Modern Databases Project

Building your own modern database.

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

23rd 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. This key is the minimum key of the node that has been created due to splitting and it should be pushed to the root node, i.e. it should be removed from its original node.

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, where you'll find a `BaseNode`, `Node` and `Leaf` class. Start by implementing the `Node._select` method. Given a key this method should return the respective child node this key should belong to. This method is then used by the `Node._insert` method, where the appropriate child node is being selected to call `_insert` on. As a result, the `Node._insert` method will be called recursively until a `Leaf` object has been found, in which case the `BaseNode._insert` method will be called instead.

The `BaseNode._insert` method should add the key and the value to the bucket of the node, and it should call `BaseNode._split` if the bucket is full. The `BaseNode._split` will then create a new node, and divide the bucket among the two nodes, whereupon it will return the new node to the caller. The `BaseNode._insert` method should return this new node as well. Then `Node._insert` should proceed by checking whether `None` or new node has been returned. In case of the latter, it should call `super()._insert()` (which calls `BaseNode._insert()`) to insert the new node. If the bucket of the node is too full, this node will be split as well, and this will keep occurring until either the root node has been reached, or a node of which the bucket won't be full.

If it is apparent that the bucket of the root node is full, then the root node should be split as well, whereupon a new root node should be created that contains a pointer to the old root node and the node that has been created due to splitting. Furthermore, the minimum key of the newly created node should then be moved, rather than copied, to the new root node. The old root node will thus become the `rest` of the new root node, and the moved key together with the newly created node will be part of the `bucket`. For the newly created node this means that the pointer that had been associated with the key should become the `rest` of the node, i.e. the `rest` serves as a pointer to the left-most child node, and the `bucket` serves as mapping from keys to the child nodes on the right of the keys.

Once the above has been implemented, it should be possible to insert new key and value pairs into the tree. However, it would also be interesting to be able to retrieve the value given a key.

In order to do that, you will have to implement Leaf.__getitem__ to return the value given a key. In addition to that you will also need a method in the Node class to recursively select nodes and perform a look up to finally reach a Leaf object and call Leaf.__getitem__. To complete the mapping you should also implement the Leaf.__iter__ and Leaf.__len__ methods.

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

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

    def __iter__(self):
        pass

    def __len__(self):
        pass
```

Once the implementation of the B+ tree has been implemented, the Tree class as shown in the code below should be implemented to finalise the B+ tree implementation. The Tree class inherits the MutableMapping-interface to make it act like a dictionary in Python. The methods of which an implementation is obligatory for this interfaces are: Tree.__getitem__, Tree.__setitem__, `Tree.__delitem__`, `Tree.__iter__` and `Tree.__len__`. Complete the implementation of these methods, except for the `Tree.__delitem__` method, as the deletion of nodes isn't required per se. However, if you are interested in implementing the deletion algorithm, then feel free to do so.

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
```

4 Document store

Even though you will now have a working B+ tree implementation, it is far from useful for a database as it operates purely in memory. In order to make it useful we have to reflect its state onto the hard disk as well. The B+ tree from the previous section will be used as the index of our document store so that we can efficiently look up documents. In this section we will be discussing the details of the document store, as well as its implementation.

Rather than writing all changes immediately to the disk, we will be working in terms of versions, where we will simply commit the changes since the last version. During each commit the newly added documents will be written and the B+ tree(s) will be updated. Because we have to cope with hard disks not being very reliable, it does make sense to think about measurements that guarantee that we can still consult our database, even if the previous commit has not been entirely stored to the disk. The kind of property we are interested in is called *atomicity*, which is the principle of all or nothing. One of these measurements that we will be using to prevent corruption is to never modify data in-place, as that would be unreliable. Instead, we will be appending all the changes to the end of the file. As an example, adding a new document to the document store will result in the document being appended to the file, followed by all the changes necessary to reflect the new state of the B+ tree. At the end of the commit a file footer will be written that contains information about the commit such as pointers to the root nodes of the B+ trees, the revision ID, etc.

In addition to the rule of atomicity, we should also be using checksums to verify the integrity of the data. A checksum is pretty much a hash of the contents of a chunk of data that will tell us whether the data is still intact or not. Furthermore, chunks should also be prefixed with fields that indicate the chunk type, and the length of the chunk.

