Rod Vagg

databases

LevelDB and Node: Getting Up and Running

Posted by Rod Vagg on May 2nd, 2013.

This is the second article in a three-part series on LevelDB and how it can be used in Node.

- Part 1: What is LevelDB Anyway?
- Part 2: Getting Up and Running

Our first article covered the basics of LevelDB and its internals. If you haven't already read it you are encouraged to do so as we will be building upon this knowledge as we introduce the Node interface in this article.

Rod Vagg

There are two primary libraries for using LevelDB in Node, LevelDOWN and LevelUP.

LevelDOWN is a pure C++ interface between Node.js and LevelDB. Its API provides limited ***sugar*** and is mostly a straight-forward mapping of LevelDB's operations into JavaScript. All I/O operations in LevelDOWN are asynchronous and take advantage of LevelDB's thread-safe nature to parallelise reads and writes.

LevelUP is the library that the majority of people will use to interface with LevelDB in Node. It wraps LevelDOWN to provide a more Node.js-style interface. Its API provides more ***sugar*** than LevelDOWN, with features such as optional arguments and deferred-till-open operations (i.e. if you begin operating on a database that is in the process of being opened, the operations will be queued until the open is complete).

LevelUP exposes iterators as Node.js-style object streams. A LevelUP **ReadStream** can be used to read sequential entries, forward or reverse, to and from any key.

LevelUP handles JSON and other encoding types for you. For example, when operating on a LevelUP instance with JSON value-encoding, you simply pass in your objects for writes and they are serialised for you. Likewise, when you read them, they are deserialised and passed back in their original form.

A simple LevelUP example

```
var levelup = require('levelup')

// open a data store
var db = levelup('/tmp/dprk.db')

// a simple Put operation
db.put('name', 'Kim Jong-un', function (err) {

  // a Batch operation made up of 3 Puts
  db.batch([
      { type: 'put', key: 'spouse', value: 'Ri Sol-ju' }
    , { type: 'put', key: 'dob', value: '8 January 1983' }
    , { type: 'put', key: 'occupation', value: 'Clown' }
  ], function (err) {

    // read the whole store as a stream and print each entry to
stdout
    db.createReadStream()
      .on('data', console.log)
      .on('close', function () {
        db.close()
      })
  })
})
```

Rod Vagg

Contact
Sponsored Content

Execute this application and you'll end up with this output:

```
{ key: 'dob', value: '8 January 1983' }
{ key: 'name', value: 'Kim Jong-un' }
{ key: 'occupation', value: 'Clown' }
{ key: 'spouse', value: 'Ri Sol-ju' }
```

Basic operations

Open

There are two ways to create a new LevelDB store, or open an existing one:

Rod Vagg


```
levelup('/path/to/database', function (err, db) {
  /* use `db` */
})

// or

var db = levelup('/path/to/database')
/* use `db` */
```

The first version is a more standard Node-style async instantiation. You only start using the `db` when LevelDB is set up and ready.

The second version is a little more opaque. It looks like a synchronous operation but the actual ***open*** call is still asynchronous although you get a `LevelUP` object back immediately to use. Any calls you make on that object that need to operate on the underlying LevelDB store are ***queued*** until the store is ready to accept calls. The actual open operation is very quick so the initial is delay generally not noticeable.

Close

To close a LevelDB store, simply call `close()` and your callback will be called when the underlying store is completely closed:

```
' close to clean up
db.close(function (err) { /* ... */ })
```

Read, write and delete

Reading and writing are what you would expect for asynchronous Node methods:

```
db.put('key', 'value', function (err) { /* ... */ })

db.del('key', function (err) { /* ... */ })

db.get('key', function (err, value) { /* ... */ })
```

Batch

As mentioned in the first article, LevelDB has a ***batch*** operation that performs atomic writes. These writes can be either ***put*** or ***delete*** operations.

LevelUP takes an array to perform a batch, each element of the array is either a `'put'` or a `'del'`:

```
var operations = [
    { type: 'put', key: 'Franciscus', value: 'Jorge Mario
Bergoglio' }
  , { type: 'del', key: 'Benedictus XVI' }
]

db.batch(operations, function (err) { /* ... */ })
```

Rod Vagg

Contact
Sponsored Content

Streams!

