

B+tree en DocumentStore implementatie

Steven Raaijmakers, Bär Halberkamp

March 2017

1 Inleiding

De afgelopen weken hebben wij ons beziggehouden met het verwerken van verschillende soorten datasets in verschillende databases. Vanaf deze week beginnen we met het implementeren van onze zelf-geschreven database, volgens de “tutorial” uit de syllabus. De eerste week zal bestaan uit het implementeren van de *b+tree* datastructuur, en de *DocumentStore*.

2 Probleemstelling

We beginnen in de eerste week met het maken van onze eigen datastructuur waarin we de bestanden uit de database logisch in kunnen gaan onthouden. Een geschikte datastructuur hiervoor is de zogeheten *b+tree*. De *b+tree* zal zelf geschreven worden in python3.

De gehele database zal opgeslagen worden in één bestand; de Document Store. De data wordt hier encoded in opgeslagen, en middels de *b+tree* worden de posities van verschillende documenten onthouden zodat deze gemakkelijk kunnen worden opgevraagd.

Om documenten lazy in te laden wordt gebruik gemaakt van de *LazyNode* class. Deze creëert een wrapper rond de Node en Leaf classes zodat data pas ingeladen hoeft te worden als deze opgevraagd wordt.

3 Theorie

3.1 Document Store

De document store is het bestand waar de hele database in opgeslagen wordt. Hij bevat de data in een compact formaat, om het inladen en herbouwen van de B+ boom na het herstarten van de database mogelijk te maken, en om een mogelijkheid te bieden om de boom lazy te genereren, zodat het mogelijk is de data beschikbaar te hebben zonder alle documenten in te laden in het geheugen. Elke stuk data in de Document Store bestaat uit de volgende elementen:

- **length:** dit veld van 8 bytes duidt aan hoeveel bytes de data uit bestaat. Dit veld wordt als eerste gelezen, waarna de data wordt ingeperkt tot de length gegeven in dit veld.
- **checksum:** dit veld van 8 bytes bevat de crc32 checksum van de data, die wordt vergeleken met de checksum van de gelezen data, om data corruption te voorkomen.
- **data:** De data, encoded met snappy. Deze data bevat een lijst met child nodes of waardes in de bucket, met een revision number om bij te houden hoe recent de entry is.

Er is geen separator tussen verschillende data entries, dus het is belangrijk om de juiste locaties in de store te onthouden. Om te zorgen dat er geen data wordt overschreven en de juiste offsets intact blijven wordt er alleen aan het einde van de document store data toegevoegd. Bij het appenden wordt de waarde van de (nieuwe) offset gereturnd, zodat de juiste offset gemakkelijk aan de node doorgegeven kan worden.

Voor het efficiënt inladen van data wordt gebruik gemaakt van de LazyNode class. Deze werkt als een wrapper voor nodes, en bevat de informatie over de offset. Bij het opslaan of inladen van data wordt hiervan gebruik gemaakt om de data te laden uit de store, of aan de parent nodes door te geven waar deze data te vinden is in de store, zodat deze bij het laden van de boom weer doorgegeven kan worden aan de nieuwe LazyNode instances.

Als laatste bij een commit wordt door de Document Store een footer doorgegeven. Deze footer bevat het nummer van de huidige revision en de offset in de store waar (de meest recente revision van) de root node te vinden is, zodat het mogelijk is de tree opnieuw op te bouwen.

Compaction wordt gedaan door een nieuwe tijdelijke store te maken, en vervolgens de hele tree te committen. Hiervoor wordt gebruik gemaakt van een **force** optie in de `commit()` functie, zodat er geen rekening wordt gehouden met of nodes wel of niet aangepast zijn sinds de laatste revision.

3.2 B+tree

De B+tree is een boom, waarbij al de data wordt opgeslagen bij de leaf-nodes (nodes zonder kinderen). Bij een b+tree hangt elke leaf op hetzelfde niveau. Dit houdt in dat dus elke leaf-node evenveel voorouders heeft als elke andere leaf-node. Verder heeft elke leaf node ook links naar andere leaf-nodes. Zo kan er op een horizontaal niveau gezocht worden.

3.3 Antwoorden op de vragen

- Explain what the differences are between a B-tree and a B+ tree in your own words.

