Introductie tot Multimediadatabases

Michel Brabants

12 augustus 2005

Ik bedank mijn promotor Professor doctor Marc Gyssens en co-promotor doctor Sofie Haesevoets voor hun hulp bij het schrijven van dit proefschrift. Ik wil ook alle gezinsleden en overige familie bedanken voor hun steun doorheen de jaren.

Inhoudsopgave

1 Inleiding **5**

2 Enkele voorafgaande begrippen **7**
2.1 Minkowski-Som . . . 7
2.2 Uniformiteit en fractale dimensie . . . 9
2.3 Opmerkingen . . . 13
2.4 Samenvatting . . . 16

3 The Curse of Dimensionality **16**
3.1 Boundary effects . . . 16
3.2 NN-afstand . . . 20
3.3 Indexen die clusteren/partitioneren . . . 24
3.4 Samenvatting . . . 27

4 Indexen **27**
4.1 SS-tree . . . 28
4.1.1 Structuur . . . 29
4.1.2 Operaties . . . 30
4.2 SR-tree . . . 32
4.2.1 Motivatie . . . 32
4.2.2 Structuur . . . 34
4.2.3 Operaties . . . 34
4.2.4 Performantie . . . 35
4.3 VA-file (vector approximation-file) . . . 35
4.3.1 Motivatie . . . 35
4.3.2 Structuur . . . 36
4.3.3 Operaties . . . 36
4.3.4 Performantie . . . 36
4.4 A-tree (approximation tree) . . . 37
4.4.1 Motivatie . . . 37
4.4.2 Structuur . . . 38
4.4.3 Operaties . . . 41
4.4.4 Performantie . . . 41
4.5 Tree Striping . . . 42
4.5.1 Motivatie . . . 42
4.5.2 Structuur . . . 42
4.5.3 Operaties . . . 44
4.5.4 Performantie . . . 46
4.6 Bulk Loading . . . 49

4.6.1 Bepalen van de hoogte $\boldsymbol{h}$ en de fan-out $\boldsymbol{f}$ 49
4.6.2 Splittree 53
4.6.3 Opsplitsen van de data 54
4.6.4 Bouwen van index en schrijven naar schijf 57
4.6.5 Performantie 58
4.7 Samenvatting 59

5 Queryen **59**
5.1 Single step incrementeel k-NN/range-algoritme 60
5.2 Optimal MultiStep k-NN algoritme 62
5.3 Kostmodel voor NN- en range-queries 66
5.3.1 Kostmodel 66
5.3.2 Kostmodel met boundary effects 72
5.3.3 Kostmodel met boundary effects en fractale dimensie . 75
5.4 Samenvatting 78

6 Concurrency **79**
6.1 Motivatie en inleidende theorie 79
6.2 Problemen en mogelijke oplossingen 80

7 Opslag **81**
7.1 Dynamische blokgrootte 82
7.2 Locality of restructuring 85
7.3 Samenvatting 86

8 Besluit **87**

1 Inleiding

Er is steeds meer informatie beschikbaar in verschillende vormen (audio, video, teksten, foto's, 3d-beelden, ...). Denk hierbij maar aan het internet, politiediensten, nieuwsdiensten, het digitaal archiveren van boeken, ... Deze informatie wordt steeds vaker opgeslagen in databases, waarin vlug en efficiënt iets kunnen vinden handig zou zijn. Men kan hierbij aan verschillende toepassingen denken: bepalen waar een bepaalde persoon voorkomt in een verzameling video's gegeven een foto van de persoon, zoeken naar info over een liedje gegeven het liedje, zoeken naar een tekst gegeven de structuur ervan en een voorbeeldtekst, ...
Het gaat hier dus eigenlijk over het zoeken naar data, gegeven een voorbeeldobject. We zoeken naar de data gelijkend op het voorbeeldobject. Deze data kan eender wat zijn: foto's, video, tekst, geluid, gestructureerde data, ... De data die kan verwerkt worden hoeft zich niet te beperken tot de atomische datatypen zoals dat bij conventiële databases het geval is. Dit soort queries en de verschillende typen complexere data worden niet door conventiële database ondersteund. *Multimedia databases* zijn databases die dit type queries en data efficiënt kunnen behandelen.

