# Chord overlay network

Bär Halberkamp
Steven Raaijmakers

March 2017

## 1 Inleiding

Na 2 weken hebben we onze eigen database gebouwd volgens de "tutorial" in de syllabus. We hebben onze database gebouwd op een b+tree en een documentstore (week 1). In de tweede week van het project hebben we de mogelijkheid toegevoegd om map-reduce queries los te laten op onze database (week 2). In de laatste week gaan we onze echter bezighouden met iets anders, het chord algoritme.

## 2 Probleemstelling

Wanneer gewerkt wordt met een peer-2-peer systeem, is het belangrijk om te weten waar zich bepaalde data bevindt. Dit kan onthouden worden door middel van het Chord protocol, en wordt een distributed hash tble genoemd.

In computing, Chord is a protocol and algorithm for a peer-to-peer distributed hash table. A distributed hash table stores key-value pairs by assigning keys to different computers (known as "nodes"); a node will store the values for all the keys for which it is responsible. Chord specifies how keys are assigned to nodes, and how a node can discover the value for a given key by first locating the node responsible for that key.

## 3 Theorie

Het Chord algoritme (ontwikkeld aan MIT) is een protocol voor een distributed hash-table. Hierbij bevinden zich verschillende nodes zich in een "ring". Een nieuwe ring kan de node joinen, waardoor andere nodes hun metadata moeten aanpassen. Elke node bewaart deze metadata in de vorm van een finger-table, en deze metadata betreft informatie over andere nodes in het netwerk. In deze finger-table bevinden zich **m** entries. M heeft de grootte van $\log_2(ring\_size)$ waarbij ring_size het totaal aantal nodes in de ring betreft.

Meer theorie achter Chord wordt duidelijk door het beantwoorden van de vragen uit de syllabus:

## 3.1 Antwoorden op vragen

- **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?**

  Een Chord Overlay Network (CON) is een protocol voor de opslag van een hash table, verdeeld over meerdere computers, een zogenaamde distributed hash table. De structuur is gebaseerd op hash-functies die keys en values gelijk verdelen over elke computer (één computer is één node).

  Elke node onthoudt een *succesor* en een *predecessor* in een ring. De succesor van een node $k$ bevindt zich na de k, en de predecessor huist voor node k. Verder beschikt elke node over een *finger-table*, waarin de hash-codes van de andere nodes in worden opgeslagen.

- **The Chord overlay network is considered scalable. Explain why this is true.**

  Elke node in de finger-table bevindt zich op $n^2$ afstand van de volgende node in de tabel. De finger-table bestaat daarom uit $O(log \cdot n)$ bij een ring van grootte n. Kort gezegd houdt dit in dat wanneer de ring exponentieel groeit, de finger-table enkel lineair groeit, wat het uitermate geschikt maakt voor schaling met grote hoeveelheden.

- **Explain what happens when a node joins an existing Chord overlay network.** Wanneer node n wil verbinden met het netwerk zal eerst de geschikte plek gevonden worden: node m. Om nu deel te nemen aan de ring zal n aan node m vragen wat zijn huidige predecessor is (inclusief zijn finger-table). Deze finger-table zal gemaakt worden door voor elke node-entry in de finger-table van node m te vragen wat zijn predecessor is. Node n zal de nieuwe predessor worden van node m, dus zal de gehele finger-table identiek zijn op de nieuwe node n na. Wanneer n geinitialiseerd is zullen de finger-tables van de overige nodes in de ring geupdate moeten worden. Dit wordt gedaan voor elke node door te kijken wat de huidige predecessor is en waar nodig zal er geupdate worden.

- **Describe the purpose of each of the following meth- ods used in the aforementioned template code.**

  *Stabilize*: stabilize checkt of de predeccesor van de node p wellicht de succesor moet woden van node p.

  *Notify*: laat de succesor weten dat node p bestaat.

  *Fix_finger*: bij het joinen van node p in de ring, zullen de finger-table geupdate worden waar nodig.

- **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.**

Hoewel de finger-table per node in een ring van 5 gelijk is aan $\log_2 5 = \approx 2.32$ hebben we er voor gekozen de weergegeven entries op 5 te houden.

