pwnall Merge pull request #945 from xiong-ang:master ... ✓ History

5 contributors    

524 lines (415 sloc) | 18.5 KB

leveldb

Jeff Dean, Sanjay Ghemawat

The leveldb library provides a persistent key value store. Keys and values are arbitrary byte arrays. The keys are ordered within the key value store according to a user-specified comparator function.

Opening A Database

A leveldb database has a name which corresponds to a file system directory. All of the contents of database are stored in this directory. The following example shows how to open a database, creating it if necessary:

```
#include <cassert>
#include "leveldb/db.h"

leveldb::DB* db;
leveldb::Options options;
options.create_if_missing = true;
leveldb::Status status = leveldb::DB::Open(options, "/tmp/testdb", &db);
assert(status.ok());
```

```
...
```

If you want to raise an error if the database already exists, add the following line before the `leveldb::DB::Open` call:

```
options.error_if_exists = true;
```

Status

You may have noticed the `leveldb::Status` type above. Values of this type are returned by most functions in leveldb that may encounter an error. You can check if such a result is ok, and also print an associated error message:

```
leveldb::Status s = ...;
if (!s.ok()) cerr << s.ToString() << endl;
```

Closing A Database

When you are done with a database, just delete the database object. Example:

```
... open the db as described above ...
... do something with db ...
delete db;
```

Reads And Writes

The database provides Put, Delete, and Get methods to modify/query the database. For example, the following code moves the value stored under key1 to key2.

```
std::string value;
leveldb::Status s = db->Get(leveldb::ReadOptions(), key1, &value);
if (s.ok()) s = db->Put(leveldb::WriteOptions(), key2, value);
if (s.ok()) s = db->Delete(leveldb::WriteOptions(), key1);
```

Atomic Updates

Note that if the process dies after the Put of key2 but before the delete of key1, the same value may be left stored under multiple keys. Such problems can be avoided by using the `WriteBatch` class to atomically apply a set of updates:

```
#include "leveldb/write_batch.h"
...
std::string value;
leveldb::Status s = db->Get(leveldb::ReadOptions(), key1, &value);
if (s.ok()) {
  leveldb::WriteBatch batch;
  batch.Delete(key1);
  batch.Put(key2, value);
  s = db->Write(leveldb::WriteOptions(), &batch);
}
```

The `WriteBatch` holds a sequence of edits to be made to the database, and these edits within the batch are applied in order. Note that we called Delete before Put so that if key1 is identical to key2, we do not end up erroneously dropping the value entirely.

Apart from its atomicity benefits, `WriteBatch` may also be used to speed up bulk updates by placing lots of individual mutations into the same batch.

Synchronous Writes

By default, each write to leveldb is asynchronous: it returns after pushing the write from the process into the operating system. The transfer from operating system memory to the underlying persistent storage happens asynchronously. The sync flag can be turned on for a particular write to make the write operation not return until the data being written has been pushed all the way to persistent storage. (On Posix systems, this is implemented by calling either `fsync(...)` or `fdatasync(...)` or `msync(..., MS_SYNC)` before the write operation returns.)

```
leveldb::WriteOptions write_options;
write_options.sync = true;
db->Put(write_options, ...);
```

Asynchronous writes are often more than a thousand times as fast as synchronous writes. The downside of asynchronous writes is that a crash of the machine may cause the last few updates to be lost. Note that a crash of just the writing process (i.e., not a reboot) will not cause any loss since even when sync is false, an update is pushed from the process memory into the operating system before it is considered done.

Asynchronous writes can often be used safely. For example, when loading a large amount of data into the database you can handle lost updates by restarting the bulk load after a crash. A hybrid scheme is also possible where every Nth write is synchronous, and in the event of a crash, the bulk load is restarted just after the last synchronous write finished by the previous run. (The synchronous write can update a marker that describes where to restart on a crash.)

`WriteBatch` provides an alternative to asynchronous writes. Multiple updates may be placed in the same WriteBatch and applied together using a synchronous write (i.e., `write_options.sync` is set to true). The extra cost of the synchronous write will be amortized across all of the writes in the batch.

