5	0	1	0	16 GB	120	120	100.0
1	32 k	8 kB	2.2	0	32 k	15	256 MB	116	116	99.6
32	1 k	8 kB	2.2	0	32 k	15	256 MB	119	119	99.6
32 k	1	8 kB	2.2	14.7	32 k	15	256 MB	115	115	99.4
1	64 k	4 kB	2.2	0	64 k	30	256 MB	116	116	99.4
64	1 k	4 kB	2.2	0	64 k	30	256 MB	117	117	99.4
64 k	1	4 kB	2.2	29.5	64 k	30	256 MB	116	115	99.0
1	256 k	1 kB	2.2	0	256 k	118	256 MB	118	116	97.9
256	1 k	1 kB	2.2	0.1	256 k	117	256 MB	117	115	97.9
256 k	1	1 kB	2.3	113.4	256 k	113	256 MB	115	111	96.2
1	512 k	512 B	2.3	0	512 k	231	256 MB	118	113	95.9
512	1 k	512 B	2.3	0.2	512 k	231	256 MB	117	113	95.9
512 k	1	512 B	2.4	217.2	512 k	217	256 MB	114	106	92.9
1	1 M	256 B	2.4	0	1 M	436	256 MB	115	106	92.3
1 k	1 k	256 B	2.4	0.4	1 M	431	256 MB	114	105	92.3
1 M	1	256 B	2.6	410.5	1 M	411	256 MB	115	100	86.9
1	2 M	128 B	2.6	0	2 M	819	256 MB	117	100	85.8
2 k	1 k	128 B	2.6	0.8	2 M	818	256 MB	117	100	85.8
2 M	1	128 B	2.9	726.6	2 M	727	256 MB	115	89	77.0
1	4 M	64 B	2.9	0	4 M	1430	256 MB	116	87	75.1
4 k	1 k	64 B	2.9	1.4	4 M	1431	256 MB	116	87	75.1
4 M	1	64 B	3.8	1101	4 M	1101	256 MB	108	67	62.2
1	8 M	32 B	4.0	0	8 M	2072	256 MB	105	63	60.2
8 k	1 k	32 B	4.0	2.1	8 M	2108	256 MB	107	64	60.1
8 M	1	32 B	6.3	1323.9	8 M	1324	256 MB	89	40	45.5

Comments on performance

The CEDA LSS has extremely impressive performance benchmarks. On an i5 machine it was able to read or write a 500MB database to a RAM drive at the rate of about 3GB/sec.

According to published benchmarks one of the best performing key-value storage engines is LMDB. It would be interesting to compare the CEDA LSS to the LMDB, though that is difficult because the CEDA LSS is lower level. Also, it is questionable to employ benchmarks involving reading/writing millions or billions of small key,value pairs as is commonly done with key-value stores because that isn't representative of many applications and workloads. Consider for example the caching of web pages by a web browser. It is more appropriate to benchmark the reading and writing of hierarchical structures which are appropriate to recording html or xml.

LMDB

We implement data structures like queues, B-trees, hash tables on top of a low level layer which can be regarded as a persistent heap with functions to allocate and deallocate addressable elements. Just as for a memory heap, any number of threads may access the elements, but it is up to the layers above to use mutexes as appropriate. There is no problem with different threads accessing unrelated elements concurrently. Concurrent reading and writing is a normal feature of using the LSS, but unlike MVCC it doesn't ensure readers see consistent snapshots of the entire database. This approach, although low level, allows for a high performance implementation by avoiding the need for writers to use copy on write.

If the existing support for concurrent reading is sufficient for your needs, we are well placed to provide some benchmarks as soon as is convenient for you. However, if you require MVCC, our implementation of this feature is nearing completion. We are working with an ETA of 2nd June 2015.

Strict versus conservative 2PL

The LSS serialises writers eliminating the possibility of dead-lock. Therefore there is no concept of needing to detect cycles and aborting a transaction (the "victim").

In general it is assumed locking is managed in layers on top of the LSS, and dead-lock scenarios must be avoided, just as one normally does when implementing multi-threaded applications.

No aborting transactions

The LSS provides no support for aborting (and hence rolling back a transaction, except for the special case of crash recovery, which respects transaction atomicity and therefore uncommiited transactions are rolled back.

Aborting a transaction isn't necessary when one writes code properly. It is similar to writing exception safe code. E.g. use the copy and swap idiom to assign a variable. If the copy fails then there is nothing to rollback because the target of the assignment is only updated by the swap.