B+trees slaan de data op bij hun leaf nodes, terwijl dit bij B-tree op de tussenliggende nodes wordt gedaan. Daarnaast hebben de leaves in

een b+tree informatie over elkaar. Hierdoor kan er gemakkelijk verticaal gezocht worden. Bij een b-tree zal er voor verschillende leaves telkens een depth-first search moeten worden toegepast

- **Describe the algorithms used to search, insert and delete a key in your own words**

– Search:

1. Begin bij de root
2. Loop over alle keys van de current-node totdat de geschikte node gevonden wordt (dat is wanneer de huidige key kleiner is dan de key waarnaar wordt gezocht)
3. Voor deze bucket wordt gekeken of het een leaf betreft. Zo ja; zoek de key in deze bucket. Zo nee; wordt er in de gehele bucket gekeken wat de geschikte plek is en zal het vanaf stap 2 herhaalt worden

– Insert

1. Door gebruik van het search algoritme kan de geschikte bucket voor de key gevonden worden.
2. Voeg de key aan de bucket toe. Wanneer de nieuwe lengte van de bucket de *tree.max_size* overschrijdt wordt de bucket in twee gelijke delen verdeeld.

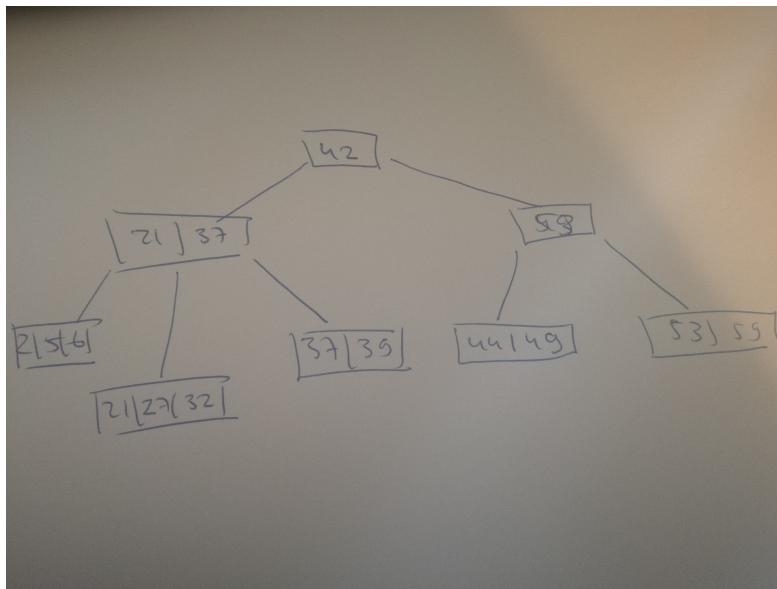
– Deleting

1. Door gebruik van het search algoritme kan de bijbehorend bucket gevonden worden.
2. Vervolgens kan de key uit de bucket gehaald worden.
3. Als de nieuwe bucket length kleiner is dan de helft van de max_size wordt gekeken of de bucket neighbors heeft
4. Met de neighbors zal er worden gemerged (indien mogelijk met links, anders met rechts).
5. Als een merge niet mogelijk is zal geprobeerd worden een item uit de buckets van de neighbor te halen waardoor de lengte groter blijft dan de helft van de max_size. De parent nodes blijven verwijzen naar de kleinste value in hun buckets.

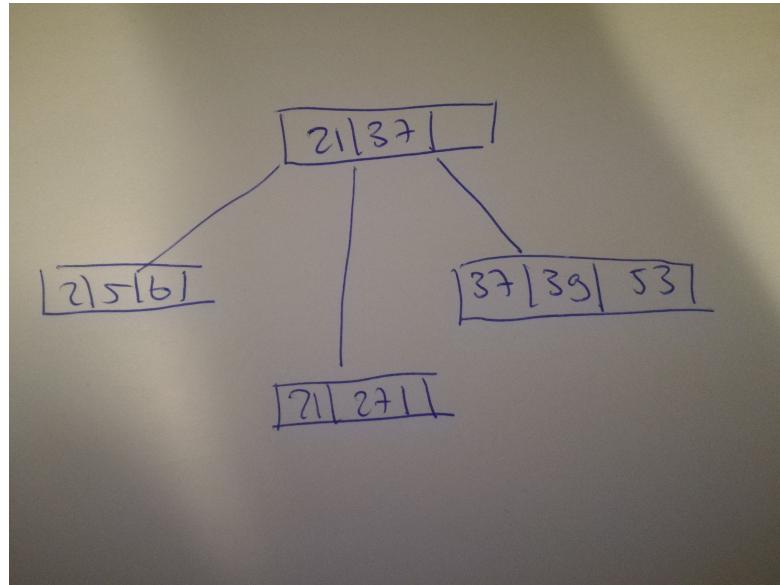
- **Explain what bulk-loading is and why it is better than inserting each key individually** Bij het inladen van vele documenten tegelijkertijd in de b+tree zullen de documenten één voor één ingeladen worden in de tree. In plaats daarvan kan er bij bulk-loading voor gezorgd worden dat de in te laden documenten al op een bepaalde manier gesorteerd zijn, waardoor er tijd bespaard wordt.