Men kan verscheidene manieren bedenken om twee objecten te vergelijken, maar de manier die we hier zullen bespreken is gebaseerd op *feature-transformation*. Hierbij wordt een object (foto, tekst, videoframe(s), ...) omgezet naar een multidimensionale vector. Deze vector wordt ook wel de *feature-vector* van een object genoemd. Een feature-transformation F van een object O naar een D-dimensionale vector wordt geformuleerd als:

$$F : O \to \mathbb{R}^D, D \in \mathbb{N}$$

Een object wordt dus een punt in een vectorruimte $\mathbb{R}^D$. Een eenvoudig voorbeeld hiervan is het opdelen van een foto in verschillende gebieden en van elk gebied de gemiddelde kleurwaarde opslaan in een feature-vector. Voor verschillende media heeft men al transformaties bedacht. Bijvoorbeeld, in de literatuur kan men transformaties vinden voor geluid [28] [35], bio-informatica/tekst [17], foto's [18] [22], video [19], ... In [21] kan men uitleg vinden over transformaties voor tekst, foto's, geluid en video. De vector kan afhankelijk van de methode een hoge dimensionaliteit hebben. Het betreft hier dan vectoren met een dimensionaliteit van 20, 100, 1000, ... Op basis van ervaring opgedaan met multimediadatabases, kunnen we laag-dimensionaal definiëren als een dimensionaliteit van 1 tot en met 6, medium-dimensionaal van 7 tot en met 12 en hoger dan 12 definiëren we als hoog-dimensionaal.

Om het queryen te versnellen wordt er een index voor de vectoren aan-

gemaakt. Het zal blijken dat het niet optimaal is om inverted lists te gebruiken. Bij inverted lists wordt voor elke dimensie een index gecreëerd. De hoge dimensionaliteit van de data is een nieuw gegeven in vergelijking met conventionele databases en speelt een belangrijke rol in de gebruikte methoden, structuren, ... van de database. Bestaande indexstructuren zoals de R-tree [34], kd-tree [34], ... zijn niet bruikbaar in hoog-dimensionale ruimten. Opslagmethoden worden er ook door beïnvloed. Dimensionaliteit zal dan ook een belangrijke rol spelen in dit proefschrift.

Een ruimte met een dimensionaliteit van 16 of hoger is niet zo eenvoudig om voor te stellen. Men heeft misschien de neiging om eigenschappen van een 3-dimensionale ruimte over te brengen naar een hoger-dimensionale ruimte. Alhoewel de voorstelling in een laag-dimensionale ruimte handig kan zijn, zullen we zien dat er bepaalde verschijnselen zijn die zich sterker uiten in een hoog-dimensionale ruimte dan in een laag-dimensionale ruimte. We geven een voorbeeld van een verkeerde intuïtie die men kan hebben:
We bevinden ons in een D-dimensionale ruimte. We stellen de ruimte voor als een genormaliseerde D-dimensionale kubus R. De ribben van de kubus hebben dus lengte 1 en bevinden zich voor elke dimensies in het interval [0,1]. Met ander woorden R=$[0,1]^D$. We definiëren het centrum R_c van een D-dimensionale ruimte als de D-dimensionale vector waarin elk element de waarde 0.5 heeft, dus $R_c(0.5, \ldots, 0.5)$. De vraag is nu: Bewijs dat de volgende stelling waar is of geef een tegenvoorbeeld.
Stelling: *Elke D-dimensionale hyperbol die alle $(D-1)$-dimensionale zijkanten van de D-dimensionale ruimte R raakt of intersecteert, bevat het centrum R_c.*
Men kan zich voorstellen en bewijzen dat de stelling waar is in een 2- of 3-dimensionale ruimte. Voor een 16-dimensionale ruimte is dit echter niet waar. Volgend tegenvoorbeeld toont dit aan: We definiëren S als de 16-dimensionale bol met als centrum de 16-dimensionale vector $S_c(0.3, \ldots, 0.3)$. De Euclidische afstand van S_c tot R_c is $\sqrt{16 \cdot (0.5-0.3)^2} = 0.8$. De straal van S moet een minimale afstand van 0.7 hebben om alle zijden van de ruimte te raken. Dit betekent echter dat S R_c niet bevat en toch alle 15-dimensionale zijkanten raakt.