LevelUP turns LevelDB's **Iterators** into Node's readable streams, making them surprisingly powerful as a query mechanism.

LevelUP's ReadStreams share all the same characteristics as standard Node readable object streams, such as being able to `pipe()` to other streams. They also emit all of the the expected events.

```
var rs = db.createReadStream()

// our new stream will emit a 'data' event for every entry in
the store

rs.on('data' , function (data) { /* data.key & data.value */ })
rs.on('error', function (err) { /* handle err */ })
rs.on('close', function () { /* stream finished & cleaned up */
})
```

But it's the various options for `createReadStream()` , combined with the fact that LevelDB sorts by keys that makes it a powerful abstraction:

Rod Vagg

```
.createReadStream({
    start     : 'somewheretostart'
  , end       : 'endkey'
  , limit     : 100           // maximum number of entries to
read
  , reverse   : true          // flip direction
  , keys      : true          // see db.createKeyStream()
  , values    : true          // see db.createValueStream()
})
```

Rod Vagg

Contact
Sponsored Content

`'start'` and `'end'` point to keys in the store. These don't need to even exist as actual keys because LevelDB will simply jump to the ***next existing key*** in lexicographical order. We'll see later why this is helpful when we explore ***namespacing*** and ***range queries***.

LevelUP also provides a **WriteStream** which maps `write()` operations to Puts or Batches.

Since ReadStream and WriteStream follow standard Node.js stream patterns, a ***copy database*** operation is simply a `pipe()` call:

```
function copy (srcdb, destdb, callback) {
  srcdb.createReadStream()
    .pipe(destdb.createWriteStream())
    .on('error', callback)
    .on('close', callback)
}
```

Encoding

P will accept most kinds of JavaScript objects, including Node's `Buffer` s, as both keys and values for all its operations. LevelDB stores everything as simple byte arrays so most objects need to be ***encoded*** and ***decoded*** as they go in and come out of the store.

Rod Vagg

You can specify the encoding of a LevelUP instance and you can also specify the encoding of individual operations. This means that you can easily store text and binary data in the same store.

`'utf8'` is the default encoding but you can change that to any of the standard Node `Buffer` encodings. You can also use the special `'json'` encoding:

```
var db = levelup('/tmp/dprk.db', { valueEncoding: 'json' })

db.put(
    'dprk'
  , {
        name       : 'Kim Jong-un'
      , spouse     : 'Ri Sol-ju'
      , dob        : '8 January 1983'
      , occupation : 'Clown'
    }
  , function (err) {
      db.get('dprk', function (err, value) {
        console.log('dprk:', value)
        db.close()
      })
    }
)
```

you the following output:

```
dprk: { name: 'Kim Jong-un',
  spouse: 'Ri Sol-ju',
  dob: '8 January 1983',
  occupation: 'Clown' }
```

Advanced example

In this example we assume the data store contains numeric data, where ranges of data are stored with ***prefixes*** on the keys. Our example function takes a LevelUP instance and a range key prefix and uses a ReadStream to calculate the variance of the values in that range using an online algorithm:

Rod Vagg

Contact
Sponsored Content

```
function variance (db, prefix, callback) {
  var n = 0, m2 = 0, mean = 0

  db.createReadStream({
        start : prefix          // jump to first key with the
prefix
      , end   : prefix + '\xFF' // stop at the last key with the
prefix
    })
    .on('data', function (data) {
      var delta = data.value - mean
      mean += delta / ++n
      m2 = m2 + delta * (data.value - mean)
    })
    .on('error', callback)
    .on('close', function () {
      callback(null, m2 / (n - 1))
    })
}
```

Rod Vagg

Contact
Sponsored Content

Let's say you were collecting temperature data and you stored your keys in the form: `location~timestamp`. Sampling approximately every 5 seconds, collecting temperatures in Celsius we may have data that looks like this:

```
au_nsw_southcoast~1367487282112 = 18.23
au_nsw_southcoast~1367487287114 = 18.22
au_nsw_southcoast~1367487292118 = 18.23
au_nsw_southcoast~1367487297120 = 18.23
au_nsw_southcoast~1367487302124 = 18.24
au_nsw_southcoast~1367487307127 = 18.24
...
```

ulate the variance we can use our function to do it while efficiently streaming values from our store by simply calling:

```
variance(db, 'au_nsw_southcoast~', function (err, v) {
  /* v = variance */
})
```

Namespacing

The concept of namespacing keys will probably be familiar if you're used to using a key/value store of some kind. By separating keys by prefixes we create discrete ***buckets***, much like a ***table*** in a traditional relational database is used to separate different kinds of data.