Node 4 (predecessor 19):

| **Finger** | **Successor** |
|---|---|
| 5 | 7 |
| 6 | 7 |
| 8 | 17 |
| 12 | 17 |
| 20 | 4 |

Node 7 (predecessor 4):

| **Finger** | **Successor** |
|---|---|
| 8 | 17 |
| 9 | 17 |
| 11 | 17 |
| 15 | 17 |
| 23 | 4 |

Node 17 (predecessor 7):

| **Finger** | **Successor** |
|---|---|
| 18 | 18 |
| 19 | 19 |
| 21 | 4 |
| 25 | 4 |
| 1 | 4 |

Node 18 (predecessor 17):

| **Finger** | **Successor** |
|---|---|
| 10 | 19 |
| 20 | 4 |
| 22 | 4 |
| 26 | 4 |
| 2 | 4 |

Node 19 (predecessor 18):

| **Finger** | **Successor** |
|---|---|
| 20 | 4 |
| 21 | 4 |
| 23 | 4 |
| 27 | 4 |
| 3 | 4 |

- **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.**

Node 6 (predecessor 22):

| **Finger** | **Successor** |
|---|---|
| 7 | 7 |
| 8 | 22 |
| 10 | 22 |
| 14 | 22 |
| 22 | 22 |

Node 7 (predecessor 6):

| Finger | Successor |
|---|---|
| 8 | 22 |
| 9 | 22 |
| 11 | 22 |
| 15 | 22 |
| 23 | 6 |

Node 22 (predecessor 7):

| Finger | Successor |
|---|---|
| 8 | 6 |
| 9 | 6 |
| 11 | 6 |
| 15 | 6 |
| 23 | 6 |

- **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.**

Node 22 (predecessor 7):

| Finger | Successor |
|---|---|
| 23 | 22 |
| 24 | 7 |
| 26 | 7 |
| 30 | 7 |
| 6 | 7 |

Node 7 (predecessor 22):

| Finger | Successor |
|---|---|
| 8 | 22 |
| 9 | 22 |
| 11 | 22 |
| 15 | 22 |
| 23 | 7 |

# 4 Experimenten

Hoewel het idee was om een chord network te simuleren en te zien hoe dit netwerk in de praktijk omgaat met failures en nieuwe nodes, is dit niet helemaal gelukt, aangezien we de chord-implementatie niet volledig werkend hebben gekregen. In eerste instantie hebben we geprobeerd de pseudocode gegeven op Wikipedia te volgen. Hier ging echter iets fout bij het joinen van een netwerk zonder andere nodes: de join functie leidde tot een infinite loop, aangezien de node consistent als zijn eigen predecessor werd aangewezen, waardoor de maximum recursion depth snel bereikt werd. Vervolgens hebben we de oude implementatie weggegooid (inclusief gegeven code bij de opdracht) en hebben we geprobeerd het algoritme opnieuw te implementeren, strict de pseudocode op Wikipedia volgend. Hier ontstond een nieuw probleem, namelijk dat elke node als fingers vier keer node 0 had, zelfs na het joinen van alle 16 nodes. Ook

bij andere ring sizes bleven de fingers hetzelfde probleem hebben.

Vervolgens is deze code weer weggegooid, en besloten we de pseudocode in de op Blackboard gelinkte Chord paper te implementeren in plaats van de Wikipedia code. Deze code zorgde in eerste instantie wederom voor problemen met een infinite loop, maar deze bleek te komen door onjuist overnemen van de intervals in de pseudocode. Na dit gefixt te hebben, bleek de code helaas nog steeds niet optimaal te werken. De waardes in de finger tables van de nodes was incorrect, en zorgde ervoor dat niet elke node vanuit iedere node te bereiken was, wat uiteraard wel de bedoeling is. Een oplossing hiervoor hebben we niet kunnen vinden, en de in de opdracht gegeven simulatie was hierdoor ook niet mogelijk. Wel hebben we door middel van de `time` library uit python de performance van de Chord implementatie gepeild.

```
- Setting up nodes
-- Set up nodes: 6.0558319091796875e-05s
-- Joined root node: 0.0007960796356201172s
-- Stabilized ring: 0.0003612041473388672s
```

Deze tijden zijn gemeten bij het runnen van de in de appendix gegeven `chord.py`. Bij het runnen van de code zal ook een lijst met alle nodes en hun finger table geprint, al is deze door de eerder genoemde onjuistheid niet bij de bovenstaande benchmark bijgevoegd.

# 5 Conclusie en Discussie

Na eerst 4 weken lang verschillende type databases te ontdekken hebben we onze de laatste tijd bezig gehouden met het bouwen van verschillende functies van een eigen database software. We hebben hierdoor een beter inzicht gekregen over wat een database software inhoudt, welke technieken worden toegepast bij het maken van een database, en (in dit geval) welke functies er problemen of uitdagingen veroorzaken bij het ontwikkelen van een database.

Een consistent en efficiënt algoritme voor een distributed hash table is erg belangrijk bij het maken van een database met scalability over meerdere nodes, en Chord is daarin één van de meest gebruikte en gangbare. Zoals eerder genoemd werd, ging het zelf implementeren van dit Chord algoritme niet zoals gehoopt. Zelfs binnen een simulatie (dus zonder rekening te hoeven houden met networking, exchange van data, failures, latency, etc. die bij een echt netwerk komen kijken) is het ons niet gelukt een werkende Chord implementatie te schrijven. Bij het gebruik van Chord in een voor daadwerkelijk gebruik bedoelde database zou hiervoor zeker gekeken kunnen worden naar één van de reeds bestaande Open-source Chord implementaties, zodat er geen mogelijke bugs en inconsistenties ontstaan door een onjuiste implementatie van het algoritme.