Men kan uit bovenstaand voorbeeld misschien ook afleiden dat afstand een belangrijke rol speelt. In dit proefschrift wordt hoofdzakelijk de Euclidische afstandsmaat L_2 gebruikt.

We starten met verschijnselen en problemen die optreden in multidimensionale ruimten. Vervolgens, trachten we deze problemen zo goed mogelijk aan te pakken in de context van verschillende onderdelen van een database:

indexen, zoekalgoritmen, ...

2 Enkele voorafgaande begrippen

Vooraleer we specifieke indexstructuren, algoritmen, ... gaan bespreken, behandelen we problemen en eigenschappen in verband met multidimensionale ruimten.

2.1 Minkowski-Som

Omdat de Minkowski-Som zal terugkomen in verschillende theorieën, bespreken we deze eerst. De Minkowski-Som is het vergroten van een geometrisch object A met een geometrisch object B waardoor een nieuw geometrisch object ontstaat. De Minkowski-Som is dit nieuwe object. Wij zullen hier echter vooral geïnteresseerd zijn in het volume van de Minkowski-Som. Formeel wordt de Minkowski-Som van twee geometrische objecten A en B gedefinieerd als:

$$A \oplus B = \{a + b \mid a \in A, b \in B\}$$

Het volume van deze Minkowksi-Som noteren we als $Vol_{A \oplus B}$. Doorheen dit proefschrift veronderstellen we altijd dat de objecten,voor de berekening van de Minkowski-Som, eerst naar de oorsprong verplaatst worden. We beschrijven enkele voorbeelden.

Het volume van de Minkowski-Som van twee D-dimensionale hyperbollen, E met straal e en F met straal f, komt overeen met het volume van de hyperbol G met straal $e + f$. Dus,

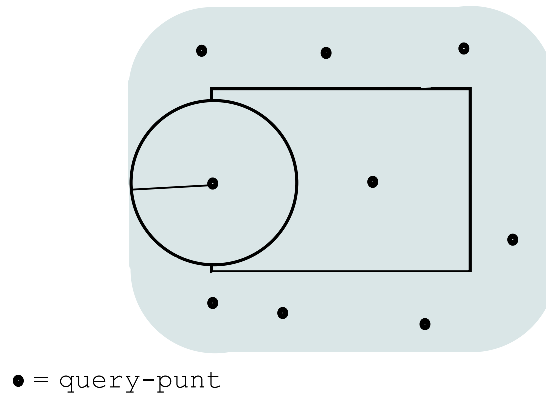
$$\begin{aligned} & Vol_{E \oplus F} = \frac{\sqrt{\Pi^D} \cdot (e+f)^D}{\Gamma\left(\frac{D}{2}+1\right)}, \text{ waarbij} \\ & \Gamma(n+1) = n \cdot \Gamma(n), \\ & \Gamma(1) = 1, \text{ en} \\ & \Gamma(\frac{1}{2}) = \sqrt{\Pi} \end{aligned}$$

Het volume van de Minkowski-Som van twee D-dimensionale hyperbalken O en P komt overeen met het volume van de hyperbalk Q waarbij elke $(D-1)$-dimensionale zijkant van O voor alle $D-1$ dimensies aan beide kanten vergroot wordt met de helft van de lengte van P in de overeenkomstige dimensie. De $D-1$ dimensies zijn dus de dimensies waarin de waarden

van de punten die behoren tot het object, variabel zijn. Dus, zij o_i en p_i, $i \in \{0, \ldots, D-1\}$ de lengte van de zijden van O, respectievelijk P, dan geldt:

$$Vol_{O \oplus P} = \prod_{i=0}^{D-1} (o_i + p_i)$$

Een iets moeilijker geval is het volume van de Minkowski-Som van een hyperbalk U en een hyperbol V. Het volume van de Minkowski-Som van U en V, is het volume van het object gevormd door de bol V langs de zijkanten van U te bewegen zodanig dat het centrum van V zich in de zijkant bevindt (Zie figuur 1). Dit geval zullen we bespreken in hoofdstuk 5.3.