- **Consider the B+ tree below. Illustrate what happens when the keys 32 and 42 are being inserted.** Het illustreren van de nieuwe boom is helaas niet gelukt via de tikz manier. Bij het toevoegen van 32, zal 32

in het lege vakje naast 27 komen te staan. Wanneer we echter 42 willen toevoegen, zal de bucket size (van 3) worden overschreden. In dat geval zal de bucket van 37, 39, 44 worden opgedeeld in twee kleiner buckets van 37, 39 en 42, 44. Omdat er nu twee buckets zijn in plaats van een, moet dat op de een of andere manier duidelijk gemaakt worden aan de parents. 42 zal omhoog verplaatsen (met pointers naar de nieuwe buckets), maar in de bucket van de parents is ook geen ruimte. Hierdoor zal de bucket van 21, 37, 49 opgesplitst worden. Hierbij zou eigenlijk de value 42 plaatsnemen maar aangezien dit de middelste value wordt (in deze te grote bucket) zal 42 weer een plek omhoog schuiven met verwijzingen naar 21, 37 en 49. Daarna moeten de takken nog logisch gemaakt worden:



- Reconsider the aforementioned B+ tree. Illustrate what happens when the keys 49, 59 and 44 are being removed. Als 49 wordt verwijderd, wordt de bucket van (44, 53, 59) verandert naar (53, 59). Omdat 53 nu het laagste item is in de bucket, zal de parent ipv naar 44 nu naar 53 verwijzen. Als vervolgt 59 verwijderd wordt heeft de bucket minder dan $\max_size/2$ items, dus zal hij gaan zoeken aan zijn linker kant naar een item. Hij vindt daar 44, en dus zal de bucket nu (44, 53) worden. Omdat 44 nu het laagste item is, zal de parent naar 44 verwijzen ipv naar 53. Tot slot wordt 44 verwijderd waardoor de bucket (53) zal gaan mergeren met zijn linker neighbor, waardoor de bucket (37, 39, 53) ontstaat:



4 Onderzoek

Hoewel de syllabus aangeeft dat het verstandig is om python in een virtual omgeving te starten, is er voor gekozen alle libraries te downloaden via een normale omgeving. Er traden namelijk vaak errors op met snappy (terwijl zlib het wel deed). Daarom zijn volgens het volgende commando de libraries gedownload voor python3:

```
python -m pip install LIBRARY
```

Omdat we de tutorial uit de syllabus hebben gevolgd was het lastig te begrijpen wat precies de functie was van de verschillende objecten in de voorbeeld code. Tree is blijkbaar een dictionary, en de functie `__getitem__` betekent dus hetzelfde als het opvragen vanuit een dictionary volgens `tree[item]`.

5 Experimenten

5.1 B+Tree

In sectie 7.3 van de syllabus staat de implementatie van de b+tree beschreven. Deze volgen we stappen voor stap.

Bij het initialiseren van de tree wordt een `max_size` meegegeven. Deze slaat op de maximale grote die elke bucket mag hebben.

Te beginnen met de `Node._select`. Deze functie moet met een gegeven key de geschikte plek vinden in de boom. Dat gaat als volgt:

```

for i, (k, v) in enumerate(self.bucket.items()):
    if i == 0 and key < k:
        return self.rest
    elif key < k:
        return self.bucket[self.bucket.iloc[i - 1]]
    else:
        return self.bucket[self.bucket.iloc[i]]
```

De Node implementeert de `BaseNode`, waarin zich de bucket bevindt. De bucket heeft de vorm van een `sortedDictionary` (<http://www.grantjenks.com/docs/sortedcontainers/sorteddict.html>), en zal dus altijd het laagste element als eerst hebben en het hoogste element pas op het einde.

Er wordt over alle elementen uit de bucket heen gelopen. Hierbij wordt telkens gekeken of het item uit de bucket een grotere waarde heeft dan te selecteren `key`. Is dit het geval zal het elementen aan de linkerzijde van het huidige elementen gereturned worden als geschikte parent.