4.1 Chunks

As mentioned before the chunks should be prefixed by a checksum, a length and possibly a chunk type. To easily add these, and to verify these fields you can use the following functions:

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
```

To efficiently store the chunk data, you can use the `msgpack` package that allows you to encode the chunk data to a compact binary JSON format. To further increase the efficiency you can make use of the `snappy` library to compress 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))
```

4.2 `LazyNode` class

In order to read the nodes from the disk in a lazy manner, i.e. to load them once we actually need them, we will 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)
```

You might not be familiar with the `__getattr__` and `__setattr__` methods that are used by this class. The `__getattr__` method will be called whenever an attribute is accessed that has not been set up by the class. If this method is being called, the class will try to dispatch the request to the node that it is wrapping. If the node hasn't been loaded yet, it will call the `_load` method. The `__setattr__` method is similar in behaviour, except that it will always be called, even when the attribute does exist. As both methods allow access of the attributes that are part of the node, the `LazyNode` tends to be very transparent class. However, it should be used with care, as a call to the `__iter__` method, for instance, will invoke the implementation of the `LazyNode` object, rather than that of the node that it is wrapping.

In order to start using the `LazyNode` class you should simply wrap all your nodes within your B+ tree implementation within instances of the `LazyNode` class.

4.3 Commits

Before we can commit the changes, we have to know what has been changed. A `commit` method should be added to the `Tree` class that will recursively call the `_commit` method of `LazyNode`. `LazyNode._commit` should check if it has been changed. If it has been changed, it should call the `_commit` method of its children. After the children have "committed" their data, it should commit its own data and set `changed` to `False`. For the purpose of testing, you might want to print the nodes that have been committed. Upon implementing this, you won't see anything yet. This is because you will also have to set the `changed` attribute to `True` in `Node._insert` to indicate that the node has been changed.

If everything is all right, it should print the leafs first, followed by the intermediate nodes, followed by the root node at the end. This is also the exact order that we will be writing the nodes in. Basically, once you have written a new state of a leaf to the disk, you will also have to update the references to that leaf by writing out their new state. As a result, you will have to follow the entire chain back to the root node. This is also why we won't have a typical B+ tree implementation where the leaves are also part of a linked list, as that would result in too many nodes being updated.

Now that we know what nodes have been changed we can extend the `_commit` method to actually write the node. First, you'll have to add a `_get_data` method to the `Node` and the `Leaf` classes. This method is supposed to get you the data to write to the chunk. For the `Node`, you'll want to store the fact that it is an intermediate node, its keys, and the file offsets of its children. For the `Leaf`, you'll want to store the fact that it is a leaf, its keys and its values (or offsets to documents). The `_commit` can then use the `_get_data` method to get the data and write it to the file as a chunk. Of course, before writing the data, you will have to retrieve the file offset to update the `LazyNode` with. To do this you will have to `seek` the end of the file, and then use `tell` to read the file offset, whereupon you can write the chunk.

Finally, once all the nodes have been written, you will have to conclude your commit with a file footer. The file footer should contain information about your commit, the most essential being the file offset to the root node.

4.4 Loading and unloading

Once you are able to commit the changes, you should also be able to load them. To do this, you'll have to locate the file footer and extract the file offset to the root node from it. This offset can then be used to create a `LazyNode` object that is to be stored as the root of a `Tree` object. When you are using the tree to retrieve documents, or insert document, it will attempt to use the `LazyNode`, which will result in a call to `LazyNode._load`. The `_load` method should use the file offset to load the chunk, and create a `Node` or `Leaf` object from the data that is stored inside. Using the offsets, you can create `LazyNode` objects for its children. This way the tree will be loaded lazily.

When you are iterating the documents, the tree will end up being fully loaded into memory. We can improve the memory usage by unloading the children once they have been accessed.

4.5 Compaction

With the above implementation of the document store, you may find that it consumes a lot of disk 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.

In order to implement a REST-interface you can make use of the tornado or the flask package. Thorough documentation for these packages can be found on the world-wide web. Both packages should be excellent for the implementation of a REST-interface, and possibly a web interface.

5.1 Map-reduce

The ability to perform flexible map-reduce queries is what makes the database useful. Map-reduce queries consists of three steps within the database:

1. Iterating over the documents using a map-query.
2. Merging the results in order to remove duplicate keys, and sorting the keys.
3. Iterating over the keys and their values using a reduce-query.

To allow the user to write the map and reduce queries, we will be using asteval, which is a package that implements an interpreter for a subset of the Python language. The following code wraps asteval in an easy to use class. A script can be loaded by creating an instance of this class, and calling the add_file method, a function that exists within the script can be invoked by calling the function as a method of the script object, and a function can be exposed by adding it to the symtable.

Wrapper object for asteval.

