Modern Databases Project

Building your own modern database.

S.J.R. van Schaik, L. Torenvliet and R. de Vries

7th March 2015

Contents

1 Introduction

The past four weeks you have been introduced to various databases such as PostgreSQL, CouchDB and Neo4j. During the next few weeks, you'll be implementing your very own document-oriented database from the grounds up.

2 Setting up your environment

Assuming you are somewhat familiar with Python, but do or do not grasp the more advanced concepts yet, Python 3.3 or later would be a fairly good choice for this project. Therefore it is advisable to use Python 3.3 or later in conjunction with some useful packages, such as:

- asteval: a safe and minimalistic evaluator of Python expressions.
- bitstring: simple construction, analysis and modification of binary data.
- msgpack-python: a fast and compact binary serialisation format, suitable for data similar to JSON.
- python-snappy: a high-speed compression and decompression library from Google.
- sortedcontainers: sorted containers for Python.
- tornado: a web framework and asynchronous networking library.

Rather than installing these packages system-wide, it is better to use `virtualenv` to create an isolated environment instead. Make sure you have `python` (3.3 or later), `pip` and `virtualenv` installed, then issue the following commands:

Shell snippet

```
virtualenv --python=python3 env
env/bin/pip install numpy
env/bin/pip install asteval
env/bin/pip install bitstring
env/bin/pip install msgpack-python
env/bin/pip install python-snappy
env/bin/pip install sortedcontainers
env/bin/pip install tornado
```

After your environment has been set up, you can start out with the following file as your entry point to make sure your environment works.

main.py

```
1 #!env/bin/python
2
3 def main():
4     print('Hello, world!')
5
6 if __name__ == '__main__':
7     main()
```

Do note that `main.py:7` is essential to ensure that `main()` will not be executed when this file is being imported as a module within the Python interpreter or another file. This is useful for the purpose of testing your modules individually.

3 B+ trees

A document-oriented database has to be able to retrieve, store and manage documents. Rather than storing these in separate files on your file system, these will reside in a single file that will serve as your document store. This will allow us to operate on the file on a much lower abstraction level. To quickly locate a document within this file, we'll have to maintain an index. A data structure that is a proper candidate to serve as this index is the B-tree, which is inspired by the binary search tree. Within binary search trees each of the nodes have a key and two children. The key will determine what child will lead to the key you are looking for, if it is present within the tree. B-trees are essentially the same, but on a much larger level. Rather than having one key and two children per node, B-trees have n sorted keys and $n + 1$ children per node, where each two sequential keys will tell you the range of keys that a child will lead to. This is beneficial for hard disks where random access tends to have poor performance characteristics, because seeking is a slow operation, but where sequential access tends to have pretty good performance characteristics. As a result B-trees are usually very shallow, which reduces the amount of seek operations but have nodes that are very wide, which facilitates sequential access.

An even better candidate would be the B+ tree which is a variant to the B-tree. Unlike the B-tree, the B+ tree doesn't store data in its interiour nodes, but it has leaf nodes present for the intermediate keys as well. As a result the interiour nodes can fit many more keys in the same amount of space. In addition to this, B+ trees may also contain a pointer to the next leaf within the leaves in order to provide fast sorted iteration of the documents.

Now that we have discussed the data structure to use, we can look at the algorithms for the operations we have to implement in order to use the data structure.

3.1 Searching

Searching a key within a B+ tree can be done as follows:

- Start with the root node.
- For the current node, find the range in which that key fits, and select the corresponding node as the current node.
- Repeat the last step until the current node is a leaf node.

For searching, do consider the corner cases where the key is less than the smallest key, or greater than or equal to the largest key.

3.2 Insertion

Inserting a key into a B+ tree can be done as follows:

- Use the search operation as described earlier to find the corresponding leaf node.
- Insert the key into the bucket of the current node.
- If the bucket is too large:
  - Split the bucket in half.
  - Allocate a new leaf and move the upper half to the new bucket.
  - Insert the smallest key of the new leaf, and a pointer to the new leaf into the parent node.
  - Select the parent node as the current node, and repeat until the parent doesn't have to be split.
- In case the root node has to be split: create a new root node with one key and two pointers.

3.3 Deletion

Deleting a key from a B+ tree can be done as follows:

- Use the search operation as described earlier to find the corresponding leaf node.
- Remove the key from the leaf node.
- If the node is not the root node, and the bucket has less than $n/2$ entries:
  - Try to redistribute the entries, by moving one entry from either siblings to the bucket of the current node such that both nodes contain at least $n/2$ entries.
  - If redistribution is not possible, then merge the node with its sibling.
  - In case of a merge, select the parent node as the current node and delete the entry with the larger key until the tree is balanced.

