

LevelDB Embedded Key Value Store Quick Cheat sheet

Aniruddha Chakrabarti

Associate Vice President and Chief Architect, Digital Practice, Mphasis

ani.c@outlook.com | [Linkedin.com/in/aniruddhac](https://www.linkedin.com/in/aniruddhac) | [slideshare.net/aniruddha.chakrabarti](https://www.slideshare.net/aniruddha.chakrabarti) | Twitter - anchakra

Agenda

- What is LevelDB
- Why LevelDB was created
- LevelDB libraries and bindings
- LevelDB operations using Node
 - Write data (put)
 - Read data (get)
 - Remove data (del)
 - Batch Operations (batch)
 - Iterating data (ReadStream)
 - Storing and Retrieving JSON data



What is LevelDB

LevelDB is a light-weight, single-purpose library for persistence with bindings to many platforms.

- LevelDB is a **key value store**, it stores data in disk. Highly optimized for fast read write (specially writes - [benchmark](#))
- Stores keys and values in arbitrary byte arrays, and data is **sorted by key**.
- LevelDB is typically used as **embedded database** (runs in the same process as your code), but can be networked (by adding protocols such as http, tcp to your process)
- LevelDB is not a key value database server like Redis or memcached. It's a **database library** similar to SQLite or Oracle Berkeley DB.
- Supported on Linux, Windows, OSX
- Has drivers and bindings for C++, Node, Python etc.



What is LevelDB

LevelDB is a light-weight, single-purpose library for persistence with bindings to many platforms.

- Written by Google fellows Jeffrey Dean and Sanjay Ghemawat (authors of MapReduce And BigTable)
- Inspired by BigTable - same general design as the BigTable tablet stack, but not sharing any of the code
- Developed in early 2011
- Chrome IndexDB feature (HTML5 API) is built on top of LevelDB
- Similar storage libraries include Oracle Berkeley DB, SQLite, Kyoto Cabinet, RocksDB etc.



Performance Benchmark

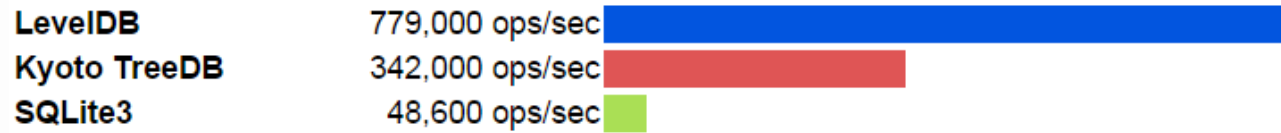
A. Sequential Reads



B. Random Reads



C. Sequential Writes



D. Random Writes



LevelDB outperforms both SQLite3 and TreeDB in sequential and random write operations and sequential read operations. Kyoto Cabinet has the fastest random read operations

<http://leveldb.googlecode.com/svn/trunk/doc/benchmark.html>



LevelDB library and bindings

- LevelDown - Low-level Node.js LevelDB binding
- LevelUp – High level Node.js-style LevelDB wrapper. **LevelUP** aims to expose the features of LevelDB in a **Node.js-friendly way**
- Rather than installing LevelDB and LevelUp separately you can install it as a whole – `npm install level`
- We would be using LevelUp Node.js LevelDB wrapper for all our examples
- **All operations in LevelUp are asynchronous** although they don't necessarily require a callback if you don't need to know when the operation was performed.



Installing LevelDB Node Bindings

Installation

- Install LevelUp Node Bindings (library) using npm (Node Package Manager) -
`npm install level` or
`npm install level -- save` (to update the dependency in `project.json`)
(LevelUp Node Bindings Location - <https://github.com/Level/levelup>)

Referencing the Node Library in code

- Reference the LevelUp Node library
`var level = require('level');`
- Create a db object by calling `level()` function passing the path where the data files would be stored. db object contains all the methods required for interacting with LevelDB. Here the data files would be stored in a subfolder called db relative to node file
`var db = level('./db');`



Write Data (put)

- Inserts/writes data into the store. Both the key and value can be arbitrary data objects.
- The Node API is asynchronous – put accepts key, value and a callback. Once the operation is completed (success or error) the callback is called.
- If error occurs then the error object is passed as argument to the callback. The callback argument is optional - if you don't provide the callback argument and an error occurs then the error is thrown, so it's better to provide the error argument.

```
var level = require('level');
var db = level('./db');
var key = "city", value = "Bangalore";

db.put(key, value, function(error){
    if (error != null){
        console.log(error);
    }
    else {
        console.log('data saved successfully to disk');
    }
});
```



Read Data (get)

- Reads/retrieves data from the store – the key for which data needs to be retrieved need to be specified.
- Similar to get, put also works asynchronously. It accepts key and a callback. Once the operation is completed (success or error) the callback is called, which accepts the value retrieved as parameter.
- If the key doesn't exist in the store then the callback will receive an error as its first argument. A not found error object will be of type 'NotFoundError'

```
var key = "city", another_key = "student";
db.get(key, function(error, city){
    console.log(city);           // displays Bangalore
});
db.get(another_key, function(error, value){
    if (error == null){
        console.log(value);
    }
    else{
        console.log(error.type + " - " + error.message);
    }
    // displays NotFoundError - Key not found in database [student]
});
```