Figuur 1: Minkowksi-Som-volume van een rechthoek en een cirkel

Waartoe dient nu de Minkowski-Som?
Om dit te illustreren moeten we eerst een aantal begrippen definiëren.

Definitie Sferische range-query. Zij X een set van D-dimensionale punten. Een *sferische range-query* met een query-punt q en een straal r geeft de punten van X terug die op een maximale afstand r van q liggen, gebruikmakend van een afstandsfunctie $d(a, b)$. Dus,

$$rangeQuery(q, r) = \{p \in X \mid d(p, q) \leq r\}$$

De hyperbol gevormd door het query-punt/centrum q en de straal r, noemt men de *query-bol*.

Definitie. Een *bounding regio* stelt een gedeelte voor van een D-dimensionale ruimte.

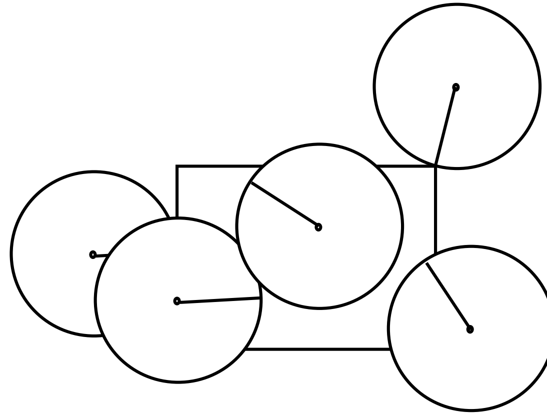
Wij zullen hier werken met convexe bounding regio's zoals hyperbalken en hyperbollen.

Definitie. Een *minimale bounding regio (MBR)* is de bounding regio met het kleinste volume uit de set van alle mogelijke bounding regio's die allemaal eenzelfde verzameling van punten bevatten.

Beschouw nu de volgende vraag: "Gegeven een bounding regio en een range-query, wat is de kans dat de range-query de bounding regio intersecteert? We houden hierbij geen rekening met het feit dat de genormaliseerde ruimte R begrensd is.".
We willen dus weten welk percentage van de mogelijke range-query's de bounding regio kunnen intersecteren. Met andere woorden, we willen weten hoeveel mogelijke hyperbollen de bounding regio kunnen intersecteren. Dit is een query over de intersectie tussen twee regio's. We kunnen deze query omvormen naar een punt-query. Bij een *punt-query* bepalen we of een punt in een bepaalde bounding regio ligt. Daartoe beschouwen we de bounding regio en het centrum van een hyperbol. De hyperbol zal de bounding regio enkel intersecteren als het centrum van de hyperbol in het object, gevormd door de Minkowski-Som van de bounding regio en de hyperbol, ligt. Aangezien we werken in de genormaliseerde ruimte R stelt het volume van het object gevormd door de Minkowski-Som dus procentueel het aantal mogelijke hyperbollen voor die de bounding regio intersecteren. Hierbij nemen we aan dat de querypunten uniform verdeeld zijn. Bijvoorbeeld, alle mogelijke intersectie-queries die de rechthoek in figuur 2 intersecteren, zijn in figuur 1 omgezet naar punt-queries.

2.2 Uniformiteit en fractale dimensie

Uniformiteit In [16] toont men aan dat het niet uniform zijn van de data geen invloed heeft op de kost van NN-queries als er geen correlatie is tussen de dimensies en als de datadistributie gelijkmatig is. Met gelijkmatig bedoelt men dat de concentratie van punten niet veel varieert binnen de Minkowksi-Som van een bounding regio en de querybol met als straal de NN-afstand. De *NN-afstand* is de afstand van een gegeven punt tot het dichtsbijzijnde punt behorende tot de verzameling datapunten. We nemen hier aan dat de query-distributie de data-distributie volgt. Merk op dat als de data niet-uniform is of de query-distributie niet de data-distributie volgt, we er niet vanuit kunnen gaan dat de Minkowski-Som het verwachte aantal punten teruggeeft die we moeten controleren bij een query. Immers, als de data niet uniform verdeeld is, is het niet onmiddellijk duidelijk of we mogen zeggen dat het volume van een bounding regio een weerspiegeling is van het aantal punten



Figuur 2: cirkels die intersecteren met de rechthoek

binnen deze regio. Hetzelfde geldt voor de distributie van de query-punten met betrekking tot de Minkowski-Som.

