O(1) zoekmethoden: Hash technieken

Dr. D.P. Huijsmans
24 okt 2012
Universiteit Leiden LIACS

Snel zoeken in ongesorteerde data

- Vroege computer toepassingen waren vaak gebaseerd op grote gesorteerde bestanden;
- Gesorteerd om in 1 doorloop (periodieke run) acties te ondernemen en wijzigingen op records aan te brengen
- Huidig gebruik is vaker momentaan wijzigingen en/of acties per record at random stuk voor stuk

- Data kan ongesorteerd blijven
- Hoe kunnen we snel een ongesorteerd record via z'n sleutel (key) aan een geheugeningang koppelen voor O(1) snelle benadering?

KEY:
UNIEK LABEL VOOR GROEP GEGEVENS

- Vaak wordt aan een groep gegevens (b.v. personeelsrecord) een uniek id toegevoegd
- Bewerkingen op gegevens binnen een record gaan via dit unieke id ofwel KEY (b.v. studentID)
- Vaak kan ook aan een groep gegevens rond een persoon, product of gebeurtenis een unieke key gevormd worden door een combinatie van veldwaarden (b.v. akte burgerlijke stand unieke key m.b.v. geboorteplaats, geboortedatum, naam)
- **Hoe kunnen we het gegeven dat een key uniek (of bijna uniek is) uitbuiten bij snel en compact opslaan en terugzoeken in computergeheugen?**

Zoek complexiteit

- Om de juiste key uit N waarden te zoeken moet je
- O(N): Gemiddeld N/2 keys doorlopen in een ongesorteerd array of een linked list
- O(logN): Gemiddeld log N keys in een gesorteerd array of BST
- O(1)?: Kan het niet in 1 stap?

- Onder welke omstandigheden kun je van een key direct het geheugenadres weten?

Zoek complexiteit

- Om de juiste key uit N waarden te zoeken moet je
- O(N): Gemiddeld N/2 keys doorlopen in een linked list
- O(logN): Gemiddeld log N keys in een BST
- O(1)?: Kan het niet in 1 stap?

- Het kan in 1 stap:
- Als de opgeslagen waarde (key) een 1-op-1 relatie heeft met het geheugen adres waar die waarde is opgeslagen
- Een hash functie is een formule/functie die een keywaarde afbeeldt op een geheugenadres welke we hash-index noemen

Voorbeeld directe link tussen waarde (key) en adres (tabel index)

- Postcode bv 1021 AE
- Stel elke postbode heeft een stel 4-cijferige wijken dan kun je direct opzoeken welke postbode bij een postcode hoort door in een wijktabel met index [1000-9999] d.m.v. het 4-cijferig deel v/d postcode direct de naam van de betrokken postbode op te zoeken:
- postbodetabel
- index waarde
- 1000 ……
- ……
- 1020 Dorknoper
- 1021 van den Bergh
- 1022 van den Bergh
- 1023 Verhulst
- ……
- 9999 ……

Snelheid?
Geheugengebruik?
Wat kan beter?

Voorbeeld directe link tussen waarde (key) en adres (tabel index)

- Postcode bv 1021 AE
- Stel elke postbode heeft een stel 4-cijferige wijken dan kun je direct opzoeken welke postbode bij een postcode hoort door in een wijktabel met index [1000-9999] d.m.v. het 4-cijferig deel v/d postcode direct de naam van de betrokken postbode op te zoeken:

- postbodetabel
- index waarde
- 1000……
- ……
- 1020 Dorknoper
- 1021 van den Bergh
- 1022 van den Bergh
- 1023 Verhulst
- ……
- 9999……

Hash-index	waarde
0	
……	……
20	Dorknoper
21	van den Bergh
22	van den Bergh
23	Verhulst
……	……
8999	……

POSTCODE HASH-TABEL

- In dit eenvoudige geval is de hash-tabel voor de postcodes de oude postcode tabel min de eerste 1000 plaatsen (geheugenbesparing)
- Berekening van de hash-index:
- H(postcode)=postcode -1000 (key-1000)