Remove Data (del)

- Removes data from the store permanently.

```
var key = "city";
```

```
db.del(key, function(error){
    if (error != null){
        console.log(error);
    }
    else {
        console.log('data deleted successfully');
    }
});

db.get(key, function(error, city){
    console.log(city);           // displays undefined
});
```



Batch Operations

- Batch operations could be used for very fast bulk-write operations (both *put* and *delete*).
- Supports an array form and a chained form (similar to Fluent API)
- Array form
 - `db.batch(array, error callback)`
 - array argument should contain a list of operations to be executed sequentially, although as a whole they are performed as an atomic operation inside LevelDB.
 - Each operation is contained in an object having the following properties: type, key, value, where the type is either 'put' or 'del'
 - If key and value are defined but type is not, it will default to 'put'.
- Chained form
 - `batch()`, when called with no arguments returns a Batch object which can be used to build, and eventually commit, an atomic LevelDB batch operation.
 - Depending on how it's used, it is possible to obtain greater performance when using the chained form of `batch()` over the array form.



batch (array form)

```
var ops = [           // All operations of the batch are listed in an array
  {type:'put', key:'city1', value:'Bangalore'},           // type of operation, key & value
  {type:'put', key:'city2', value:'Kolkata'},
  {type:'put', key:'city3', value:'Chennai'},
  {type:'del', key:'city'},
]

db.batch(ops, function (error){
  if (error != null){
    console.log(error);
  }
  else {
    console.log('batch operation successful');
  }
})
```



batch (chained form)

- First line of code returns a Batch object which is then used to build all operations one by one
- Once it's built, it's eventually gets committed through write, which is an atomic LevelDB batch operation.

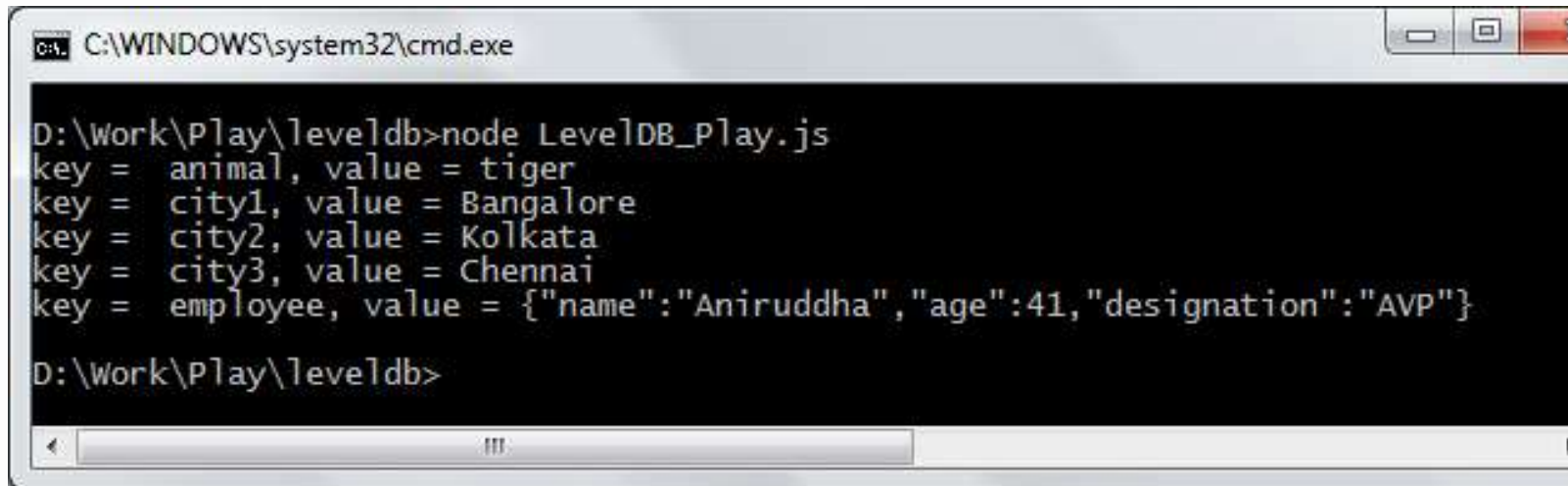
```
db.batch()                                // returns a Batch object which is used to chain all operations
    .del("city")
    .put("city1", "Bangalore")
    .put("city2", "Kolkata")
    .put("city3", "Chennai")
    .write(function() {                  // committing the batch operation
        console.log('batch operation successful');
    });
```



Iteration and ReadStream

- Get a ReadStream of the full database by calling the createReadStream() method – the returned stream is a complete Node.js-style Readable Stream. data object have key and value property.

```
var stream = db.createReadStream()
stream.on('data', function (data) {
    console.log('key=', data.key + ",value=" + data.value);
});
```