Concurrency

A database may only be opened by one process at a time. The leveldb implementation acquires a lock from the operating system to prevent misuse. Within a single process, the same `leveldb::DB` object may be safely shared by multiple concurrent threads. I.e., different threads may write into or fetch iterators or call Get on the same database without any external synchronization (the leveldb implementation will automatically do the required synchronization). However other objects (like Iterator and `WriteBatch`) may require external synchronization. If two threads share such an object, they must protect access to it using their own locking protocol. More details are available in the public header files.

Iteration

The following example demonstrates how to print all key,value pairs in a database.

```
leveldb::Iterator* it = db->NewIterator(leveldb::ReadOptions());
for (it->SeekToFirst(); it->Valid(); it->Next()) {
  cout << it->key().ToString() << ": "  << it->value().ToString() << endl;
}
assert(it->status().ok());  // Check for any errors found during the scan
delete it;
```

The following variation shows how to process just the keys in the range [start,limit):

```
for (it->Seek(start);
   it->Valid() && it->key().ToString() < limit;
   it->Next()) {
```

```
  ...
}
```

You can also process entries in reverse order. (Caveat: reverse iteration may be somewhat slower than forward iteration.)

```
for (it->SeekToLast(); it->Valid(); it->Prev()) {
  ...
}
```

Snapshots

Snapshots provide consistent read-only views over the entire state of the key-value store. `ReadOptions::snapshot` may be non-NULL to indicate that a read should operate on a particular version of the DB state. If `ReadOptions::snapshot` is NULL, the read will operate on an implicit snapshot of the current state.

Snapshots are created by the `DB::GetSnapshot()` method:

```
leveldb::ReadOptions options;
options.snapshot = db->GetSnapshot();
... apply some updates to db ...
leveldb::Iterator* iter = db->NewIterator(options);
... read using iter to view the state when the snapshot was created ...
delete iter;
db->ReleaseSnapshot(options.snapshot);
```

Note that when a snapshot is no longer needed, it should be released using the `DB::ReleaseSnapshot` interface. This allows the implementation to get rid of state that was being maintained just to support reading as of that snapshot.

Slice

The return value of the `it->key()` and `it->value()` calls above are instances of the `leveldb::Slice` type. Slice is a simple structure that contains a length and a pointer to an external byte array. Returning a Slice is a cheaper alternative to returning a `std::string` since we do not need to copy potentially large keys and values. In addition, leveldb methods do not return null-terminated C-style strings since leveldb keys and values are allowed to contain `'\0'` bytes.

C++ strings and null-terminated C-style strings can be easily converted to a Slice:

```
leveldb::Slice s1 = "hello";

std::string str("world");
leveldb::Slice s2 = str;
```

A Slice can be easily converted back to a C++ string:

```
std::string str = s1.ToString();
assert(str == std::string("hello"));
```

Be careful when using Slices since it is up to the caller to ensure that the external byte array into which the Slice points remains live while the Slice is in use. For example, the following is buggy:

```
leveldb::Slice slice;
if (...) {
  std::string str = ...;
  slice = str;
}
Use(slice);
```

When the if statement goes out of scope, str will be destroyed and the backing storage for slice will disappear.

Comparators

The preceding examples used the default ordering function for key, which orders bytes lexicographically. You can however supply a custom comparator when opening a database. For example, suppose each database key consists of two numbers and we should sort by the first number, breaking ties by the second number. First, define a proper subclass of `leveldb::Comparator` that expresses these rules:

```
class TwoPartComparator : public leveldb::Comparator {
 public:
  // Three-way comparison function:
  //   if a < b: negative result
  //   if a > b: positive result
  //   else: zero result
```

```
  int Compare(const leveldb::Slice& a, const leveldb::Slice& b) const {
    int a1, a2, b1, b2;
    ParseKey(a, &a1, &a2);
    ParseKey(b, &b1, &b2);
    if (a1 < b1) return -1;
    if (a1 > b1) return +1;
    if (a2 < b2) return -1;
    if (a2 > b2) return +1;
    return 0;
  }

  // Ignore the following methods for now:
  const char* Name() const { return "TwoPartComparator"; }
  void FindShortestSeparator(std::string*, const leveldb::Slice&) const {}
  void FindShortSuccessor(std::string*) const {}
};
```

