Ceda Log structured store (LSS)

David Barrett-Lennard
16 Oct 2016

Introduction

The Log Structured Store (LSS) is used to persist all the objects in the OODB as binary blobs.

The LSS has all the features required for an industrial strength storage system. In particular it fully supports transactions and guarantees their atomicity in the face of power failures.

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

The LSS has excellent performance. Typically write performance is at least twice as fast as other systems such as BTrieve or ObjectStore, often much faster.

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

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

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

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

The Recoverable Packet Map (RPM) is an 8 level hierarchical map keyed by 64 bit OID used to record the locations of blobs in the store. The Segment Utilisation Table (SUT) records the utilisation of every segment in the LSS. Both of these data structures are only updated on disk during a check point. Check points are performed after writing 64MB to the store.

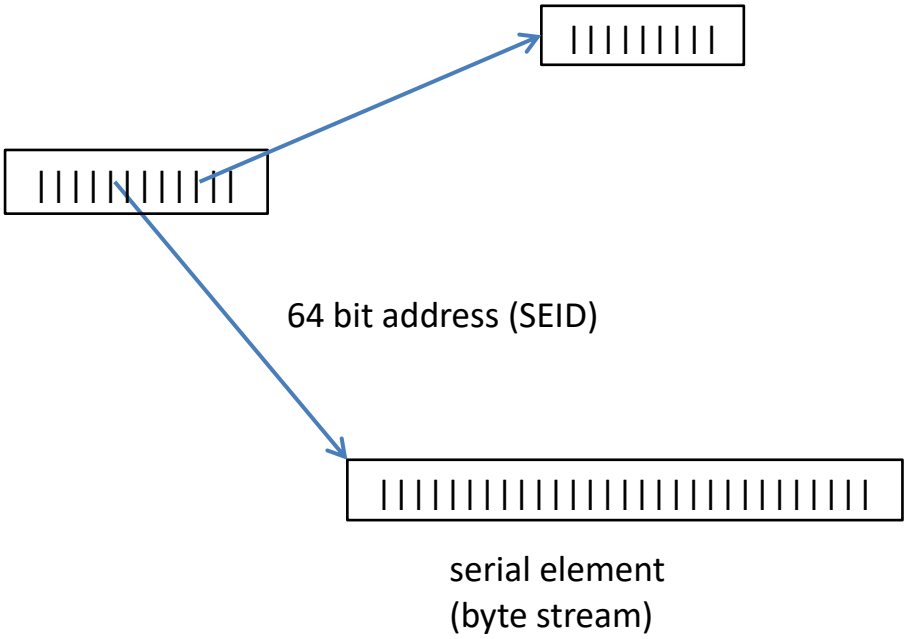
Features

- Up to 500TB in a single store
- Atomic transactions
- Concurrent reads
- Optional durability
- Not WAL
- Lighting fast

- Hotstandby, incremental backup
- Bounded recovery (<1sec)
- Self cleaning (defragmentation)