De edge cases zijn als de te selecteren `key` al kleiner is dan het eerste item uit de bucket, in dat geval zal de rest worden gereturned. Als de `key` groter is dan de `key` van het laatste item uit de bucket zal het laatste item zelf gereturned worden.

De `Node._select` functie wordt gebruikt in `Node._insert` om dus de juiste plek te vinden.

```

n = self._select(key)._insert(key, value)

if n:
    self.bucket[min(n.bucket)] = n

if len(self.bucket) > self.tree.max_size:
    return self._split()

return None
```

In deze functie wordt uit de node dus de geschikte plek geselecteerd en vervolgens wordt op deze plek de te inserten (`key, value`) geplaatst. Als dit resulteert in een splitsing van de bucket zal de het kleinste item uit de nieuwe bucket in de huidige bucket verwijzen naar deze nieuwe node.

De `BaseNode._insert` voegt de `key` en de `value` aan de bucket toe van de node, en roept waar nodig de `split` functie aan wanneer door de toevoeging de max size van de bucket wordt overschreden:

```

self.bucket[key] = value

if len(self.bucket) > self.tree.max_size:
    return self._split()
```

```
return None
```

In de *BaseNode._split* wordt de bucket in twee delen opgesplitst. Dit wordt gedaan door de oude bucket te vervangen door helft van de oude bucket, en de overige helft wordt in een nieuwe bucket gestopt. Deze nieuwe bucket (other) wordt vervolgens gereturned:

```
other = self.bucket[len(self.bucket) // 2:]
self.bucket = self.bucket[:len(self.bucket) // 2]
```

6 Resultaat

De b+tree kan getest worden, door de volgende code te gebruiken

```
t = Tree(storage=store, max_size=5)
for i in range(0, 10):
    t[i] = i ** 2
```

Hiermee wordt aangegeven dat er 10 elementen in de boom moeten worden gestopt, die een *bucket_size* van 5 heeft. Elke element krijgt de waarde van zijn eigen index tot de macht twee.

Als nu vervolgens de tree geprint wordt, krijgen we de volgende output:

```
(3, Leaf: {3: 9, 4: 16, 5: 25})
(6, Leaf: {8: 64, 9: 81, 6: 36, 7: 49})
```

Hierbij wordt de leaf van de linkerzijde niet correct geprint. Hier bevindt zich namelijk nog een leaf:

```
Leaf: {0: 0, 1: 1, 2: 4}
```

De informatie uit deze leaf kan wel opgevraagd worden (via de dictionary manier), waarmee we de correctheid van de boom kunnen verifiëren

7 Conclusie en Discussie

Bij het implementeren van de b+tree was een duidelijke handleiding gegeven. Na het uitvoeren van alle stappen in de handleiding wilden we graag testen of de b+tree het deed. Dit ging echter niet gemakkelijk omdat de b+tree lastig te visualiseren is. Echter door het maken van een for-loop die elementen aan de b+tree toevoegde konden we bepalen of de implementatie geslaagd was. Door bijvoorbeeld het inladen van 1000 nodes in een boom met een *max_size* van 10 konden we bepalen of de nodes in de root klopte. Vervolgens konden we uit diezelfde boom weer waardes opvragen die ook correct bleken.

Helaas lukte het niet helemaal om de LazyNode werkend te krijgen. Hoewel onze tests voor de Document Store de juiste uitkomsten hadden, werkte hij in de praktijk niet correct. Het is ons niet duidelijk geworden waar het probleem

precies ligt, maar bij het oproepen van de eerder gecommitte data kwam er een Key error in de Leaf.