- Als niet alle postcodes in [1000..9999] bestaan zou een ingewikkelder hash-functie de bestaande postcodes op een kleiner bereik kunnen afbeelden (als b.v. alleen even postcodes zouden bestaan: H(postcode)=(postcode-1000)/2 geeft hash-index voor postcode hash-tabel met half zoveel plaatsen

Een volledige postcode tabel

- Nemen we ook de 2 letters van de postcode mee dan zijn er niet 9000 tabel entries nodig maar
- 9000x26x26 ~ 6.084.000 miljoen entries
- De vorming van een index (sleutel) vanuit een postcode ccccll kan dan plaatsvinden door bij elk 4 cijferig deel een reeks van 26x26=676 entries te declareren waarbinnen de 2-letterige combinatie wordt afgebeeld
- Een alfabetische reeks letters [a..z] kan via de plaats in de ASCII Latin1 codering Ord(letter) als volgt in een getal met bereik [1..26] worden omgezet

ASCII Latin1 codering hoofdletters

Kar	dec	hex	Kar	dec	hex	Kar	dec	hex	Kar	dec	hex
@	64	40	A	65	41	B	66	42	C	67	43
D	68	44	E	69	45	F	70	46	G	71	47
H	72	48	I	73	49	J	74	4a	K	75	4b
L	76	4c	M	77	4d	N	78	4e	O	79	4f
P	80	50	Q	81	51	R	82	52	S	83	53
T	84	54	U	85	55	V	86	56	W	87	57
X	88	58	Y	89	59	Z	90	5a	[	91	5b

- De decimale waarde van een hoofdletter wordt met ord(kar)-64 afgebeeld op [1..26]

Sleutelformule voor index volledige postcodetabel

- De volledige sleutelformule voor de index gegeven de postcode CCCCLL wordt nu:
- Part1=CCCC-1000 (postcodes beginnen bij 1000AA)
- Part2=ord(L1)-65
- Part3=ord(L2)-64
- Index=part1*676+part2*26+part3

- Hiermee gaat postcode 1000AA -> index 1
- En postcode 9999ZZ -> index 6048000
- (als index ε [1,max])

Tabel te groot?

- In de praktijk komen situaties voor waarbij voor een unieke identificatie in een bestand veel meer mogelijkheden zijn geschapen dan er gerealiseerd zijn waardoor een tabel voor alle mogelijke indexen te groot kan zijn (of te kostbaar)
- Burger Service Nummer: een 9 cijferig uniek persoonsnummer
- Bereik [000000000..999999999] 1 miljard indexen
- Er zijn maar ~ 16 miljoen Nederlanders
- Naar verwachting is maar 1 op de ~64 nummers gebruikt

Verkleinen tabelgrootte met mod

- M.b.v. de modulo functie zouden we de BSNtabel index kunnen verkleinen tot
- bsnindex = bsn mod 16 miljoen
- De bsnindex zou dan een bereik van ~[0..16 miljoen] hebben
- Als de uitgedeelde bsn nummers at random uit de totale reeks waardes zijn gekozen verwachten we gemiddeld precies 1 voorkomen per gereduceerde bsnindex en zou het precies 1 opzoekactie per bsn kosten
- Dit is het best denkbare geval

Slechtst denkbare geval botsingen

- Als de bsn nummers beginnend bij 000000000 opeenvolgend zouden zijn uitgedeeld zou een index mod 16 miljoen ook voldoen
- Toch zijn er omstandigheden denkbaar waarom er vaker bsn nummers die mod 16 miljoen van elkaar verschillen zijn uitgedeeld; in het slechtst denkbare geval zouden er 64 bsn nummers op dezelfde bsnindex afgebeeld worden.
- Dit creëert botsingen (collisions) in de BSNtabel
- Botsingen kun je op een aantal manieren oplossen, beter is het om ze te voorkomen

Voorkomen van botsingen

- Na analyse van de bezetting van een identificatie reeks een betere index formule opstellen die wel zo goed mogelijk 1 op 1 afbeeldt:
- Als b.v. de bsn nummers vanaf 0 oplopend zijn uitgedeeld is een beste bsnindex:
- Bsnindex = bsnnummer
- Tabel beperken tot [000000000..16000000]