It is better for users of a database to write exception safe code than handles errors and exceptions properly, rather than have the storage engine try to take its own steps to deal with poorly written developer code that ignores correct ways to update variables.

Insertions into very large indexes

The generation of large indexes using either hash tables or B-Trees which don't fit in main memory can result in poor performance because of random accesses to disk for a large proportion of the insertion operations. For an SSD this might limit the system to 10000 insertions per second, or 100 insertions per second for a HDD.

This is essentially the same problem as *External Sorting*. One approach is to

1. Divide all the input data into n parts which individually fit in memory allowing for efficient sorting using QuickSort before being written back to disk.
2. Perform an n-way merge sort of the n sorted files, writing the sorted output in a single pass over the data.

For extremely large n the n-way merge sort can lead to significant random accesses because only a very small amount of each sorted file can be read into memory. It that case it can be beneficial to use more passes over the data.

A B-Tree can utilise a similar approach to improve write performance by taking advantage of merge sorting. Keys to be inserted are in effect buffered so they are applied in much larger batches.

The CEDA LSS doesn't provide any specialised support for external sorting. This however can easily be implemented in layers on top.

Dual license of the LSS

GPL license

Commerical license

There is a nominal commerical license per processor. There are also special deals for clients with special needs.

Rename the LSS

Need a catchy name

MVCC

Really need MVCC on the RPM, so that

1. Readers see a consistent snapshot of the entire database
2. We avoid number-of-thread scalability bottleneck with the mutex protecting the RPM
3. Possibly allow for abortable txns

But this is a significant issue for recycling segments.

COW of the RPM

The RPM will be stored as a DAG. There may be multiple "root" nodes (i.e. RPM7), each corresponding to a different snapshot of the entire database. Root nodes have a ref count for counting the number of readers which are reading that particular snapshot. When the counter falls to zero the RPM7 node can be deleted.

Looking downwards from an RPM7 node we see a normal rooted 8-level tree structure. However looking up is different - a node might have more than one parent node. Therefore the level0 to level 6 nodes have a ref count, which counts the number of parent nodes. When the ref count falls to zero the node can be deleted.

During a writer transaction one of the RPM7 corresponds to the mutable database. This uses COW, so the very first time a serial element is written an RPM0 node must be copied in order to be modified. This triggers copying of parent nodes all the way up to the RPM6 node (the RPM7 node will already have been copied).

The RPM nodes which are dirty are exactly the RPM nodes which have been copied in order to make them mutable. These dirty RPM nodes are written to disk during a check point. Just as with all other data written to the log, it avoids overwriting previous data, except for the root RPM node which is written to the root block in strict alternation. So a reader which is pointing at the root node of an RPM in memory can without any problem see a consistent snapshot of the entire persistent RPM and therefore a consistent snapshot of all objects in the database, even though the RPM is updated on disk during check points.

Double checked locking in C++11

```
std::atomic<Singleton*> Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance() {
    Singleton* tmp = m_instance.load(std::memory_order_relaxed);
    std::atomic_thread_fence(std::memory_order_acquire);
    if (tmp == nullptr) {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_relaxed);
        if (tmp == nullptr) {
            tmp = new Singleton;
```

```
            std::atomic_thread_fence(std::memory_order_release);
            m_instance.store(tmp, std::memory_order_relaxed);
        }
    }
    return tmp;
}
```

Alternative is:

```
std::atomic<Singleton*> Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance() {
    Singleton* tmp = m_instance.load(std::memory_order_acquire);
    if (tmp == nullptr) {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_relaxed);
        if (tmp == nullptr) {
            tmp = new Singleton;
            m_instance.store(tmp, std::memory_order_release);
        }
    }
    return tmp;
}
```

Loading of RPM nodes from disk

Although the readers and the writer can have different RPM7 nodes, they may still share nodes further down the tree.

An RPMi node stores 256 *atomic* pointers to its child nodes.

```
std::atomic<RPMNode*> nodes_[256];
```

We use double checked locking to initialise these as child nodes are loaded from disk. We assume an “immutable” RPM snapshot can only grow, not shrink.

A global mutex for the entire LSS ensures that at most one thread takes on responsibility for loading an RPM node from disk.

We can increase concurrency by using an array of 16 mutexes, and using the low 4 bits of the address of the element of the pointer array to determine which mutex will be used to ensure only one thread takes on responsibility for loading the RPM node.

```
std::lock_guard<std::mutex> lock(mutexArray_[&nodes_[i] & 0xF]);
```

Cleaning versus recycling

The SUT records the utilisation of each segment. Over time the percentage of live data falls. When this reaches 85% the segment might be *cleaned*. Cleaning only refers to the concept of using a writer to rewrite the live serial elements of a segment to the end of the log, so that their utilisation falls to zero. This allows them to be *recycled*.