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The prompt is at "D:\Work\Play\leveldb>". The user has entered "node LevelDB_Play.js". The output shows five lines of data: "key = animal, value = tiger", "key = city1, value = Bangalore", "key = city2, value = Kolkata", "key = city3, value = Chennai", and "key = employee, value = {\"name\":\"Aniruddha\",\"age\":41,\"designation\":\"AVP\"}". The prompt is now "D:\Work\Play\leveldb>".

```
C:\WINDOWS\system32\cmd.exe
D:\Work\Play\leveldb>node LevelDB_Play.js
key = animal, value = tiger
key = city1, value = Bangalore
key = city2, value = Kolkata
key = city3, value = Chennai
key = employee, value = {"name":"Aniruddha","age":41,"designation":"AVP"}
D:\Work\Play\leveldb>
```

- Use the gt, lt and limit options to control the range of keys that are streamed.

```
var stream = db.createReadStream({limit:2})
```



KeyStream and ValueStream

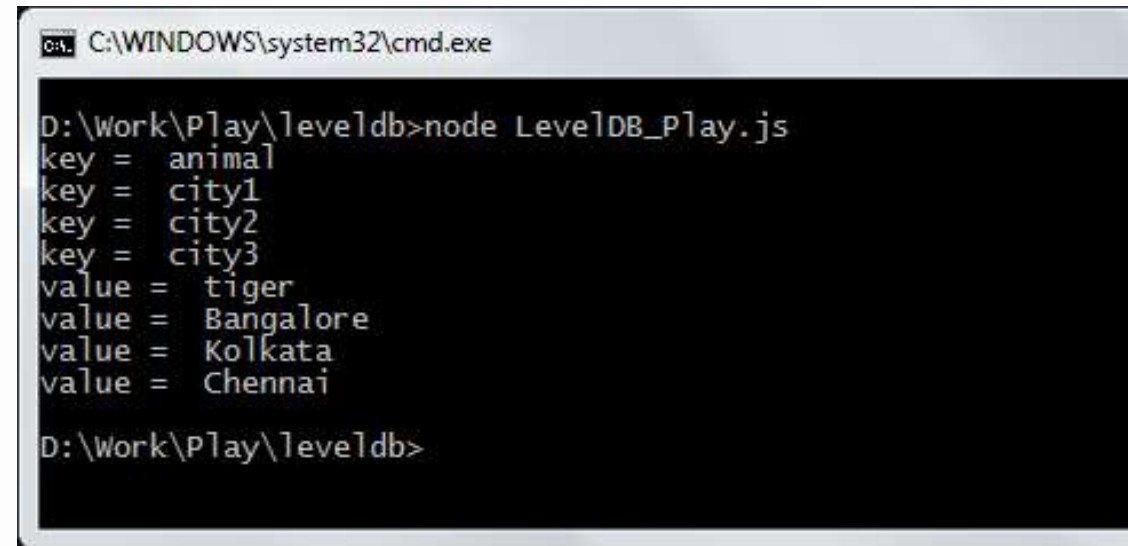
- A KeyStream is a ReadStream where the 'data' events are simply the keys from the database – use `createKeyStream()` method to get a KeyStream
- A ValueStream is a ReadStream where the 'data' events are simply the values from the database – use `createValueStream()` method to get a ValueStream

Iterating through all keys

```
var keyStream = db.createKeyStream()
keyStream.on('data', function (data) {
    console.log('key = ', data)
})
```

Iterating through all values

```
var valueStream = db.createValueStream()
valueStream.on('data', function (data) {
    console.log('value = ', data)
})
```



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The prompt shows the execution of a Node.js script: `D:\Work\Play\leveldb>node LevelDB_Play.js`. The output of the script is displayed as follows:

```
key = animal
key = city1
key = city2
key = city3
value = tiger
value = Bangalore
value = Kolkata
value = Chennai

D:\Work\Play\leveldb>
```



Storing JSON objects

- JSON objects could be stored as value, apart from simple strings, numbers etc.
- To store JSON objects, set the `valueEncoding` options parameter to `json` while constructing the db object.
- Multiple level of nested objects could be stored as value

```
var db = level('./dbJson', { valueEncoding : 'json' });
var employee = { name : "XYZ", age : 45, designation : "VP" };

db.put('employee', employee, function(error){
    db.get('employee', function(error, employee){
        console.log(employee.name);           // Aniruddha
        console.log(employee.age);           // 45
        console.log(employee.designation);    // VP
    });
});
```



Storing JSON objects (complex)

```
var db = level('./db.json', { valueEncoding : 'json' });
var employee = { name : "XYZ", age : 45, designation : "VP" };

var employee = {
  name : "XYZ",
  age : 45,
  designation : "VP",
  skills : ["NoSQL", "Management", "Languages", "Oracle"],           // Array
  address : { streetAddress: "123 M G Road", city: "Bangalore", country: "IN" } // Object
};

db.put('employee', employee, function(error){
  db.get('employee', function(error, employee){
    console.log(employee.name);           // XYZ
    console.log(employee.skills[0]);      // NoSQL
    console.log(employee.address.streetAddress + " - " + employee.address.city);
    // 123 M G Road - Bangalore
  });
});
```