```python
from asteval import Interpreter

import functools
import re

class Script(object):
    def __init__(self):
        """
        Sets up an interpreter.
        """
        self.interpreter = Interpreter()
        self.symtable['re'] = re

    @property
    def symtable(self):
        """
        Expose the internal symbol table.
        """
        return self.interpreter.symtable

    @symtable.setter
    def symtable(self, symtable):
        """
        Apply changes to the internal symbol table.
        """
        self.interpreter.symtable = value

    def add_file(self, path):
        """
        Adds and loads code from a script file.
        """
        with open(path, 'rb') as f:
            self.interpreter(f.read())

    def invoke(self, name, *args, **kwargs):
        """
        Invokes a function in the script with the appropriate arguments.
        """
        f = self.interpreter.symtable.get(name, None)

        if not callable(f):
            return

        return f(*args, **kwargs)

    def __getattr__(self, name):
        """
        Returns the function to invoke a function in the script, if a function
        with that name exists within the symbol table. Otherwise, an attribute
        error is being raised (default behaviour).
        """
        if name in ['symtable', 'interpreter']:
            raise AttributeError("{} instance has no attribute '{}'".format(
                self.__class__.__name__, name))

        if not callable(self.symtable.get(name, None)):
            raise AttributeError("{} instance has no attribute '{}'".format(
                self.__class__.__name__, name))

        return functools.partial(self.invoke, name)
```

With the above script you should be able to implement the `map` and `reduce` stages. For the `map` queries, you should use a temporary document store to store the results in by exposing an `emit` function that accepts a key and a value. The benefit of using a document store is that you can append values to existing keys by updating the documents, and that your keys will be sorted, if you have the appropriate B+ tree set up for this. Furthermore, this will allow you to perform map-reduce queries on huge amounts of data. By iterating this document store, you can also easily apply the `reduce` script.

Finally, offer the ability to execute map-reduce queries through your REST-interface.

6 Assignments

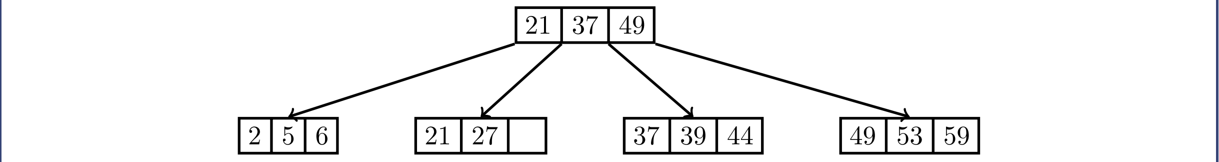
In order to evaluate both your comprehension and the progress you have managed to book thus far, you will be expected to write and hand in a report each week. The report should not only describe your implementation in full detail, but it should also contain a thorough description of the process you had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

6.2 Second week

For this week you should implement a REST-interface with support for:

- retrieval, creation, updates and deletion of documents.
- retrieval of sets of documents.
- insertion of sets of documents via bulk loading (**optional**).
- execution of map-reduce queries.
- compaction of the database.

Optionally, you can also provide a user-friendly web interface for all of the above operations.

Once you have finished the REST-interface together with support for map-reduce queries, it is time to put your database to a test. Perform the following exercises *individually*:

- Install the database.
- Import the **N**ational **V**ulnerability **D**atabase (`http://nvd.nist.gov/download.cfm`) using a SAX parser. Even though the NVD is related to the **C**ommon **V**ulnerability and **E**xploits database (`https://cve.mitre.org/data/downloads`), you shouldn't need it.
- Write a map-reduce query to figure out the top ten of software products of the previous year by the total number of vulnerabilities and exposures. Plot these results in a graph.
- Write a map-reduce query to figure out the top ten of software vendors of the previous year by the total number of known vulnerabilities and exposures affecting their software product(s). Plot these results in a graph.
- Compare your results with the results available at `http://www.cvedetails.com/top-50-products.php?year=2015`. Do notice that they are using various sources for the exploits (e.g. `http://www.exploit-db.com` and Metasplit).
- Profile your database to see how well your database performs. Report your findings and explain them. Think of possible improvements and discuss them.

6.3 Third week

Read the paper on the Chord overlay network [1] and do the following exercises *individually*:

- Describe the structure of a Chord overlay network. What kind of information does each node contain and in what way are they linked with each other?
- The Chord overlay network is considered scalable. Explain why this is true.
- Explain what happens when a node joins an existing Chord overlay network.
- Describe the purpose of each of the following methods used in the aforementioned template code:
  - `stabilise`
  - `notify`
  - `fix_fingers`
- Consider a stable Chord network with a ring size of 32 that contains the following set of node IDs 4, 7, 17 18, 19. Show the predecessor and the fingers for each of the nodes.
- Consider a stable Chord network with a ring size of 32 that contains the following set of node IDs 6, 7, 22. Show the predecessor and the fingers for each of the nodes.

- Illustrate what happens when the node with node ID 6 leaves the network due to a failure by showing the predecessor and the fingers for each of the nodes.

Once you have finished the exercises, implement the Chord algorithm locally using the template code below:

chord.py