8 Appendix

btree.py

```
from __future__ import division
from collections import Mapping, MutableMapping
from sortedcontainers import SortedDict
from storage import DocumentStore

store = DocumentStore()

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 _commit(self):
        offset = self.root._commit()

    def __getitem__(self, key):
        s = self.root._select(key)
        while(not isinstance(s, Leaf)):
            s = s._select(key)
        return s[key]

    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.
"""
# inserts new item
n = self.root._insert(key, value)
# check if split happened
if n:
    self.root = self._create_root(self.root, n)

# hoeft niet deze
def __delitem__(self, key):
    pass

def __iter__(self):
    yield from self.root

def __len__(self):
    return len(self.root)

def __repr__(self):
    return str(self.root)

# Basenode
class BaseNode(object):
    def __init__(self, tree, changed=False):
        self.tree = tree
        self.bucket = SortedDict()
        self.changed = changed

    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__(self.tree)

        # fill "other" with first part of original bucket
        # + remove first part from original bucket
        for k, v in self.bucket.items()[len(self.bucket) // 2:]:
            other.bucket[k] = v
            del self.bucket[k]

        return other

    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.
"""
self.bucket[key] = value
self.changed = True

# determine whether bucket should be split
if len(self.bucket) > self.tree.max_size:
    return self._split()

# using None to indicate no split has happened
return None

# Node
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.
        """
        # determine where key should be put
        for i, (k, v) in enumerate(self.bucket.items()):
            # if first item has a bigger key then the key-2b-placed
            if i == 0 and key < k:
                return self.rest
            # if current key is bigger than k2p, return PREV
            elif key < k:
                return self.bucket[self.bucket.iloc[i - 1]]

        # nothing found; return last
        return self.bucket[self.bucket.iloc[i]]

    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.
        """
        n = self._select(key)._insert(key, value)

        # upon split: current buckets lowest val should point to split

```

```

        if n:
            self.bucket[min(n.bucket)] = n

        # split if necessary
        if len(self.bucket) > self.tree.max_size:
            return self._split()

    return None

# printing for debugging
def __repr__(self):
    t = []
    for i in self.bucket:
        t.append(i)
    return "Node: " + str(t)

# wht
class Leaf(Mapping, BaseNode):
    def __getitem__(self, key):
        return self.bucket[key]

    def __iter__(self):
        for key in self.bucket.keys():
            yield key, self[key]

    def __len__(self):
        return len(self.bucket)

    def __repr__(self):
        return "Leaf: " + str(dict(self.bucket))

class LazyNode(object):
    _init = False

    def __init__(self, tree=None, offset=None, node=None):
        self.offset = offset
        self.node = node
        self.tree = tree
        self._init = True

    @property
    def changed(self):
        if self.node is None:
            return False

    return self.node.changed

```

```

def _commit(self):
    if not self.changed:
        return

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

def _load(self):
    if not self.offset:
        self.offset = 0

    data = decode(store.get(self.offset))
    self.node = Node(tree=self.tree, id=self.offset)
    if 'children' in data:
        for child in data.children:
            self.node._insert(child, LazyNode(tree=self.tree, offset=child, node=None))

def __getattr__(self, name):
    if not self.node:
        self.node = self._load()

    return getattr(self.node, name)

def __setattr__(self, name, value):
    if not self._init or hasattr(self, name):
        return super().__setattr__(name, value)

    setattr(self.node, name, value)

```

storage.py

```

import os
from checksum import add_integrity, check_integrity, pack_footer, unpack_footer

class DocumentStore:
    def __init__(self, filename='./storage.txt'):
        self.file = filename
        self.oldfile = None
        self.compacting = False

    def get(self, offset):
        filesize = os.stat(self.file).st_size

        if filesize < offset:
            raise ValueError('Trying to retreieve offset {}, but file is only {} bytes long.')

```

```

        with open(self.file, 'rb') as f:
            f.seek(offset)
            return check_integrity(f.read())

    def append(self, data):
        data = add_integrity(data)
        with open(self.file, "ab") as f:
            f.seek(0, 2)
            end = f.tell()
            f.write(data)
        return end

    def write_footer(self, rev, offset):
        with open(self.file, "ab") as f:
            f.write(pack_footer(rev, offset))

    def read_footer(self):
        with open(self.file, "rb") as f:
            f.seek(-8, 2)
            return unpack_footer(f.read())

    def start_compacting(self):
        if self.compacting:
            return

        self.oldfile = self.file
        self.file = self.file + "_new"
        self.compacting = True

    def stop_compacting(self):
        if not self.compacting:
            return

        os.remove(self.oldfile)
        os.rename(self.file, self.oldfile)

        self.file = self.oldfile
        self.oldfile = None
        self.compacting = False

```

checksum.py

```

from binascii import crc32
from struct import Struct

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

```

```

def pack_footer(rev, offset):
    return pack_uint32(rev) + pack_uint32(offset)

def unpack_footer(footer):
    if len(footer) != 8:
        raise ValueError('expected 8 bytes, got {}'.format(len(footer)))

    return unpack_uint32(footer[:4]), unpack_uint32(footer[4:])

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 {}'.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:size]

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

    return data

```