LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values.

build passing build passing

Authors: Sanjay Ghemawat (sanjay@google.com) and Jeff Dean (jeff@google.com)

Features

- Keys and values are arbitrary byte arrays.
- Data is stored sorted by key.
- Callers can provide a custom comparison function to override the sort order.
- The basic operations are `Put(key,value)`, `Get(key)`, `Delete(key)`.
- Multiple changes can be made in one atomic batch.
- Users can create a transient snapshot to get a consistent view of data.
- Forward and backward iteration is supported over the data.
- Data is automatically compressed using the Snappy compression library.
- External activity (file system operations etc.) is relayed through a virtual interface so users can customize the operating system interactions.

Documentation

LevelDB library documentation is online and bundled with the source code.

Limitations

- This is not a SQL database. It does not have a relational data model, it does not support SQL queries, and it has no support for indexes.
- Only a single process (possibly multi-threaded) can access a particular database at a time.
- There is no client-server support builtin to the library. An application that needs such support will have to wrap their own server around the library.

Building

This project supports CMake out of the box.

Build for POSIX

Quick start:

```
mkdir -p build && cd build
cmake -DCMAKE_BUILD_TYPE=Release .. && cmake --build .
```

Building for Windows

First generate the Visual Studio 2017 project/solution files:

```
mkdir build
cd build
cmake -G "Visual Studio 15" ..
```

The default default will build for x86. For 64-bit run:

```
cmake -G "Visual Studio 15 Win64" ..
```

To compile the Windows solution from the command-line:

```
devenv /build Debug leveldb.sln
```

or open leveldb.sln in Visual Studio and build from within.

Please see the CMake documentation and `CMakeLists.txt` for more advanced usage.

Contributing to the leveldb Project

The leveldb project welcomes contributions. leveldb's primary goal is to be a reliable and fast key/value store. Changes that are in line with the features/limitations outlined above, and meet the requirements below, will be considered.

Contribution requirements:

1. **Tested platforms only**. We *generally* will only accept changes for platforms that are compiled and tested. This means POSIX (for Linux and macOS) or Windows. Very small changes will sometimes be accepted, but consider that more of an exception than the rule.

2. **Stable API**. We strive very hard to maintain a stable API. Changes that require changes for projects using leveldb *might* be rejected without sufficient benefit to the project.

3. **Tests**: All changes must be accompanied by a new (or changed) test, or a sufficient explanation as to why a new (or changed) test is not required.

4. **Consistent Style**: This project conforms to the Google C++ Style Guide. To ensure your changes are properly formatted please run:

   ```
   clang-format -i --style=file <file>
   ```

Submitting a Pull Request

Before any pull request will be accepted the author must first sign a Contributor License Agreement (CLA) at https://cla.developers.google.com/.

In order to keep the commit timeline linear squash your changes down to a single commit and rebase on google/leveldb/master. This keeps the commit timeline linear and more easily sync'ed with the internal repository at Google. More information at GitHub's About Git rebase page.

Performance

Here is a performance report (with explanations) from the run of the included db_bench program. The results are somewhat noisy, but should be enough to get a ballpark performance estimate.

Setup

We use a database with a million entries. Each entry has a 16 byte key, and a 100 byte value. Values used by the benchmark compress to about half their original size.

```
LevelDB:    version 1.1
Date:       Sun May  1 12:11:26 2011
CPU:        4 x Intel(R) Core(TM)2 Quad CPU    Q6600  @ 2.40GHz
CPUCache:   4096 KB
Keys:       16 bytes each
Values:     100 bytes each (50 bytes after compression)
Entries:    1000000
Raw Size:   110.6 MB (estimated)
File Size:  62.9 MB (estimated)
```

Write performance

The "fill" benchmarks create a brand new database, in either sequential, or random order. The "fillsync" benchmark flushes data from the operating system to the disk after every operation; the other write operations leave the data sitting in the operating system buffer cache for a while. The "overwrite" benchmark does random writes that update existing keys in the database.

```
fillseq      :       1.765 micros/op;   62.7 MB/s
fillsync     :     268.409 micros/op;    0.4 MB/s (10000 ops)
fillrandom   :       2.460 micros/op;   45.0 MB/s
overwrite    :       2.380 micros/op;   46.5 MB/s
```

Each "op" above corresponds to a write of a single key/value pair. I.e., a random write benchmark goes at approximately 400,000 writes per second.

Each "fillsync" operation costs much less (0.3 millisecond) than a disk seek (typically 10 milliseconds). We suspect that this is because the hard disk itself is buffering the update in its memory and responding before the data has been written to the platter. This may or may not be safe based on whether or not the hard disk has enough power to save its memory in the event of a power failure.

Read performance

We list the performance of reading sequentially in both the forward and reverse direction, and also the performance of a random lookup. Note that the database created by the benchmark is quite small. Therefore the report characterizes the performance of leveldb when the working set fits in memory. The cost of reading a piece of data that is not present in the operating system buffer cache will be dominated by the one or two disk seeks needed to fetch the data from disk. Write performance will be mostly unaffected by whether or not the working set fits in memory.

```
readrandom  : 16.677 micros/op;  (approximately 60,000 reads per second)
readseq     :  0.476 micros/op;  232.3 MB/s
readreverse :  0.724 micros/op;  152.9 MB/s
```

LevelDB compacts its underlying storage data in the background to improve read performance. The results listed above were done immediately after a lot of random writes. The results after compactions (which are usually triggered automatically) are better.

```
readrandom  : 11.602 micros/op;  (approximately 85,000 reads per second)
readseq     :  0.423 micros/op;  261.8 MB/s
readreverse :  0.663 micros/op;  166.9 MB/s
```

Some of the high cost of reads comes from repeated decompression of blocks read from disk. If we supply enough cache to the leveldb so it can hold the uncompressed blocks in memory, the read performance improves again:

```
readrandom  : 9.775 micros/op;  (approximately 100,000 reads per second before
compaction)
readrandom  : 5.215 micros/op;  (approximately 190,000 reads per second after
compaction)
```

Repository contents

See doc/index.md for more explanation. See doc/impl.md for a brief overview of the implementation.

The public interface is in include/leveldb/*.h. Callers should not include or rely on the details of any other header files in this package. Those internal APIs may be changed without warning.

Guide to header files:

- **include/leveldb/db.h**: Main interface to the DB: Start here.
- **include/leveldb/options.h**: Control over the behavior of an entire database, and also control over the behavior of individual reads and writes.
- **include/leveldb/comparator.h**: Abstraction for user-specified comparison function. If you want just bytewise comparison of keys, you can use the default comparator, but clients can write their own comparator implementations if they want custom ordering (e.g. to handle different character encodings, etc.).
- **include/leveldb/iterator.h**: Interface for iterating over data. You can get an iterator from a DB object.
- **include/leveldb/write_batch.h**: Interface for atomically applying multiple updates to a database.
- **include/leveldb/slice.h**: A simple module for maintaining a pointer and a length into some other byte array.
- **include/leveldb/status.h**: Status is returned from many of the public interfaces and is used to report success and various kinds of errors.
- **include/leveldb/env.h**: Abstraction of the OS environment. A posix implementation of this interface is in util/env_posix.cc.
- **include/leveldb/table.h, include/leveldb/table_builder.h**: Lower-level modules that most clients probably won't use directly.