It may be tempting to create separate LevelDB stores for different buckets of data but you can take better advantage of LevelDB's caching mechanisms if you can keep the data organised in a single store.

Because LevelDB is sorted, choosing a namespace separator character can have an impact on the order of your entries. A commonly chosen namespace character often used in NoSQL databases is `':'`. However, this character lands in the middle of the list of ***printable ASCII characters*** (character code 58), so your entries may not end up being sorted in a useful order.

Imagine you're implementing a web server session store with LevelDB and you're prefixing keys with usernames. You may have entries that look like this:

Rod Vagg

Contact
Sponsored Content

```
rod.vagg:last_login     = 1367487479499
rod.vagg:default_theme = psychedelic
rod1977:last_login      = 1367434022300
rod1977:default_theme   = disco
rod:last_login          = 1367488445080
rod:default_theme       = funky
roderick:last_login     = 1367400900133
roderick:default_theme = whoa
```

Rod Vagg

Contact
Sponsored Content

Note that these entries are sorted and that `'.'` (character code 46) and `'1'` (character code 49) come before `':'`. This may or may not matter for your particular application, but there are better ways to approach namespacing.

Recommended delimiters

At the beginning of the printable ASCII character range is `'!'` (character code 33), and at the end we find `'~'` (character code 126). Using these characters as a delimiter we find the following sorting for our keys:

```
rod!...
rod.vagg!...
rod1977!...
roderick!...
```

or

```
rod.vagg~...
rod1977~...
roderick~...
rod~...
```

Rod Vagg

But why stick to the printable range? We can go right to the edges of the single-byte character range and use `'\x00'` (*null*) or `'\xff'` (*ÿ*).

For best sorting of your entries, choose `'\x00'` (or `'!'` if you really can't stomach it). But whatever delimiter you choose, you're still going to need to control the characters that can be used as keys. Allowing user-input to determine your keys and not stripping out your delimiter character could result in the NoSQL equivalent of an ***SQL Injection Attack*** (e.g. consider the unintended consequences that may arise with the dataset above with a delimiter of `'!'` and allowing a user to have that character in their username).

Range queries

LevelUP's ReadStream is the perfect range query mechanism. By combining `'start'` and `'end'`, which just need to be approximations of actual keys, you can pluck out the exact the entries you want.

Using our namespaced dataset above, with `'\x00'` as delimiters, we can fetch all entries for just a single user by carafting a ReadStream range query:

```
var entries = []

db.createReadStream({ start: 'rod\x00', end: 'rod\x00\xff' })
  .on('data', function (entry) { entries.push(entry) })
  .on('close', function () { console.log(entries) })
```

Would give us:

```
[ { key: 'rod\x00last_login', value: '1367488445080' },
  { key: 'rod\x00default_theme', value: 'funky' } ]
```

The `'\xff'` comes in handy here because we can use it to include every string of characters preceding it, so any of our user session keys will be included, as long as they don't start with `'\xff'`. So again, you need to control the allowable characters in your keys in order to avoid surprises.

Namespacing and range queries are heavily used by many of the libraries that extend LevelUP. In the final article in this series we'll be exploring some of the amazing ways that developers are extending LevelUP to provide additional features, applications and complete databases.

If you want to jump ahead, visit the Modules page on the LevelUP wiki.

Rod Vagg

Contact
Sponsored Content

DailyJS

Comments for this thread are now closed.

0 Comments DailyJS 1 Login

Recommend Share Sort by Best

This discussion has been closed.

Subscribe Add Disqus to your site Add Disqus Add Privacy

Proudly published with Ghost.

Rod Vagg

Contact

Sponsored Content

Subscribe

@dailyjs

Facebook

Google+

Feed

Email