Do note that merging may propagate to the root node, decreasing the height of the tree as a result.

3.4 Iteration and bulk-loading

In a B-tree iteration can be implemented by performing a depth-first search and only listing the leaves. For B+ trees it is sufficient to traverse to the first leaf and then iterate the linked list.

In case you have a lot of documents that you want to index some key of. One approach is to start with an empty tree and insert each of the documents. However, this is slow because the tree has to be traversed for each of the inserts. Instead by sorting the documents by key in ascending order, one can easily construct a B+ tree from the sorted documents with maximal utilisation of the nodes.

3.5 Implementation

The code below will aid you with the implementation of the B+ tree. In order to be able to use the `Tree` as a mutable map it should implement the `MutableMapping`-interface. Implement its functions once you are done with implementing the B+ tree itself.

btree.py

```python
from collections import Mapping, MutableMapping
from sortedcontainers import SortedDict

class Tree(MutableMapping):
    def __init__(self, max_size=1024):
        self.root = self._create_leaf(tree=self)
        self.max_size = max_size

    @staticmethod
    def _create_leaf(*args, **kwargs):
        return Leaf(*args, **kwargs)

    @staticmethod
    def _create_node(*args, **kwargs):
        return Node(*args, **kwargs)

    def _create_root(self, lhs, rhs):
        root = self._create_node(tree=self)
        root.rest = lhs
        root.bucket[min(rhs.bucket)] = rhs

        return root

    def __getitem__(self, key):
        pass

    def __setitem__(self, key, value):
        """
        Inserts the key and value into the root node. If the node has been
        split, creates a new root node with pointers to this node and the new
        node that resulted from splitting.
        """
        pass

    def __delitem__(self, key):
        pass

    def __iter__(self):
        pass

    def __len__(self):
        pass
```

Implement the selection of the node in the bucket, the insertion of a new key, and splitting a node using the algorithms described earlier in the code below:

btree.py

```python
class BaseNode(object):
    def __init__(self, tree):
        self.tree = tree
        self.bucket = SortedDict()

    def _split(self):
        """
        Creates a new node of the same type and splits the contents of the
        bucket into two parts of an equal size. The lower keys are being stored
        in the bucket of the current node. The higher keys are being stored in
        the bucket of the new node. Afterwards, the new node is being returned.
        """
        other = self.__class__()

        return

    def _insert(self, key, value):
        """
        Inserts the key and value into the bucket. If the bucket has become too
        large, the node will be split into two nodes.
        """
        pass

class Node(BaseNode):
    def __init__(self, *args, **kwargs):
        self.rest = None

        super(Node, self).__init__(*args, **kwargs)

    def _select(self, key):
        """
        Selects the bucket the key should belong to.
        """
        pass

    def _insert(self, key, value):
        """
        Recursively inserts the key and value by selecting the bucket the key
        should belong to, and inserting the key and value into that back. If the
        node has been split, it inserts the key of the newly created node into
        the bucket of this node.
        """
        pass

```

```python
class Leaf(Mapping, BaseNode):
    def __getitem__(self, key):
        pass

    def __iter__(self):
        pass

    def __len__(self):
        pass
```

3.6 Storing B+ trees on the hard disk

Once you have implemented the B+ tree in memory, it should be modified to operate from disk as well. First, we have to introduce `commit()` to reflect the changes by writing them to disk. Add `changed` as a boolean to the nodes, and modify `_insert()` to tag the nodes as changed. `commit()` should then be able to list the changes, and reset the changed nodes.

Furthermore, you should implement `encode()` and `decode()` to encode and decode nodes. Using `msgpack-python` you can efficiently store the nodes in a binary JSON format. To further increase the efficiency you can make use of the `snappy` library to compress and decompress the data:

encode.py using `snappy`.

```python
from msgpack import packb, unpackb
from snappy import compress, decompress

def encode(data):
    return compress(packb(data))

def decode(data):
    return unpackb(decompress(data))
```

Alternatively, you can make use of the `zlib` instead to compress and decompress the data:

encode.py using `zlib`.

```python
from msgpack import packb, unpackb
from zlib import compress, decompress

def encode(data):
    return compress(packb(data))

def decode(data):
    return unpackb(decompress(data))
```