LSS is a persistent heap

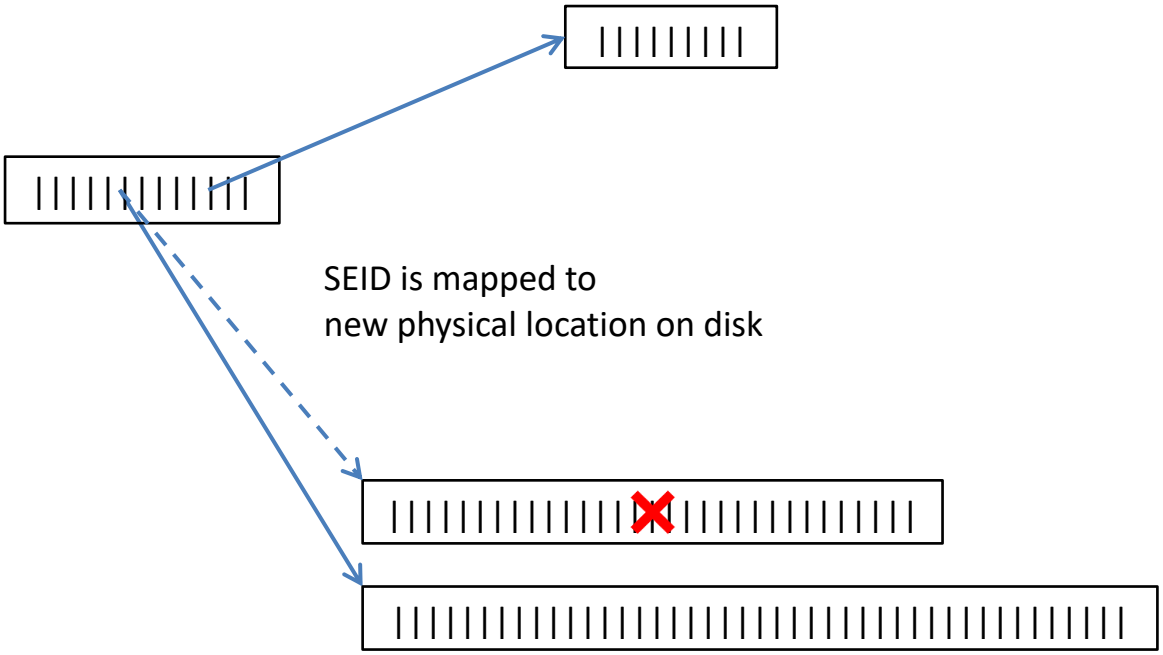
The LSS is a ***persistent heap***. There are ***serial elements*** which are byte streams identified by 64 bit identifiers called ***SEIDs*** (Serial Element Identifiers).



Typically serial elements reference other serial elements by storing SEID values in the byte streams.

SEIDs are virtual addresses

SEIDs are "virtual" or "logical" addresses, not physical addresses, allowing serial elements to be written to a new location on disk.



Four operations

There are just four basic operations on the LSS, to create, delete, write and read serial elements. Serial elements are always written (and normally) read in the entirety from start to finish.

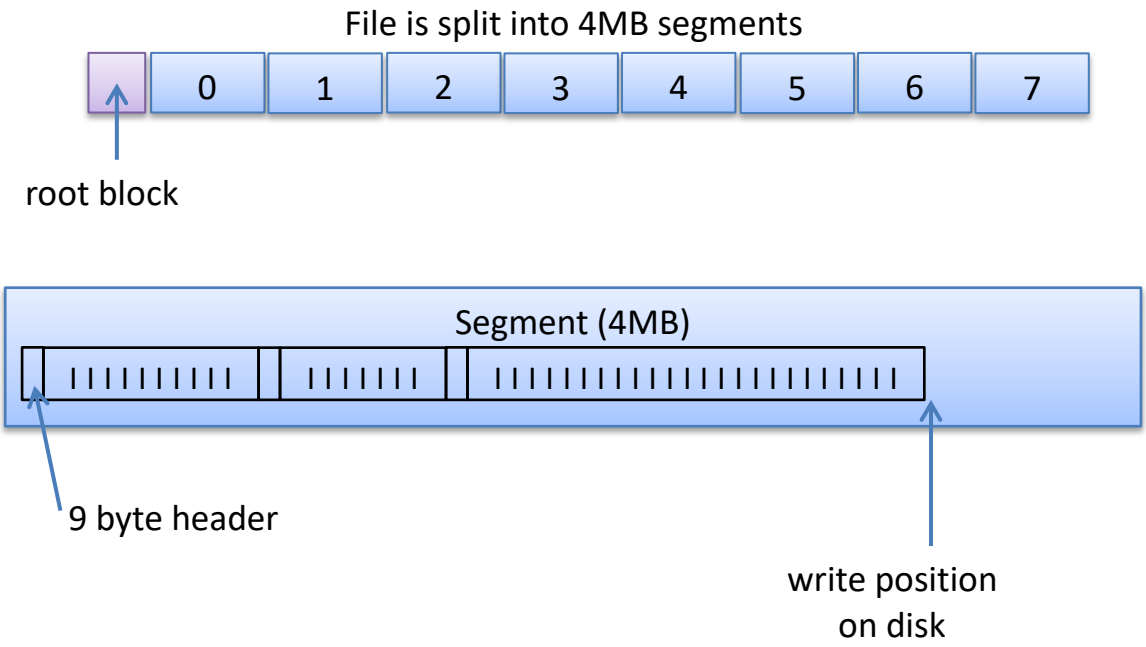
- Create serial element
- Delete serial element
- Write serial element
- Read serial element