Perfecte Hash tabel

- Doel van een hash index is:
- Een (pseudo) random verdeling van de sleutels over de hash tabel indexen te realiseren
- als de grootte van de hashtabel gelijk is aan het aantal te hashen keys is de verwachte bezetting 1

- Perfecte hash index: elke hashsleutel komt 1 maal voor (loadfactor $\alpha = 1.0$)
- (geen opzoektijd verspild)

Perfecte minimale Hash tabel

- Perfecte hash index: elke hashsleutel komt 1 maal voor **en**
- Minimale perfecte hash index: het aantal hashtabel entries is gelijk aan het aantal voorkomende sleutels
- (noch ruimte, noch opzoektijd verspild)

- Voor het hashen van strings zijn speciale minimale perfecte hash functies voorgesteld:
- - Cichelli's methode (zie Drozdek)
- En varianten daarop

Aantal mogelijke hash functies

- Een hash functie h: Key -> hash-index
- Invoer n elementen
- Uitvoertabel m elementen
- Voor n <= m geldt:
- - m^n functies mogelijk
- Aantal perfecte hash functies:
- - m!/(m-n)!
- Naarmate m>>n is het mogelijk aantal perfecte hash functies een zeer klein percentage van alle mogelijke functies en kan het lastig zijn er een formule vorm voor te vinden

PAUZE

Oplossen van botsingen

- Als de verwachte bezetting >1 is treden er zeker botsingen op, die we kunnen oplossen door een verbonden lijst van keys per hashindex toe te staan
- Als de verwachte bezetting <1 is kan uitgeweken worden naar een naastgelegen hashindex of er kan een tweede hashsleutel bepaald worden

- Zo zou een hashtabel van 9000 voor de volledige postcode met hashsleutel CCCC-1000 een verwachte bezetting van volledige postcodes hebben van max 676 stuks (in de praktijk maar ~72 stuks omdat lang niet alle mogelijke lettercombinatie toegelaten en/of gebruikt zijn)
- We zouden dan moeten rekenen op een linked list van ong 72 lang bij volledige vulling

Eis aan/t.g.v. uitwijkmanoeuvres

- Als de kans op bezetting van een hash key <1 is, kan i.p.v. een verbonden lijst per hash index uitgeweken worden naar een alternatieve index
- Eis hierbij is dat zowel bij plaatsing van een hash key als bij terugzoeken dezelfde volgorde van alternatieve indexen geprobeerd wordt
- Als een hash methode hieraan voldoet spreken we van Open Adressering

- Een andere eis bij uitwijkmanoeuvres is dat een eenmaal toegevoegde key niet verwijderd wordt

Lineair hashen

- Lineair hashen is zo'n open adressering:
- Wanneer een plek in de hash tabel bezet is wordt de voorganger/opvolger geprobeerd (cyclisch):
- Gegeven: Tabel T[maxind]; h(K)=K mod maxind
- Plaatsen en terug/verder zoeken gaat startend met h(K)

VOORBEELD LINEAIR HASHEN

K ε [0..160] h(K)=K mod 16
Input: 4,7,11,16,22,27,41,72,92,28,.....
h(K): 4,7,11, 0, 6,11, 9, 8, 12,12,....
Plaatsing gaat 1e keer goed tot K=27 met h(K)=11 die uitwijkt naar 10

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16				4		22	7	72	41	27	11	92			

Bij 28 treedt een conflict op met 92 die ook op 12 wordt afgebeeld;
Omdat de 6 buren eronder ook al bezet zijn komt 28 op de eerste vrije plaats -> 5

Nadeel lineair hashen: er treedt primair clusteren op; aaneengesloten reeksen hash indexen worden bezet; de kans op bezetting van een nog lege index door een volgende key gaat sterk variëren

Vermijden primair clusteren: dubbele hash functie

- Om te zorgen dat een alternatieve hash index voor keys die een zelfde hash index krijgen (synoniemen) als alternatief niet weer eenzelfde alternatief krijgen aangeboden, kunnen we het beste een tweede onafhankelijke hash functie invoeren die een plaats t.o.v. het al bezette adres bepaalt (de stap of probe functie p)

Mogelijke hash functies

- Digit selectie:
- Stel Key= $d_0d_1d_2d_3d_4d_5d_6d_7d_8d_9$
- Stel d ε [0..9] dan
- h(Key) = $d_id_jd_k$ met i,j,k ε [0..9] is een tabel van max 1000 groot

- Wat is nu de beste keus voor i,j,k als een random verdeling van h(Key) het doel is?