Upon implementing these methods, the `encode()`-method can be used in `commit()` to write the changes to the disk. In order to read the node from the disk in a lazy manner, we'll be introducing a proxy wrapper that loads the node when necessary. The code below implements the `LazyNode`-class:

btree.py

```python
class LazyNode(object):
    _init = False

    def __init__(self, offset=None, node=None):
        """
        Sets up a proxy wrapper for a node at a certain disk offset.
        """
        self.offset = offset
        self.node = node
        self._init = True

    @property
    def changed(self):
        """
        Checks if the node has been changed.
        """
        if self.node is None:
            return False

        return self.node.changed

    def _commit(self):
        """
        Commit the changes if the node has been changed.
        """
        if not self.changed:
            return

        self.node._commit()
        self.changed = False

    def _load(self):
        """
        Load the node from disk.
        """
        pass

    def __getattr__(self, name):
        """
        Loads the node if it hasn't been loaded yet, and dispatches the request
        to the node.
        """
        if not self.node:
            self.node = self._load()

        return getattr(self.node, name)

    def __setattr__(self, name, value):
        """
        Dispatches the request to the node, if the proxy wrapper has been fully
        set up.
        """
        if not self._init or hasattr(self, name):
            return super().__setattr__(name, value)

        setattr(self.node, name, value)
```

Finish the implementation by completing `_load()`.

4 Document store

Now that we have implemented B+ trees for the purpose of indexing our documents, we can implement the actual document store. As has been described before our documents will reside in a single file on the filesystem, that will contain both the documents and the B+ tree representing the index.

4.1 Atomicity

Because we have to cope with hard disks not being very reliable, it makes sense to contemplate on measurements to guarantee that we can still consult our database, even if the previous commit did not succeed. The kind of property we are interested in is called atomicity: the principle of all or nothing. To prevent corruption we will start by introducing the simple rule that rather than modifying the data in-place, all the data has to be appended to the file. As an example, adding a new document to the database will result in the document being appended to the file, together with all the changes necessary to reflect the new state of the B+ tree. Upon committing all the changes to the file, the commit will be concluded with a file footer that contains a pointer to the B+ tree that maps document IDs to their respective documents.

4.2 Checksums

In addition to the rule of atomicity, we will also want to verify the integrity of the data. This can simply be done by introducing a checksum and the length to each chunk that is going to be written to the file. This can be implemented as follows:

checksum.py

```python
from binascii import crc32
from struct import Struct

pack_uint32 = Struct('>L').pack
unpack_uint32 = lambda x: Struct('>L').unpack(x)[0]

def add_integrity(data):
    data = pack_uint32(crc32(data)) + data
    data = pack_uint32(len(data)) + data

    return data

def check_integrity(data):
    if len(data) < 8:
        raise ValueError('expected at least 8 bytes, got {} bytes.'.format(
            len(data)))

    size, data = unpack_uint32(data[:4]), data[4:]

    if len(data) < size:
        raise ValueError('expected at least {} bytes, got {} bytes.'.format(
            size, len(data)))

    checksum, data = unpack_uint32(data[:4]), data[4:]

    if crc32(data) != checksum:
        raise ValueError('checksum does not match, data may possibly be '
            'corrupt.')

    return data
```

4.3 Revisions

Since we have introduced the rule that all changes have to be appended, rather than modified in-place, updating or deleting a document can only be achieved by appending a new version of the document to the file, and replacing the pointer to the document in the B+ tree. Each file footer will thus represent a different version or revision of the database. Because of this, it does not only makes sense to tag the document with the current version, but it also makes sense to manage a secondary B+ tree that maps these version numbers to their respective sets of documents.

4.4 File footers

From the above we can conclude that the file footer should contain the following fields:

- a checksum to verify the integrity of the file footer,
- the length of the data within the file footer,
- a pointer to the B+ tree containing a mapping from document IDs to their respective docu-

ments,

- a pointer to the B+ tree containing a mapping from database revisions to their respective sets of documents,
- and a number that represents the current revision, that gets incremented with each commit.

Now that we know what a file footer should look like, it would probably help to devise a method to identify the different file footers. Primarily, because we have to find the most recent one that isn't corrupt, in case a commit hasn't been written successfully.

4.5 Compaction

With the above implementation of the document store, you may find that it consumes a lot of space fairly quickly. Therefore, we should also implement compaction, which will simply build a compact version of the database from the most recent revision. This can be done by:

- collecting all the documents and revisions from the index tables,
- writing those to a new file that will serve as the new document store,
- constructing new B+ trees by bulk-loading the documents and revisions,
- and writing a file footer to conclude the commit.