In terms of function signatures:

```
SEID create();
void delete(SEID);
OutputStream write(SEID);
InputStream read(SEID);
```

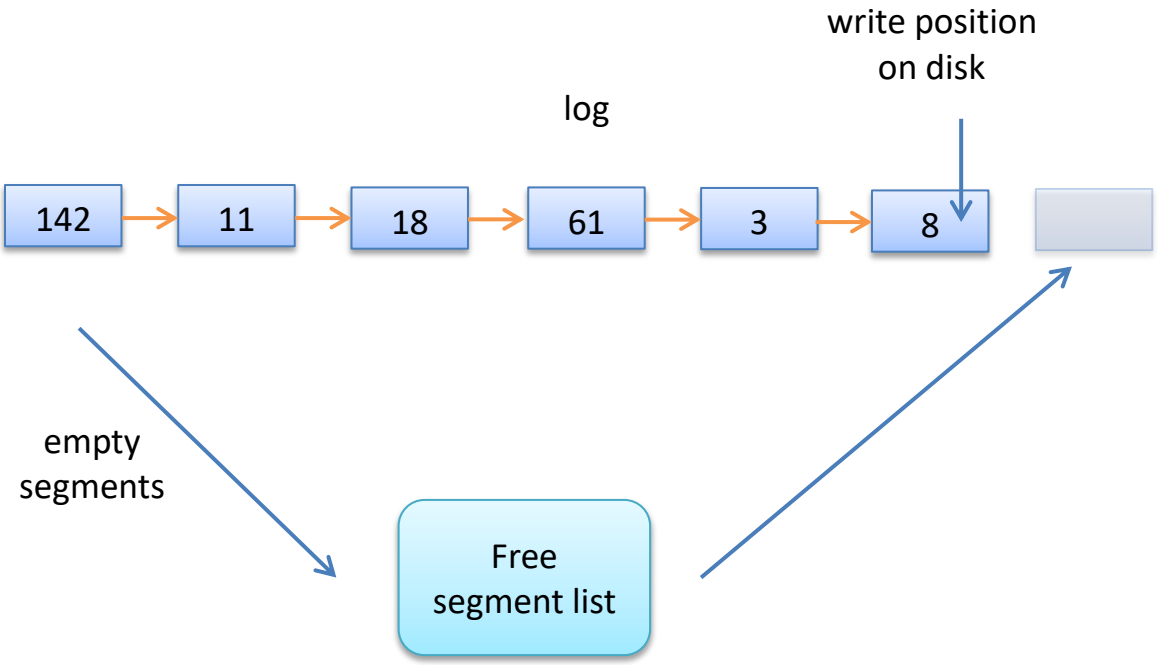
High performance writing

Serial elements are written one after the other to ***segments***



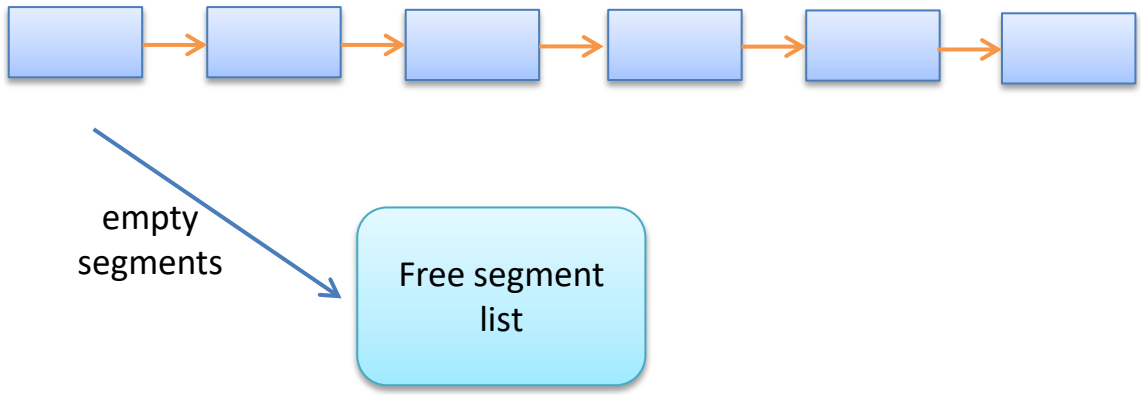
Forward chaining of segments

Segments are forward chained to allow for a crash recovery scan if the store is not closed properly.