Zelfs al zou onze huidige Chord implementatie naar verwachting werken, is deze toch niet voldoende om in de praktijk werkend te zijn. Ten eerste is het nodig om een wrapper te schrijven waardoor data ook daadwerkelijk bijgehouden kan worden in de nodes, verdeeld kan worden over het hele netwerk, en de lookup functie voor keys hier ook gebruik van maakt. Verder is het nodig een achtergrond-taak te hebben die het netwerkt controleert, de `fix_fingers()` en `stabilize()` functies periodiek aanroept, en opmerkt wanneer er failures plaatsvinden. Het hele begrip "alive node" en "failure" verandert ook bij het converteren naar een daadwerkelijk netwerk, aangezien en node die niet connected is tot het netwerk onmogelijk kan zeggen dat hij niet meer alive is. Een mechanisme hiervoor moet ook geschreven worden.

Kortom, er komt veel kijken bij de ontwikkeling van een nieuwe database-software, en met de grote hoeveelheid keuzes reeds beschikbaar is het belangrijk vast te stellen wat de database goed moet kunnen voor gebruik in een project, en een kritische afweging te maken of de grote hoeveelheid extra moeite die nodig is voor het bouwen van een nieuwe database-software het waard is voor het project, of dat een al bestaande database volstaat.

# 6 Appendix

## 6.1 chord.py

```
from time import time

import math

# Code based on pseudocode found on wikipedia, as well as the pseudocode
# found in the paper about chord provided on blackboard.

# Wikipedia link: https://en.wikipedia.org/wiki/Chord_(peer-to-peer)

class Node:
    ring_size = 16
    m = int(math.log(ring_size, 2))

    def __init__(self, node_id):
        self.predecessor = None
        self.id = node_id
        self.fingers = [self for _ in range(self.m)]

    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 find_successor(self, node_id):
        n = self.find_predecessor(node_id)
        return n.fingers[0]

    def find_predecessor(self, node_id):
        n = self
        while not self.in_range(node_id, n.id+1, n.fingers[0].id+1):
            n = n.closest_preceding_finger(node_id)
        return n

    def closest_preceding_finger(self, node_id):
        for i in reversed(range(self.m)):
            if self.in_range(self.fingers[i].id, self.id+1, node_id):
                return self.fingers[i]
        return self

    def join(self, node):
        if node:
            self.init_finger_table(node)
            self.update_others()
        else:
            for i in range(self.m):
                self.fingers[i] = self
            self.predecessor = self

    def init_finger_table(self, node):
        self.fingers[0] = node.find_successor(self.id + 1)
        self.predecessor = self.fingers[0].predecessor
        self.fingers[0].predecessor = self
        for i in range(self.m-1):
            start = (self.id + 2 ** i) % (2 ** self.m)
            if self.in_range(start, self.id, self.fingers[i].id):
                self.fingers[i+1] = self.fingers[i]
            else:
                node_id = (self.id + 2 ** (i + 1)) % (2 ** self.m)
                self.fingers[i+1] = node.find_successor(node_id)

    def update_others(self):
        for i in range(self.m):
            p = self.find_predecessor(self.id - 2 ** i)
            p.update_finger_table(self, i)
```

```
    def update_finger_table(self, node, index):
        if self.in_range(node.id, self.id, self.fingers[index].id):
            self.fingers[index] = node
            p = self.predecessor
            p.update_finger_table(node, index)

    def stabilize(self):
        x = self.fingers[0].predecessor
        if self.in_range(x.id, self.id+1, self.fingers[0].id):
            self.fingers[0] = x
        self.fingers[0].notify(self)

    def notify(self, node):
        if self.predecessor is None or \
        self.in_range(node.id, self.predecessor.id+1, self.id):
            self.predecessor = node

    def fix_fingers(self):
        for i in range(self.m):
            self.fingers[i] = self.find_successor(self.id + 2 ** i)

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

    def print_fingers(self):
        print('Finger table for node #{}: {}'.format(self, self.fingers))
        print('Predecessor: {}'.format(self.predecessor))

if __name__ == '__main__':
    print("- Setting up nodes")
    timer = time()
    nodes = [Node(i) for i in range(Node.ring_size)]
    print("-- Set up nodes: {}s".format(time() - timer))
    timer = time()
    nodes[0].join(None)
    for i in range(1, len(nodes)):
        nodes[i].join(nodes[0])
    print("-- Joined root node: {}s".format(time() - timer))
    timer = time()
    for node in nodes:
        node.stabilize()
        node.fix_fingers()
    print("-- Stabilized ring: {}s".format(time() - timer))
    for n in nodes:
        n.print_fingers()
```