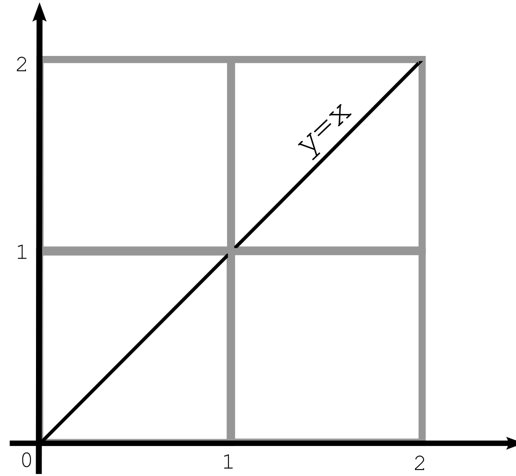
Definitie Nearest neighbour-query. Zij X een set van D-dimensionale punten. Een *nearest neighbour-query (NN-query)* op een query-punt q geeft de dichtstbijzijnde buur van q terug gebruikmakend van een afstandsfunctie $d(a, b)$, die een afstand tussen de punten a en b bepaalt. De dichtstbijzijnde buur van een punt q, is het punt $NN(q) \in X$, waarvoor dus geldt:

$$NN(q) = \{p \in X \mid \forall p' \in X : d(p, q) \leq d(p', q)\}$$

Een NN-query kan voorgesteld worden als een range-query met als straal de (verwachte) NN-afstand $E[NN]$.
We tonen de intuïtie van de stelling uit [16], betreffende de invloed van het niet uniform zijn van de data op de kost van NN-queries. Een theoretisch uitgewerkt voorbeeld kan men vinden in [16] op pagina's 9–12. De kost van een NN-query wordt uitgedrukt in het aantal bounding regio's die moeten onderzocht worden om de nearest neighbour te vinden. In algemene modellen wordt dit berekent door het verwachte aantal bounding regio's te vermenigvuldigen met de kans dat een bounding regio bezocht wordt. Bij een uniforme verdeling van punten wordt er vanuit gegaan dat elke bounding regio hetzelfde volume heeft. We gaan er ook vanuit dat elke bounding regio evenveel punten bevat, waardoor het verwachte aantal bounding regio's gelijk is aan het totaal aantal punten gedeeld door het aantal punten per bounding regio. We willen nu weten als bij het stellen van een NN-query, de kans dat een bounding regio bezocht moet worden nog altijd hetzelfde is als bij een

uniforme verdeling. Deze kans is gelijk aan het aantal query-punten binnen de Minkowski-Som gevormd door een bounding regio en de query-bol met als straal de verwachte NN-afstand ten opzichte van het totaal aantal query-punten. Hieruit volgt dat we moeten aantonen dat het aantal query-punten binnen de Minkowksi-Som van een bounding regio en de hyperbol van de NN-query in het gelijkmatige geval, hetzelfde is als in het uniforme geval. Stel dat de bounding regio en de uitbreiding van de Minkowksi-Som zich bevinden in een regio waarin de concentratie van punten weinig verschilt. Zij de concentratie v keer hoger dan het gemiddelde. Aangezien de concentratie niet erg verschilt binnen deze regio, benaderen we dus een uniforme verdeling. Omdat elke bounding regio evenveel feature-vectoren bevat, is het volume van een bounding regio binnen deze regio gemiddeld v keer kleiner dan het gemiddelde. Immers, de data-distributie binnen deze regio benadert een uniforme verdeling en bij een uniforme verdeling kan het aantal punten binnen het object gebruikt worden om het volume ervan te benaderen. Ook het volume van de Minkowksi-Som-uitbreiding wordt gemiddeld v keer kleiner dan het gemiddelde volume van de Minkowski-Som-uitbreidingen. Dit komt doordat de NN-afstand kleiner wordt door de verhoogde concentratie query-punten. Het volume van de Minkowksi-Som is dus v keer kleiner dan het gemiddelde volume van een Minkowksi-Som, maar de concentratie aan query-punten is wel v keer hoger dan de gemiddelde concentratie. Bijgevolg bevat de Minkowksi-Som in het gelijkmatige geval evenveel query-punten als in het uniforme geval. Zolang de Minkowksi-Som zich dus in dit soort regio bevindt, kunnen we er vanuit gaan dat de Minkowksi-Som een betrouwbare representatie is van de kans dat een bounding regio bezocht wordt.
Bij een range-query hebben we uniformiteit nodig om een betrouwbaar kostmodel op te stellen. Voor het geval dat de data-distributie niet-uniform of niet-gelijkmatig is, zijn er nog geen resultaten bekend.