Cleaning segments

Segments are cleaned automatically when their space utilisation falls below a threshold. If segment utilisation falls below 80% then the remaining serial elements are copied to the end of the log and the segment is recycled (put into a free segment list).



Performance test by Runge

Comparison to Btrieve in 2005:

- Write block model: 20x faster (2min to 6sec)
- Read block model: 7x faster

> *Got some timing on LSS and Btrieve (BT) for 7.7 & 8.0. I've had some help from Fractal and the summary is that LSS is performing very, very, very well.*
>
> *You ask, is there a LSS cache? Well indeed there is. You have to set it to a fixed size (we would probably give this option to the user) when opening the database. I have only tried this in 8.0 and the results are very exceptional for LSS.*
>
> *Putting LSS in 8.0 is a no brainer even though it will take a month or so of work.*
>
> (engineer at Runge)

Comparison to BerkeleyDB

The following are the results of a comparison of the write performance of the CEDA LSS against Oracle's BerkeleyDB which is another embedded DB.

The test involved writing a million (key,value) pairs using 1000 atomic transactions (not flushed) on a laptop which is an x64 Windows 7 platform with a pair of SSDs in RAID0. In both cases the database used a 512MB cache.

With 4096 byte mapped values, BerkeleyDB took 2 minutes 20 seconds and had over 200% disk space overhead, and CEDA LSS took 4 seconds and had less than 1% disk space overhead. Results for other sizes of the mapped values are tabulated below. Also shown are results for a B+Tree implemented on top of the CEDA LSS.

BerkeleyDB

Mapped value size	Time (sec)	Database size	Log files size	__db files size	Effective Rate (MB/sec)	Disk space wastage factor

4	3.78	30416896	178257920	550969344	3.03	63.30
8	3.80	34021376	188743680	550969344	4.02	48.36
16	3.83	45162496	199229440	550969344	5.98	33.14
32	4.21	72228864	251658240	550969344	9.06	21.87
64	4.44	102146048	314572800	550969344	15.46	13.44
128	5.49	230932480	503316480	550969344	23.62	9.45
256	7.36	421306368	828375040	550969344	34.21	6.82
512	11.9	678215680	1342177280	550969344	41.67	4.94
1024	30.9	1756733440	2967470080	550969344	31.85	5.11
2048	66.4	8226021376	2380267520	550969344	29.53	5.43
4096	140	8226021376	4424990720	550969344	27.96	3.22
8192	212	16418021376	8671723520	550969344	36.89	3.13

CEDA LSS

Mapped value size	Time (sec)	Database size	Effective Rate (MB/sec)	Disk space wastage factor
4	0.66	25690112	17.3	2.1408
8	0.66	29360128	23.1	1.8350
16	0.66	37748736	34.7	1.5729
32	0.67	53477376	56.9	1.3369
64	0.69	85458944	99.5	1.1869
128	0.74	149422080	175.3	1.0987
256	0.81	277348352	310.8	1.0506
512	1.01	533725184	491.0	1.0264
1024	1.09	1045430272	902.9	1.0130
2048	1.64	2069364736	1195.6	1.0065
4096	3.99	4117757952	980.9	1.0034
8192	10.1	8213495808	774.3	1.0016

CEDA B+Tree implemented on top of the LSS

Mapped Value Size	Time (sec)	Database Size	Effective date rate (MB/s)	Disk space wastage factor
4	0.26	12582912	44.0	1.0486
8	0.26	16777216	58.7	1.0486
16	0.27	24641536	84.8	1.0267
32	0.30	40370176	127.2	1.0093
64	0.38	72876032	180.7	1.0122
128	0.50	136839168	259.4	1.0062
256	0.76	264765440	331.3	1.0029
512	1.35	520617984	367.3	1.0012
1024	2.24	1032847360	439.4	1.0008
2048	3.86	2057830400	508.0	1.0009
4096	6.87	4106747904	569.7	1.0007
8192	14.7	8217165824	532.0	1.0021