Mogelijke hash functies

- Digit selectie:
- Stel Key= $d_0d_1d_2d_3d_4d_5d_6d_7d_8d_9$
- Stel d ε [0..9] dan
- h(Key) = $d_id_jd_k$ met i,j,k ε [0..9] is een tabel van max 1000 groot

- Beste keus is afhankelijk van de verzameling sleutels waarop de hash moet worden toegepast
- Digit analyse brengt uitkomst:
- Kies 3 digits die het meest uniform verdeeld zijn

Voorbeeld digit selectie 2e jaars

- De volgende 32 7-cijferige nummers zijn van onze 2e jaars informatica studenten
- 1034367 1045091 1017713 1080644 1052616 1047493 1075152 1023144 1045105 934615 1021869 1015265 1045113 1047515 1085298 1045121 1075160 1045148 704598 1068423 1045156 1075659 1023160 1045164 1080652 1036718 1023187 1019872 1073192 1014528 1021931 1055305

	d1	d2	d3	d4	d5	d6	d7
0	2	30	1	2	1	2	2
1	30	0	4	2	12	6	3
2	0	0	5	5	2	3	4
3	0	0	3	4	2	1	4
4	0	0	9	4	2	3	3
5	0	0	2	13	3	4	5
6	0	0	1	1	5	6	2
7	0	1	4	3	2	1	2
8	0	0	3	1	2	1	5
9	0	1	0	1	1	5	2

Tabel geeft frequentie per digit positie

Vervolg voorbeeld 2e jaars D_6D_7

- Stel we willen het bereik van de tabel tot 100 beperken [0..99]
- Statistisch gezien zijn d_6 en d_7 het meest normaal verdeeld zodat $h(d_1d_2d_3d_4d_5d_6d_7)= d_6d_7$ genomen zou kunnen worden.
- Dit is in wezen niets anders dan h(Key)=key mod 100 opgeleverd zou hebben!
- De volgende 14 student-ID's leveren hierbij 7 dubbelen op in de tabel van 100:
- 1045105 1055305 1017713 1045113 934615 1047515
- 1023144 1080644 1075152 1080652 1023160 1075160
- 704598 1085298

Folding als hash functie

- $h(Key)= h(d_1d_2d_3d_4d_5d_6d_7)=d_1+d_2+d_3+d_4+d_5+d_6+d_7$
- Range [13..33] Dit levert 8 2-4x dubbelen op (25x)

13	1023160	**nil**			
14	1045121	nil			
15	1023144	1045113	nil		
16	1045105	nil			
17	1021931	nil			
18	nil				
19	1055305	nil			
20	1015265	1017713	1045091	1075160	nil
21	1014528	1045164	1052616	1075152	nil
22	1023187	1045156	1080652	nil	
23	1045148	1047515	1073192	1080644	nil
24	1034367	1068423	nil		
25	nil				

Gem.bezetting ~1.52

Gesorteerde linked list per hash index

- De methode om een verbonden lijstje te maken per mogelijke hash-index heet chaining
- Door nieuwe entries op een hash tabel gesorteerd in te voegen in een bestaand linked lijstje
- Besparing op zoeken tot key gevonden (Key in hash-index lijstje > gezochte) of
- Binary search in gesorteerde lijstjes

Folding als hash functie - 2

- Per 2 cijfers:
- h(Key)= h($d_1d_2d_3d_4d_5d_6d_7$)=0d_1+d_2d_3+d_4d_5+d_6d_7
- Range [48..213] met 3 dubbelen
- 1034367 1068423 (114)
- 1015265 1075160 (119)
- 1019872 1047493 (172)
- Alle 3 dubbele kunnen op 1 eronder uniek afgebeeld worden (folding plus linear probing)