Even though serial elements have been rewritten to the end of the log, we cannot assume that the old serial elements are redundant yet. It depends on whether they are reachable from any of the snapshots of the RPM for any of the readers, and also whether they might be needed for a crash recovery scan.

TSNs

Since writers are serialised there is a total order in which they execute. Let type TSN mean a 64 bit writer transaction sequence number.

```
typedef int64 TSN;
```

The TSN represents a notion of time as it were over the life of the database (which might be many years). Each TSN corresponds to a snapshot of the entire database. A reader has a TSN which defines exactly which snapshot of the entire database it sees.

API to open and close reader and writer transactions

The following functions are defined on ILogStructuredStore to open transactions:

```
IWriterTxn* OpenWriterTxn()
IReaderTxn* OpenReaderTxn()
```

This allows the API to ensure that reader transactions do not delete/write serial elements. Both types of transactions allow for reading serial elements, and have a Close() method.

TSN of the oldest reader which is open

There is a mutex which is briefly locked when transactions are opened and when they are closed. This mutex protects things like

- The current writer TSN
- A deque<int> which is indexed by an offset TSN. The deque records the number of readers for a given TSN. The TSN offset is such that the 0^{th} element of the deque corresponds to the oldest reader (i.e. the reader with the smallest TSN) which is open.

The deque makes it trivial to calculate the minimum TSN of all the readers which are open. Another useful TSN is the TSN of the second last check point. We do not want to recycle any segments containing serial elements written after these TSNs.

Let Ktsn refer to the smallest tsn of the set of snapshots of the database for which we want to *keep* all the data. Then

Ktsn = min({ r.tsn | r∈readers } ∪ { w.tsn | w is writer of second last check point })

Note that as reader transactions are closed or as check points are performed Ktsn monotone increases.

The readers which potentially need to access a segment

Consider a version of a serial element appearing in a segment. Let it have been written by a writer with TSN tsn1. Let the next writer to have rewritten that serial element have TSN tsn2. Then the range of TSNs of readers which potentially need access to the version of the serial element is given by

[tsn1,tsn2)

Consider a segment whose utilisation has fallen to zero. Then we can consider the union of the [tsn1,tsn2) over all the serial elements in the segment. This tells us which readers potentially need access to the segment.

For a given segment, let ubTsn be the maximum over all the tsn2 values of the serial elements in the segment. We can be certain that readers with tsn $\geq$ ubTsn do not need access to the segment. So as long as the Ktsn $\geq$ ubTsn the segment can be recycled.

SUT

Currently the SUT is a persistent map from segid to the total number of bytes of live data in the segment.

Consider that in addition the SUT maps segid to the TSN of the last writer that made a serial element in the segment obsolete (by writing a new version to the end of the log).

This is updated as follows: When a writer having TSN = writerTsn rewrites a serial element of 'size' it must use the RPM to look up the old log position {segid,offset} of the serial element. That allows it to update the persistent SUT as follows

```
SUT[segid].utilisation -= size;
SUT[segid].ubTsn = max(SUT[segid].ubTsn , writerTsn);
```

When the utilisation of a segment falls to zero the segid is queued for recycling by adding the (segid,ubTsn) to the PRSQ.

The Pending Recycle Segment Queue (PRSQ)

Currently the SUT has a member deltaFss_. This will be replaced by the PRSQ which is also used to delay the recycling of segments.

Segments whose utilisation has transitioned to zero and pending recycle are added to the PRSQ. This queue records a list of (segid,ubTsn) entries ordered by ubTsn.

```
struct Entry
{
```

```
    SegId segid;        // segment to be recycled
    TSN ubTsn;          // TSN of last writer to rewrite a serial element rendition in the segment
};
```

As reader transactions end, and Ktsn increases, the segments with ubTsn $\leq$ Ktsn are removed from the PRSQ and inserted into the FSS, making them available to the segment writer.

Free Segment Stack (FSS)

This is just a std::deque<SegId> stored in the SUT. The FSS doesn't persist on disk. Instead when the LSS is opened the entire SUT is scanned to find which segments have zero utilisation. For a 1TB store with 1MB segments this involves scanning through a million entries which is reasonable given that the SUT is well clustered on disk.

It might be more appropriate to use a std::set for the FSS, so there is a tendency to fill segments near the start of the file, and also to order the segids.