Fractale dimensie Om het aantal onafhankelijk dimensies van een dataset te bepalen, bestaan er een aantal technieken zoals single-value-decomposition (SVD) en principal component analysis (PCA). Het probleem met SVD en PCA, is dat ze enkel de lineaire afhankelijkheden vinden. Met behulp van de fractale dimensies kan men ook niet-lineaire-afhankelijkheden bepalen. Het principe van de fractale dimensie D_F van een eindig aantal punten is dat we een D_F-dimensionaal object nemen dat we uniform schalen in alle dimensies. Als het aantal punten in dezelfde mate evolueert als de uitbreiding/krimping door de uniforme schaling, dan is D_F de fractale dimensie.
Als we in het voorbeeld uit Figuur 3 schalen volgens twee dimensies, dan neemt de oppervlakte toe met vier, terwijl het aantal punten van het lijnstuk



Figuur 3: illustratie powerlaw met fractale dimensie

op de recht $y = x$ met twee vermeerdert. Als we volgens één dimensie schalen, dan vermeerdert de lengte zoals het aantal punten met twee. De fractale dimensies is dus één hier. Dus, zij $N_{hyperkubus}$ het aantal punten binnen een D-dimensionale hyperkubus K binnen een D-dimensionale ruimte. We zoeken dan naar een formule die weergeeft dat het aantal punten binnen K evolueert naarmate K in alle D_F dimensies uniform schaalt. Zij Vol_{DS} het volume van de D-dimensionale ruimte en zij $\rho_F = N/\left(Vol_{DS}\right)^{(D_F/D)}$ de *fractal point density*, dan is

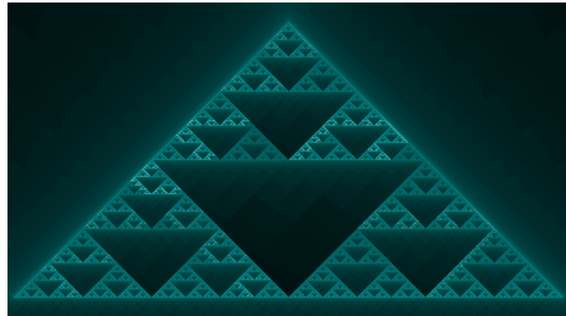
$$N_{hyperkubus} = \rho_F \cdot Vol_K^{\left(\frac{D_F}{D}\right)} \tag{1}$$

Formule 1 is de *powerlaw van de fractale dimensie*. In de formule kan K een ander object dan een hyperkubus zijn dat uniform schaalt.

Definitie Fractale dimensie. De *fractale/intrinsieke dimensie* D_F van een eindig aantal D-dimensionale punten, waarbij D de Euclidische/embedded dimensie is, is de exponent waarvoor formule 1 geldt op voorwaarde dat het centrum van K zich in het deel van R bevindt dat punten bevat.