```python
import math

class Finger(object):
    def __init__(self, start, node):
        self.start = start
        self.node = node

class Node(object):
    ring_size = 2 ** 5
    finger_count = int(math.log(ring_size, 2))

    @property
    def successor(self):
        return self.fingers[0].node

    @successor.setter
    def successor(self, value):
        self.fingers[0].node = value

    def __init__(self, node_id):
        self.alive = True
        self.node_id = node_id
        self.predecessor = self
        self.fingers = [Finger((node_id + 2 ** k) % self.ring_size, self) for k
            in range(self.finger_count)]

    def distance(self, lhs, rhs):
        return (self.ring_size + rhs - lhs) % self.ring_size

    def in_range(self, value, lower, upper):
        if lower is upper:
            return True

        return self.distance(lower, value) < self.distance(lower, upper)

    def print_fingers(self):
        print('Finger table for node #{}:'.format(self.node_id))
        print('\n'.join('{}: {}'.format(finger.start, finger.node.node_id) for
            finger in self.fingers))

    def join(self, node):
        pass

    def stabilise(self):
        pass

    def notify(self, node):
        pass

    def fix_fingers(self):
        pass

    def find_successor(self, node_id):
        pass

    def closest_preceding_node(self, node_id):
        pass

    def is_alive(self):
        return self.alive

    def kill(self):
        self.alive = False
```

In order to handle node failures, you should extend your implementation by maintaining a successor list that is to be populated by the response from a node to its predecessor. Furthermore, you should implement the `is_alive` method to be able to simulate node failures.

Finally, you can use the code below to simulate joins and failures, essentially allowing you to test your Chord implementation thoroughly:

chord-simulation.py

```python
from random import randint, sample
from node import Node

node_ids = sample(range(Node.ring_size), 5)
node_failures = 2
nodes = []

for node_id in node_ids:
    print('Creating node with node ID: #{}.'.format(node_id))
    node = Node(node_id)
    nodes.append(node)

    if len(nodes) <= 1:
        continue

    known_node = nodes[0]

    print('Attempting to join network via node ID: #{}.'.format(
        known_node.node_id))

    node.join(known_node)

    print('Stabilising...')

    for _ in range(Node.finger_count):
        for node in nodes:
            node.stabilise()

        for node in nodes:
            node.fix_fingers()

    print('Network:')

    for node in nodes:
        node.print_fingers()

for _ in range(node_failures):
    node = nodes[randint(0, len(nodes) - 1)]

    print('Killing node #{}...'.format(node.node_id))
    node.kill()

    print('Stabilising...')

    for _ in range(Node.finger_count):
        for node in nodes:
            node.stabilise()

        for node in nodes:
            node.fix_fingers()

    print('Network:')

    for node in nodes:
        node.print_fingers()
```

7 References

[1] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.