111	112	113	114	115	116	117	118	119	120	121	122
1075152	1045156		1034367	nil	nil	nil		1015265	1045164	1023187	nil
			1068423					1075160			

Gem.bezetting ~ 20 %

Rol van priemgetallen bij hashen

- Een aantrekkelijke (makkelijk berekenbare) hash functie bevat vaak “modulo tabelgrootte”

- Om clustering in hash indexen te voorkomen kan het best als tabelgrootte een priemgetal genomen worden

Mod met priemgetal 37

- Het eerste priemgetal boven 32 is 37
- Effect van h(Key)= Key mod 37:
- Meervoudig voorkomen (21x):
- 1021869 1045105 1052616 (3)
- 704598 1073192 (7)
- 1045113 1068423 (11)
- 1075160 1085298 (14)
- 1015265 1080644 (22)
- 1014528 1045164 (25)
- 1023187 1045091 (26)
- 1017713 1021931 1055305 (28)
- 934615 1034367 1075659 (32)

Mod met priemgetal 97,79

- Mod 97 geeft 4 dubbelen
- 1023160 1075152 (4)
- 1047515 1075160 (12)
- 1017713 1045164 (86)
- 704598 1047493 (87)

- Mod 79 geeft 1 dubbele:
- 1034367 1052616 (20)
- Gemiddelde bezetting ~ 40%
- Mod 79 geeft verreweg het beste resultaat;
- de enige dubbele kan op 1 index lager uniek worden afgebeeld

Midden van kwadraat index

- Een veel gebruikte en simpel te bepalen vaak random verdeelde index is te verkrijgen via:
- Kwadrateren van de keywaarde
- Selecteren van het middenstuk van het berekende kwadraat als hash-index
- Een handige opzet is een 2^n grootte tabel met n-midden bits van de binaire representatie van het kwadraat als hash-index
- M.b.v. mask en shift operaties kan deze binaire index makkelijk en snel berekend worden

Hashing
Donald Knuth's analyse resultaten

- In Knuth's "the Art of Programming" vol.3
- Vergelijkt 3 opzetten:
- - linear probing: 1 hash key, bij botsing volgende lege plek zoeken
- - open addressing: 2e hash key bij botsingen na 1e hash key
- Chained hashing: 1 hash key; lijst van keys met zelfde hash key

- Statistische analyse op basis van load factor α fractie elementen op aantal hash keys

Linear Probing

- Gebruik van 1 hash key met uitwijk naar lege buur

Load factor α	Gemiddelde zoektijd
0.5	1.5
0.6	1.75
0.7	2.17
0.8	3.0
0.9	5.5

$$½(1 + 1/(1 – α))$$

Double hashing

Laad factor α	Gemiddelde zoektijd
0.5	1.39
0.6	1.53
0.7	1.72
0.8	2.01
0.9	2.56

$$(-\ln(1 - \alpha))/\alpha$$

Chained Hashing

Load factor α	Gemiddelde zoektijd
0.5	1.25
0.6	1.3
0.7	1.35
0.8	1.4
0.9	1.45
1.0	1.5
2.0	2.0
4.0	3.0

$$1 + \alpha/2$$

Vergelijking gemiddelde zoektijd bij de 3 Hash methoden

Load factor α	Linear probing	Double hash	Chained hash
0.5	1.5	1.39	1.25
0.6	1.75	1.53	1.3
0.7	2.17	1.72	1.35
0.8	3.0	2.01	1.4
0.9	5.5	2.56	1.45
1.0	Kan niet	Kan niet	1.5
2.0	Kan niet	Kan niet	2.0
4.0	Kan niet	Kan niet	3.0

Effectiviteit van chained hashing is onverwacht goed

Samenvatting
Hashen versus sorteren

- Sorteren laat snel zoeken O(logN) toe op geordende elementen
- Hashen maakt direct adresseren O(1) mogelijk op een verzameling ongeordende elementen
- Perfect hash genereert unieke indexen
- Perfect minimal hash heeft minimaal nodige tabel voor deze unieke indexen nodig
- Chained hashing snelst, maar linked lists per hash index nodig