De fractale dimensie komt van het domein van de fractalen. Een manier om de fractale dimensie te verwoorden is het aantal dimensies dat nodig is

om de basisstructuur van een fractaal voor te stellen. Een set van punten is een *fractaal* als het "self-similarity" vertoont over alle schalen. Hiermee bedoelt men dat de structuur of een deel ervan die gevormd wordt door de punten hetzelfde (exact of statistisch) blijft als men op de structuur inzoomt en uitzoomt. Bij fractale dimensie-reductie probeert men de fractale dimensionaliteit van een dataset te achterhalen. Zo is de fractale dimensie van een cirkel één. Intuïtief kan men dit zien door het feit dat als men helemaal inzoomt op een cirkel, men een lijn heeft. Bij een een cirkel is ook één van de twee dimensies afhankelijk van de andere. De fractale dimensie van een 3-dimensionale bol zal dus twee zijn. Merk op dat de fractale dimensie geen natuurlijk getal hoeft te zijn, het kan ook een positief reëel getal of een complex getal zijn. Zo is de fractale dimensie van de Sierpinski-driehoek 1.5849 (Zie figuur 4).



Figuur 4: Sierpinski driehoek/Sierpinski gasket (gegenereerd met Gnofract4D)

In [10] bekijkt men datasets uit de praktijk en concludeert men dat deze data telkens een fractale structuur heeft. Methoden om de fractale dimensie te berekenen kan men onder andere terugvinden in [3] en [39]. In [11] toont men aan dat de performantie van het zoeken in een multidimensionale ruimte met behulp van structuren die gelijken op de R-tree afhankelijk is van de fractale dimensie in plaats van de Euclidische dimensie.

2.3 Opmerkingen

We geven een aantal opmerkingen om meer inzicht te krijgen in multidimensionale ruimten. We nemen hierbij aan dat de data- en query-distributie voldoet aan de eisen die vermeld zijn in sectie 2.2. De gehele ruimte is een D-dimensionale genormaliseerde kubus R met zijden van lengte 1. De zijden zijn dus voor elke dimensie gelegen in het interval $[0, 1]$. Met andere woorden,

$R = [0,1]^D$.

Wat is nu eigenlijk het verschil tussen zoeken in een laag-dimensionale ruimte en zoeken in een hoog-dimensionale ruimte in de context van nearest neighbour-queries (NN-queries)?

Opmerking 1 Laten we een eenvoudig splitsingsalgoritme beschouwen waarbij een hiërarchische structuur opgesteld wordt door elke dimensie eenmaal in tweeën te splitsen. Dit is dus een ruimte-partitionerende structuur. De KD-tree [34] is een voorbeeld van een structuur die op deze manier de ruimte opsplitst. Voor een D-dimensionale ruimte krijgen we dan 2^D delen. Bijvoorbeeld, indien D=100, dan zijn er $2^{100} \approx 10^{60}$ delen. In een ruimte met 10^6 punten zijn er dan veel lege delen, wat voor veel overhead zorgt tijdens het zoeken, ... Dimensionaliteit is dus iets waarmee men rekening moet houden als men een indexstructuur opstelt.

Opmerking 2 Een ander soort van query die we beschouwen is de range-query. Stel dat we een sferische range-query uitvoeren met als centrum het centrum van de dataruimte R_c en met straal 0.5. Dit is de grootste sferische range-query die we kunnen stellen, zonder dat de straal zich verlegt tot buiten R. De kans dat een punt zich binnen de straal van de query bevindt, is het volume van de hyperbol die de query vormt, gedeeld door het total volume van R. Het volume van de dataruimte R is 1. De kans wordt dan voorgesteld door de formule van het volume van de hyperbol.

$$RelVol = \frac{\sqrt{\Pi^D} \cdot (\frac{1}{2})^D}{\Gamma\left(\frac{D}{2}+1\right)}$$

Voor een stijgend aantal dimensies (stijgende D), wordt het volume van de bol ten opzichte van het totale volume van de ruimte blijkbaar steeds kleiner en kleiner. Dit kan men observeren als men enkele waarden uitrekent.

D	$relVol$
2	0.785
10	0.002
20	$2.461 \cdot 10{-}8$
100	$1.868 \cdot 10^{-70}$

De kans dat we binnen de range van 0.5 nog iets gaan vinden, wordt zeer klein naarmate het aantal dimensies stijgt. Het relatieve volume daalt zeer snel. Elke uitbreiding met een dimensie vergroot het totale volume en telkens daalt het volume van deze hyperbol ten opzichte van het totale volume sterk. Dezelfde vaststelling kunnen we doen voor een balkvormige range-query.

