Backup and hot standby for the LSS

7 Mar 2006
David Barrett-Lennard

Requirements

- The LSS must *optionally* support *hot standby* and *incremental backup*.
- "Hot standby" has fairly relaxed requirements about how up to date the standby store must be.
- Backup can be configured to provide snapshots in the past. This must involves the use of deltas so that extremely large (many terrabyte) stores are supported efficiently.
- A schema change (if required) must be backwards compatible. ie the new LSS must be able to read old versions of the LSS. Converse is not required.
- The LSS will ensure that only the right delta is applied to a given store. ie it must be "idiot proof".
- The backup / hot-standby system must be compatible with 24x7 operation of a store which continuously reads/writes large amounts of data. The LSS must adequately support applications that are write bound for prolonged periods.
- The backup system should preferably use the existing segment write cache and lazy writer thread, to avoid making the thread with a CedaLock block on I/O.
- 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

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 will be named as follows

```
nnnnnn.lssdelta
```

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

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 "ced" 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 *Check Point Sequence Number* (CPSN) in the "ced" 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 store time stamps, to allow the back-up system to generate a snapshot at a given epoch.

A console application called `applylssdeltas.exe` will be written that is able to apply deltas to an existing LSS store. The command line will be provided with

1. The path to the "ced" file
2. The path to the directory containing the delta-files
3. A mode of operation. One of the following:-
    a. Apply all delta files and exit
    b. Apply all delta files up to a given maximum CPSN then exit
    c. Apply all delta files up to a given date/time stamp then exit
    d. Remain running, applying all delta files as they appear

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

The LSS generates a 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 `applylssdeltas.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 `applylssdeltas.exe` idiot proof.

Hot standby configuration

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

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

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

Consider that we have previously created a "standby" store (by making a file system copy). Let the 24x7 application be configured to automatically create the delta-files

in the normal way. Let `applylssdelta.exe` be run using the command line option that causes it to remain running, and it applies deltas to the "standby" as soon as they become available. At quiescence the standby will match the main store.

Let the LSS write each delta-file with exclusive write access but allow shared read access. This will allow `applylssdelta.exe` to apply individual transactions to the standby, synchronising it with the main store as soon as possible.

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.

Coalescing delta files

A console application will be written that can merge a sequence of delta-files into a single "composite" delta-file, that now represents a *range* of CPSN. The application will confirm that the output CPID from each delta-file matches the input CPID of the next delta-file in the sequence. The composite delta inherits the input CPID from the first delta-file and the output CPID from the last delta-file in the sequence. When the composite is applied to a store, CPID validity testing is done in the normal way. Note that composite deltas themselves can take part in merging with other delta-files.

A composite delta file has a file name of the form

```
nnnnnn-mmmmmm.lssdelta
```

where `nnnnnn` is the first CPSN and `mmmmmm` is the last CPSN.

Note that a directory may contain individual delta-files as well as composite delta-files. `applylssdelta.exe` is able to *automatically* find the files it needs to most efficiently bring a store up to date.

A merged delta may be somewhat smaller than the sum of the sizes of the individual deltas from which it was derived. For example, a serial element (with given OID) may have changed many times, or a serial element may have been deleted. Merging ignores the intermediate changes, and therefore is "lossy" because it is impossible to recover the fine grained shot-shots.

This approach is useful when the backup system should provide course granularity snapshot points in the longer past. For example consider the following requirements:-

- For the last 5 years, record quarterly snap shots
- For the last year record monthly snap shots
- for the last month record weekly snapshots
- for the last week record daily snapshots
- for the last day record hourly snapshots
- for the last day hour record 5 minute snapshots

The ability to merge deltas into composites, as many times as required allows such a backup system to be implemented easily and efficiently.

Implementation details
The LSS will store a *Check Point Sequence Number* (CPSN) in the root block of the "ced" file. The LSS root block already stores a schema number so it is very easy to add this additional state.

In the current implementation, the LSS generates a GUID for each check point, called a *Check Point Identifier* (CPID). LSS segments are written in the form of *flush units*. Each flush unit contains a header with the CPID, a flush unit sequence number, and a 32 bit CRC to validate the flush unit. This allows the LSS to reliably perform a recovery on startup. The LSS root block is written with a CPID.

When a check point is performed the CPID that is generated is stored in the root block, and also in all subsequent flush unit headers that are written to the log.

The lazy writer thread is currently responsible for writing flush units to disk. It is proposed that these flush units are also written verbatim to the currently open delta-file. This is a very simple change to the existing lazy writer thread.

So we see that a delta-file is simply a sequence of flush units. These will all share the same CPID. Note therefore that a delta-file stores the input CPID in the flush unit headers, so we already have the basis for validating the delta-file.

Ideally delta-files won't employ the Windows system file cache, because there is a very low probability that a backup file will be read soon after it has been written, so it is generally better to avoid wasting space for backup files in the system file cache. A suitable approach is to use the RAS implementation to write the delta-files.