Upon writing the new file, it can be moved to replace the old version of the document store.

5 REST-interface

At this point we should have a working document store. However, it is useless as long as it doesn't come with an interface that easily allows us to add new documents and look up, update and remove existing documents. Therefore, we will now focus on implementing a REST-interface. With **Re**presentational **S**tate **T**ransfer, we will be using the HTTP-protocol to perform the aforementioned operations on our database using verbs that are commonly used by web browsers to retrieve documents and post data.

Within REST, we will be operating on *collections* and *elements*. For example, the entire collection of documents will be accessible as `http://localhost:8080/documents/`, whereas a single document will be accessible as `http://localhost:8080/document/<id>/`.

For *collections* the verbs are commonly defined as follows:

- `GET` - Retrieves the entire collection of documents.
- `POST` - Creates a new document.
- `PUT` - Replaces the entire collection of documents with another.
- `DELETE` - Deletes the entire collection of documents.

For *elements* the verbs are commonly defined as follows:

- `GET` - Retrieves a document.
- `PUT` - Updates an existing document.
- `DELETE` - Deletes an existing document.

5.1 Using `curl` to test

To test our web interface, we'll be using a nifty utility called `curl`. Let's first look at some of the flags that we will be using:

- `--dump-header` or `-D` will show the HTTP headers alongside the response, as some operations may only yield a status code.
- `--request` or `-X` can be used to specify the kind of request we will be sending.
- `--header` or `-H` can be used to add specific headers to the request. As an example, we may be interested in telling what type of content we are sending, and will be expecting as a response.
- `--data-binary` can be used to specify the data to send. When prepended with an @-sign, a file can be specified instead.

6 Assignments

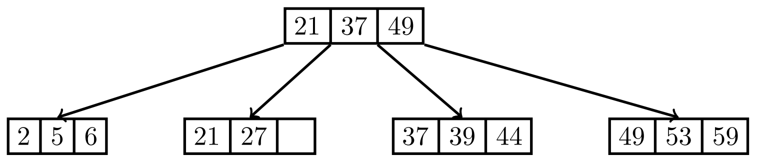
In order to evaluate both your comprehension and the progress you have managed to book thus far, you will be expected to write and hand in a report each week. The report should not only describe your implementation in full detail, but it should also contain a thorough description of the process you had to undertake. Furthermore, the report should also include your answers to the exercises you had to prepare for that week.

6.1 First week

These are the exercises for the first week:

- Explain what the differences are between a B-tree and a B+ tree in *your own words*.
- Describe the algorithms used to search, insert and delete a key in *your own words*.
- Explain what bulk-loading is and why it is better than inserting each key individually.
- Consider the B+ tree below. Illustrate what happens when the keys 32 and 42 are being inserted.

B+ tree



- Reconsider the aforementioned B+ tree. Illustrate what happens when the keys 49, 59 and 44 are being removed.

To conclude this week, you should have implemented:

- An index using a B+ tree with the possibility to search for existing documents and to insert new documents.
- The document store itself, which is the append-only file that consists of both the documents and an index.
- The possibility to bulk-load documents into the document store (**optional**).

- The possibility to compact your document store.

In order to shift the focus towards the project, you'll be provided with the TikZ code for the aforementioned B-tree:

btree.tex

```
\begin{tikzpicture}[node distance=1cm, scale=1, line width=0.1em,
        rectangle split, rectangle split horizontal, rectangle split parts=3]
\node[name=B1, draw] {
        $2$
        \nodepart{second} $5$
        \nodepart{third} $6$
};
\node[name=B2, draw, right=of B1] {
        $21$
        \nodepart{second} $27$
        \nodepart{third} {\ \ }
};
\node[name=B3, draw, right=of B2] {
        $37$
        \nodepart{second} $39$
        \nodepart{third} $44$
};
\node[name=B4, draw, right=of B3] {
        $49$
        \nodepart{second} $53$
        \nodepart{third} $59$
};
\path (B2) -- node (BC) {} (B3);
\node[name=A1, draw, above=of BC] {
        $21$
        \nodepart{second} $37$
        \nodepart{third} $49$
};
\path[draw, ->] (A1.south west) -- (B1.north);
\path[draw, ->] (A1.one split south) -- (B2.north);
\path[draw, ->] (A1.two split south) -- (B3.north);
\path[draw, ->] (A1.south east) -- (B4.north);
\end{tikzpicture}
```