Now create a database using this custom comparator:

```
TwoPartComparator cmp;
leveldb::DB* db;
leveldb::Options options;
options.create_if_missing = true;
options.comparator = &cmp;
leveldb::Status status = leveldb::DB::Open(options, "/tmp/testdb", &db);
...
```

Backwards compatibility

The result of the comparator's Name method is attached to the database when it is created, and is checked on every subsequent database open. If the name changes, the `leveldb::DB::Open` call will fail. Therefore, change the name if and only if the new key format and comparison function are incompatible with existing databases, and it is ok to discard the contents of all existing databases.

You can however still gradually evolve your key format over time with a little bit of pre-planning. For example, you could store a version number at the end of each key (one byte should suffice for most uses). When you wish to switch to a new key format (e.g., adding an optional third part to the keys processed by `TwoPartComparator`), (a) keep the same comparator name (b) increment the version number for new keys (c) change the comparator function so it uses the version numbers found in the keys to decide how to interpret them.

Performance

Performance can be tuned by changing the default values of the types defined in `include/options.h` .

Block size

leveldb groups adjacent keys together into the same block and such a block is the unit of transfer to and from persistent storage. The default block size is approximately 4096 uncompressed bytes. Applications that mostly do bulk scans over the contents of the database may wish to increase this size. Applications that do a lot of point reads of small values may wish to switch to a smaller block size if performance measurements indicate an improvement. There isn't much benefit in using blocks smaller than one kilobyte, or larger than a few megabytes. Also note that compression will be more effective with larger block sizes.

Compression

Each block is individually compressed before being written to persistent storage. Compression is on by default since the default compression method is very fast, and is automatically disabled for uncompressible data. In rare cases, applications may want to disable compression entirely, but should only do so if benchmarks show a performance improvement:

```
leveldb::Options options;
options.compression = leveldb::kNoCompression;
... leveldb::DB::Open(options, name, ...) ....
```

Cache

The contents of the database are stored in a set of files in the filesystem and each file stores a sequence of compressed blocks. If options.block_cache is non-NULL, it is used to cache frequently used uncompressed block contents.

```
#include "leveldb/cache.h"

leveldb::Options options;
options.block_cache = leveldb::NewLRUCache(100 * 1048576);  // 100MB cache
leveldb::DB* db;
leveldb::DB::Open(options, name, &db);
... use the db ...
```

```
delete db
delete options.block_cache;
```

Note that the cache holds uncompressed data, and therefore it should be sized according to application level data sizes, without any reduction from compression. (Caching of compressed blocks is left to the operating system buffer cache, or any custom Env implementation provided by the client.)

When performing a bulk read, the application may wish to disable caching so that the data processed by the bulk read does not end up displacing most of the cached contents. A per-iterator option can be used to achieve this:

```
leveldb::ReadOptions options;
options.fill_cache = false;
leveldb::Iterator* it = db->NewIterator(options);
for (it->SeekToFirst(); it->Valid(); it->Next()) {
  ...
}
delete it;
```

Key Layout

Note that the unit of disk transfer and caching is a block. Adjacent keys (according to the database sort order) will usually be placed in the same block. Therefore the application can improve its performance by placing keys that are accessed together near each other and placing infrequently used keys in a separate region of the key space.

For example, suppose we are implementing a simple file system on top of leveldb. The types of entries we might wish to store are:

```
filename -> permission-bits, length, list of file_block_ids
file_block_id -> data
```

We might want to prefix filename keys with one letter (say '/') and the `file_block_id` keys with a different letter (say '0') so that scans over just the metadata do not force us to fetch and cache bulky file contents.