When a check point is performed we perform the following steps

- Flush the log. In particular make sure that the lazy writer flushes all remaining data to the currently open delta-file.
- Generate the next CPID
- Write the next CPID in the form of a special "footer" to the currently open delta-file.
- Close the delta-file
- Write the dirty RPM nodes and dirty SUT section nodes to the log. Note that this LSS meta-data is not written to the delta-file.
- Open a new delta-file
- Write the root block

Implementation of applylssdeltas.exe
Given the presence of composite delta-files, there can be multiple ways of applying deltas to the level 0. Similarly there can be multiple ways of generating a composite delta for a given range of CPSN.

The following algorithm finds the optimal set of delta-files to be used

1. Iterate over all delta-files in the directory. Use this to build a directed graph data structure. Nodes correspond to CPSN values. Each edge of the graph corresponds to a delta-file that maps from an input CPSN to an output CPSN. Note that the graph is acyclic. On each edge store a *distance* (which represents the cost of traversing that edge) equal to the size in bytes of the associated delta-file.
2. For each node store an *accumulated distance*, initialised to -1 to indicate that it hasn't been calculated yet.
3. The problem is to find the path with minimum total distance between a given start node and end node. A depth first tree search is used to test every possible path through the graph. There is no risk of an infinite loop because the graph is acyclic. During the traversal the accumulated distance is calculated. This is compared to the value stored at the node. If the current tree traversal path is no better than the stored result it is possible to give up on this part of the tree traversal. We also give up if we reach a node that comes after the designated end node. Otherwise the better (ie smaller) accumulated distance is assigned to the node. Now also during the traversal we store the current path (using a stack of integers that is pushed and popped during the recurse). Each time we reach the end node and find that the accumulated distance is shorter than the existing distance we assign this path to a "best path" variable.

In practice it may not be necessary to optimise the algorithm by storing the accumulated distance at each node, and perform exhaustive searching of the graph.

64 bit distances should be used, because composite delta files could exceed 4GB.

Applying deltas to the level 0

Consider that while a delta is being applied to the LSS there is a power failure. This should be fine because the root block in the level 0 won't have been overwritten, therefore the store will behave as though nothing has changed. The next time the delta is applied, it will simply need to start from scratch. Note that irrespective of the size of the delta it is important that segments not be recycled when their utilisation falls to zero, so that on power failure none of the original data is lost.

After the delta is applied to the LSS, we must close the LSS. A special check point must be performed. This needs to use the output CPID specified by the delta-file, rather than generate one as done in a normal check point.

Note than opening a level 0 backup file using `gfnCreateOrOpenLSS()` runs the risk of check pointing it (even if the file is closed without performing any transaction - because it may recover transactions in a recovery scan), invalidating it as a level 0 backup.

API changes

`gfnCreateOrOpenLSS` will now take an additional argument `deltasDirPath`. If non NULL then it specifies the path to a directory in which to create delta files.

```
cxLss_API ILogStructuredStore* gfnCreateOrOpenLSS(
    const char* path,
    const char* deltasDirPath,
```

```
    bool* alreadyExists,
    const LSSSettings& settings,
    const LSSTraceSettings& traceSettings);
```

The following function merges delta-files in the given directory. The generated file represents the range [cpsn1, cpsn2). Various return error codes are returned.

```
cxLss_API ELssError gfnMergeLSSDeltas(
    const char* deltasDirPath,
    int cpsn1, int cpsn2);
```

The following function applies deltas up to but not including cpsn2 to the LSS. Various error codes can be returned.

```
cxLss_API ELssError gfnApplyDeltasToLSS(
    const char* path,
    const char* deltasDirPath,
    int cpsn2);
```

Support for very large deltas

One approach to achieve scalability is to use an LSS to store a delta. This works well because an LSS is nothing more than a (sophisticated) map from OID to binary blob. A nice feature is that applying a delta to a delta is equivalent to applying a delta to a level zero. Even CPID validity testing works properly.

To allow this, the root block of an LSS must store both an input CPID and output CPID. A fundamental "operation" is to merge two adjacent LSS's.

An LSS can be quite wasteful for small stores. However a delta is about 64 MB, so wastage won't be a problem.

However there are some disadvantages to this approach

- Directly storing deltas will always be more space efficient than using an LSS
- We have previously been thinking that merging deltas doesn't modify the original delta files, whereas applying a delta to a level zero, modifies the level zero.
- Atomicity for merging deltas is overkill if it is not done "in-place"
- While the main store is running it will need independent segment caches (for the main LSS and delta-LSS) which is wasteful on memory.
- The existing deltas contain the snapshot records, allowing any required version of the store to be generated.
- It should be fairly straightforward to use an extra thread in the lazy writer to write data to the delta files, allowing for high performance.

Algorithm for merging very large deltas:-

- Perform a scan of both files, in order to record information about the "live" renditions.
- Perform a second scan of both files, only writing out the live objects to the output file.

This is relatively simple and quite fast. Note that the same approach is able to remove redundancy within a single delta-file. The algorithm easily supports directly merging any number of delta-files without the need to generate intermediate files.

The only problem is the need to store a map of all the live renditions. Making this space efficient would help avoid running out of memory. Note that it will always be possible to develop a more sophisticated (ie scalable) system when the need arises.