Backup and hot standby for the LSS

11 Sep 2008
David Barrett-Lennard

Conventional database systems

Most database systems allocate an area of a hard-disk for the actual data, typically organized into *pages* as well as a separate *transaction log* (often on a separate disk) used to record the changes made by each transaction. The log allows for

- A transaction to be aborted (and therefore rolled back), by undoing all the changes recorded in the log.
- Efficiently making a transaction durable by flushing all the changes to the log without needing to flush the associated changes to the data pages. This assumes that during system recovery it may be necessary to (re)do the changes specified in the log.
- Rolling back transactions that were never committed during system recovery, by undoing all the changes recorded in the log.

Such systems support atomicity and durability of transactions, but only assuming *Write Ahead Logging* (WAL) is used: The changes made by a transaction must be recorded in the transaction log *before* the corresponding changes are written to the actual data pages.

Overview of the LSS

The Log Structured Store (LSS) is a high performance binary object store that features transactional writing (with atomicity) and concurrent read access.

By contrast to conventional database systems, the LSS *identifies the transaction log with the actual data*, eliminating the need to write changes to disk twice and in fact eliminating the need for WAL. This allows all writing to be clustered by using very coarse units of I/O (by default the LSS uses 512k byte segments).

Conceptually the LSS always writes new objects to the end of a linearly growing log file, and provides an efficient index called the RPM to record the current locations of all objects. Similar techniques were invented for *Write Once Read Many* (WORM) drives, where there is no option to be able to rewrite previously written objects at their original location.

When an object changes, rather than rewrite it in its original location, the LSS instead writes the new version of the object at the end of the log, and updates the RPM to point at the new location. The previous rendition of the object (at an earlier position in the log) is rendered obsolete. The LSS employs an automated cleaning process working at the granularity of the segments to recover areas of the disk containing very few live objects. A segment is cleaned by rewriting the live objects within the segment to the end of the log.

It follows that the LSS is very well tailored to the ability to record a separate and independent set of log files (so called *multiplexing* of the writing of the log) and to allow these to be applied to a back up LSS store as an efficient delta. This is exactly what's needed to support hot standby and incremental backup. In the literature this technique is sometimes called *log shipping*.

Features of the support for delta log files

- The delta log files can be "played" up to many different positions, allowing for restore to many different snapshots of the entire database that occurred in the past.
- The deltas are very efficient – even for extremely large (multi terrabyte) stores.
- The LSS will ensure that only the right delta is applied to a given store. ie it is completely "idiot proof".
- The backup / hot-standby system is fully compatible with 24x7 operation of a store which continuously reads/writes large amounts of data. This includes applications that are write bound for prolonged periods.
- The backup system use the existing write cache and lazy writer thread internal to the LSS, so enabling support for writing the delta files has minimal impact on the transaction throughput.
- If there is a "transient" failure (such as a power failure) while deltas are applied to a level 0 backup, the backup will recover to a consistent state. ie deltas are applied to the backup *atomically*. Only low level errors in the hard-disk can lead to corruption of the backup.

Functional specification

The LSS optionally supports hot standby and incremental backup.

When the LSS is created or opened, a path to a directory can optionally be provided. LSS *delta-files* will automatically be written to this directory. These files are named as follows

```
nnnnnn.lssdelta
```

where `nnnnnn` is a sequence number, called a *Check Point Sequence Number* (CPSN).

The current delta file being written is named

```
nnnnnn.partial
```

This is only renamed `nnnnnn.lssdelta` after it is completed.

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

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

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

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

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

Delta files respect check point boundaries. Note in turn that check point boundaries respect both flush unit and transaction boundaries. Therefore delta files can in practice be larger than 64MB.

Delta-files store time stamps (as a 64 bit time in 100 nanosecond units since Jan 1st 1600), to allow the back-up system to restore to a given epoch.

Check point identifiers

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

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

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

Hot standby configuration

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

A level 0 backup is not something one would want to do on a regular basis. There are two significant problems with making a complete copy of the store

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

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

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

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

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

Restore using LssApplyDeltas.exe

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

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

LssApplyDeltas level0path deltasDirPath [cpsn2]

The arguments are as follows

Argument 1	level0path	The path to an existing LSS store, called the "level 0"
Argument 2	deltasDirPath	the path to the directory containing the delta-files
Argument 3 [optional]	cpsn2	A "one past end" value of the cpsn, to specify what delta files should be applied to the level 0. The half open interval [cpsn1, cpcn2) is applied. cpsn1 is determined automatically from the level 0 file. Note that delta files are applied up but not including cpsn2.

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

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

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

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

LssCompare.exe

This console application can be used to compare two LSS stores to see if they are (logically) equal - ie as a mapping from OID to byte stream.

Command line:

```
LssCompare path1 path2
```

where path1 and path2 are paths to two different LSS files to be compared.

This is useful for validating the backup system.

Example

The following approach continually validates the delta files by applying them to a level 0 and then in turn running the application defined data validation. This provides a lot of confidence in the ability to restore from backup.

Hardware

- A primary machine (which we call M1) with
    - a dedicated HDD for the main LSS store
    - a dedicated HDD for the LSS delta files.
- a secondary (standby) machine (which we call M2) with
    - a dedicated HDD for the main LSS store
    - a dedicated HDD for the LSS delta files.
- Offsite backup for the level 0 stores, and the delta files.

Installation procedure

- Run the main application on M1 to generate the original LSS file (typically with extension ‘ced’).
- Shut down the application on M1.

- Copy the LSS file from M1 to M2 and also to the offsite backup system as a level 0 backup file.
- Ensure the main application is configured to write delta files to a (local) dedicated HDD on M1.

While the system runs, a script on M2 repeats the following in an infinite loop:

- Copy completed *.lssdelta from M1 to M2, and also to the off-site backup
- Run LssApplyDeltas.exe (without the cpsn2 parameter) to apply all relevant deltas on M2 to the level 0 backup on M2 to bring it as up to date as possible.
- Locally run the main application on M2 so that it
    - Opens the level 0 backup on M2
    - Runs the main application defined data validation
    - Closes the level 0 backup on M2

This script should automatically notify admin staff when there is a problem.

Each week, the level 0 on M2 is copied to off-site backup.

Restore procedure:

1. If the level 0 on M2 is valid, then restore from M2
2. Otherwise, restore M2 from offsite backup, and allow the script to run that apples all delta files to the level 0 on M2 then runs the data validation. Then restore from M2.

A faster restore can involve making M2 take on the role of M1, as long as it is possible to find a new machine to take on the role of M2 – including storage of an appropriate level 0 and delta files.