Filters

Because of the way leveldb data is organized on disk, a single `Get()` call may involve multiple reads from disk. The optional FilterPolicy mechanism can be used to reduce the number of disk reads substantially.

```
leveldb::Options options;
options.filter_policy = NewBloomFilterPolicy(10);
leveldb::DB* db;
leveldb::DB::Open(options, "/tmp/testdb", &db);
... use the database ...
delete db;
delete options.filter_policy;
```

The preceding code associates a Bloom filter based filtering policy with the database. Bloom filter based filtering relies on keeping some number of bits of data in memory per key (in this case 10 bits per key since that is the argument we passed to `NewBloomFilterPolicy`). This filter will reduce the number of unnecessary disk reads needed for Get() calls by a factor of approximately a 100. Increasing the bits per key will lead to a larger reduction at the cost of more memory usage. We recommend that applications whose working set does not fit in memory and that do a lot of random reads set a filter policy.

If you are using a custom comparator, you should ensure that the filter policy you are using is compatible with your comparator. For example, consider a comparator that ignores trailing spaces when comparing keys. `NewBloomFilterPolicy` must not be used with such a comparator. Instead, the application should provide a custom filter policy that also ignores trailing spaces. For example:

```
class CustomFilterPolicy : public leveldb::FilterPolicy {
 private:
  leveldb::FilterPolicy* builtin_policy_;

 public:
  CustomFilterPolicy() : builtin_policy_(leveldb::NewBloomFilterPolicy(10)) {}
  ~CustomFilterPolicy() { delete builtin_policy_; }

  const char* Name() const { return "IgnoreTrailingSpacesFilter"; }

  void CreateFilter(const leveldb::Slice* keys, int n, std::string* dst) const {
    // Use builtin bloom filter code after removing trailing spaces
    std::vector<leveldb::Slice> trimmed(n);
    for (int i = 0; i < n; i++) {
      trimmed[i] = RemoveTrailingSpaces(keys[i]);
    }
```

```
    builtin_policy_->CreateFilter(trimmed.data(), n, dst);
  }
};
```

Advanced applications may provide a filter policy that does not use a bloom filter but uses some other mechanism for summarizing a set of keys. See `leveldb/filter_policy.h` for detail.

Checksums

leveldb associates checksums with all data it stores in the file system. There are two separate controls provided over how aggressively these checksums are verified:

`ReadOptions::verify_checksums` may be set to true to force checksum verification of all data that is read from the file system on behalf of a particular read. By default, no such verification is done.

`Options::paranoid_checks` may be set to true before opening a database to make the database implementation raise an error as soon as it detects an internal corruption. Depending on which portion of the database has been corrupted, the error may be raised when the database is opened, or later by another database operation. By default, paranoid checking is off so that the database can be used even if parts of its persistent storage have been corrupted.

If a database is corrupted (perhaps it cannot be opened when paranoid checking is turned on), the `leveldb::RepairDB` function may be used to recover as much of the data as possible

Approximate Sizes

The `GetApproximateSizes` method can used to get the approximate number of bytes of file system space used by one or more key ranges.

```
leveldb::Range ranges[2];
ranges[0] = leveldb::Range("a", "c");
ranges[1] = leveldb::Range("x", "z");
uint64_t sizes[2];
db->GetApproximateSizes(ranges, 2, sizes);
```

The preceding call will set `sizes[0]` to the approximate number of bytes of file system space used by the key range `[a..c)` and `sizes[1]` to the approximate number of bytes used by the key range `[x..z)`.

Environment

All file operations (and other operating system calls) issued by the leveldb implementation are routed through a `leveldb::Env` object. Sophisticated clients may wish to provide their own Env implementation to get better control. For example, an application may introduce artificial delays in the file IO paths to limit the impact of leveldb on other activities in the system.

```
class SlowEnv : public leveldb::Env {
  ... implementation of the Env interface ...
};

SlowEnv env;
leveldb::Options options;
options.env = &env;
Status s = leveldb::DB::Open(options, ...);
```

Porting

leveldb may be ported to a new platform by providing platform specific implementations of the types/methods/functions exported by `leveldb/port/port.h`. See `leveldb/port/port_example.h` for more details.

In addition, the new platform may need a new default `leveldb::Env` implementation. See `leveldb/util/env_posix.h` for an example.

Other Information

Details about the leveldb implementation may be found in the following documents:

1. Implementation notes
2. Format of an immutable Table file
3. Format of a log file