Files

The implementation of leveldb is similar in spirit to the representation of a single Bigtable tablet (section 5.3). However the organization of the files that make up the representation is somewhat different and is explained below.

Each database is represented by a set of files stored in a directory. There are several different types of files as documented below:

Log files

A log file (*.log) stores a sequence of recent updates. Each update is appended to the current log file. When the log file reaches a pre-determined size (approximately 4MB by default), it is converted to a sorted table (see below) and a new log file is created for future updates.

A copy of the current log file is kept in an in-memory structure (the `memtable`). This copy is consulted on every read so that read operations reflect all logged updates.

Sorted tables

A sorted table (*.ldb) stores a sequence of entries sorted by key. Each entry is either a value for the key, or a deletion marker for the key. (Deletion markers are kept around to hide obsolete values present in older sorted tables).

The set of sorted tables are organized into a sequence of levels. The sorted table generated from a log file is placed in a special **young** level (also called level-0). When the number of young files exceeds a certain threshold (currently four), all of the young files are merged together with all of the overlapping level-1 files to produce a sequence of new level-1 files (we create a new level-1 file for every 2MB of data.)

Files in the young level may contain overlapping keys. However files in other levels have distinct non-overlapping key ranges. Consider level number L where L >= 1. When the combined size of files in level-L exceeds (10^L) MB (i.e., 10MB for level-1, 100MB for level-2, ...), one file in level-L, and all of the overlapping files in level-(L+1) are merged to form a set of new files for level-(L+1). These merges have the effect of gradually migrating new updates from the young level to the largest level using only bulk reads and writes (i.e., minimizing expensive seeks).

Manifest

A MANIFEST file lists the set of sorted tables that make up each level, the corresponding key ranges, and other important metadata. A new MANIFEST file (with a new number embedded in the file name) is created whenever the database is reopened. The MANIFEST file is formatted as a log, and changes made to the serving state (as files are added or removed) are appended to this log.

Current

CURRENT is a simple text file that contains the name of the latest MANIFEST file.

Info logs

Informational messages are printed to files named LOG and LOG.old.

Others

Other files used for miscellaneous purposes may also be present (LOCK, *.dbtmp).

Level 0

When the log file grows above a certain size (4MB by default): Create a brand new memtable and log file and direct future updates here.

In the background:

1. Write the contents of the previous memtable to an sstable.
2. Discard the memtable.
3. Delete the old log file and the old memtable.
4. Add the new sstable to the young (level-0) level.

Compactions

When the size of level L exceeds its limit, we compact it in a background thread. The compaction picks a file from level L and all overlapping files from the next level L+1. Note that if a level-L file overlaps only part of a level-(L+1) file, the entire file at level-(L+1) is used as an input to the compaction and will be discarded after the compaction. Aside: because level-0 is special (files in it may overlap each other), we treat compactions from level-0 to level-1 specially: a level-0 compaction may pick more than one level-0 file in case some of these files overlap each other.

A compaction merges the contents of the picked files to produce a sequence of level-(L+1) files. We switch to producing a new level-(L+1) file after the current output file has reached the target file size (2MB). We also switch to a new output file when the key range of the current output file has grown enough to overlap more than ten level-(L+2) files. This last rule ensures that a later compaction of a level-(L+1) file will not pick up too much data from level-(L+2).

The old files are discarded and the new files are added to the serving state.

Compactions for a particular level rotate through the key space. In more detail, for each level L, we remember the ending key of the last compaction at level L. The next compaction for level L will pick the first file that starts after this key (wrapping around to the beginning of the key space if there is no such file).

Compactions drop overwritten values. They also drop deletion markers if there are no higher numbered levels that contain a file whose range overlaps the current key.

Timing

Level-0 compactions will read up to four 1MB files from level-0, and at worst all the level-1 files (10MB). I.e., we will read 14MB and write 14MB.

Other than the special level-0 compactions, we will pick one 2MB file from level L. In the worst case, this will overlap ~ 12 files from level L+1 (10 because level-(L+1) is ten times the size of level-L, and another two at the boundaries since the file ranges at level-L will usually not be aligned with the file ranges at level-L+1). The compaction will therefore read 26MB and write 26MB. Assuming a disk IO rate of 100MB/s (ballpark range for modern drives), the worst compaction cost will be approximately 0.5 second.

If we throttle the background writing to something small, say 10% of the full 100MB/s speed, a compaction may take up to 5 seconds. If the user is writing at 10MB/s, we might build up lots of level-0 files (~50 to hold the 5*10MB). This may significantly increase the cost of reads due to the overhead of merging more files together on every read.

Solution 1: To reduce this problem, we might want to increase the log switching threshold when the number of level-0 files is large. Though the downside is that the larger this threshold, the more memory we will need to hold the corresponding memtable.

Solution 2: We might want to decrease write rate artificially when the number of level-0 files goes up.

Solution 3: We work on reducing the cost of very wide merges. Perhaps most of the level-0 files will have their blocks sitting uncompressed in the cache and we will only need to worry about the O(N) complexity in the merging iterator.

Number of files

Instead of always making 2MB files, we could make larger files for larger levels to reduce the total file count, though at the expense of more bursty compactions. Alternatively, we could shard the set of files into multiple directories.

An experiment on an ext3 filesystem on Feb 04, 2011 shows the following timings to do 100K file opens in directories with varying number of files:

Files in directory	Microseconds to open a file
1000	9
10000	10
100000	16

So maybe even the sharding is not necessary on modern filesystems?

Recovery

- Read CURRENT to find name of the latest committed MANIFEST
- Read the named MANIFEST file
- Clean up stale files
- We could open all sstables here, but it is probably better to be lazy...
- Convert log chunk to a new level-0 sstable
- Start directing new writes to a new log file with recovered sequence#

Garbage collection of files

`DeleteObsoleteFiles()` is called at the end of every compaction and at the end of recovery. It finds the names of all files in the database. It deletes all log files that are not the current log file. It deletes all table files that are not referenced from some level and are not the output of an active compaction.