Opmerking 3 Uitgaande van bovenstaande vaststelling, zou het interessant zijn om te weten hoeveel uniform verdeelde punten we zouden nodig hebben om minsten één punt te vinden in de bovenstaande range-query. De kans dat een punt in de range-query valt moeten we dus vergroten tot één, zodanig dat we statistisch zeker zijn dat minstens één punt door de range-query teruggegeven wordt. De bovenstaande formule zegt dat als we N punten hebben in R, $relVol * N$ er statistisch gezien in de range-query zullen vallen. Hoeveel punten hebben we dus nodig zodanig dat $relVol * N = 1$, waarbij N het aantal punten is? Dit is dus $1/relVol$. Volgende tabel geeft weer hoe dit aantal evolueert afhankelijk van het aantal dimensies.

D	aantal punten
2	1.273
4	3.242
10	401.5
20	40631627
100	$5.353 \cdot 10^{69}$

We zien dus dat het aantal in laag-dimensionale ruimten klein is, maar dat dit zeer groot wordt als het aantal dimensies stijgt. We stellen vast dat het aantal punten zeer snel toeneemt. De genormaliseerde ruimte R moet dus al een zeer hoog aantal punten bevatten opdat de gegeven range-query iets zou teruggeven. Dit komt overeen met de vorige formule dat het volume buiten de range-query, steeds groter wordt ten opzichte van het volume van de range-query en dat daar waarschijnlijk procentueel steeds meer en meer punten liggen.

Opmerking 4 Het vergroten van de dimensionaliteit veroorzaakt een exponentiële groei van het volume van de dataruimte als de nieuwe dimensie minstens 2 waarden bevat. Hierdoor komt er dus veel lege ruimte bij.

2.4 Samenvatting

In het begin hebben we een aantal begrippen uitgelegd zoals NN-query, range-query, Minkowksi-Som, ... die nodig zijn om de verdere inhoud van dit proefschrift te begrijpen. We hebben ook de eisen beschreven waaraan de data- en query-distributie moeten voldoen om de Minkowksi-Som te kunnen gebruiken als de verwachte kans dat een bounding region geïntersecteerd wordt. We hebben ook een aantal opmerkingen gegeven om een eerste kennismaking te hebben met hoog-dimensionale ruimten.

3 The Curse of Dimensionality

Er zijn twee belangrijke eigenschappen die gelden in hoog-dimensionale ruimten.

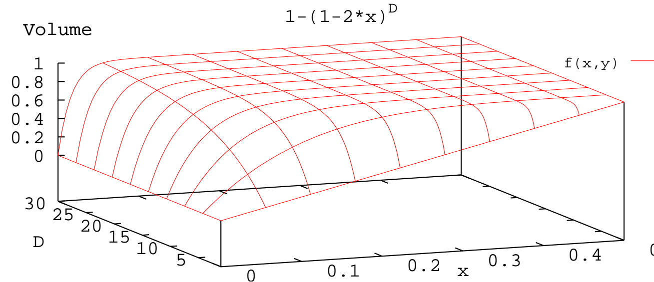
1. het meeste volume van een hoog-dimensionale ruimte bevindt zich aan de grenzen van de dataruimte.
2. het zoeken in indexen die de data(ruimte) partitioneren/clusteren evolueert naarmate het aantal dimensies stijgt naar een lineaire scan.

Deze eigenschappen verstaat men onder de term *The Curse of Dimensionality.*

3.1 Boundary effects

In Voorbeeld 2 van sectie 2.3 hebben we de grootst mogelijk range-query die volledig binnen R ligt besproken. We hebben hierbij gemerkt dat deze query in hoge dimensies maar een kleine kans heeft om een punt te bevatten. Als men alle queries met straallengte $0 \leq r \leq 1$ beschouwt die mogelijk zijn in R, ziet men dus dat er heel wat query-bollen tot buiten R reiken. In sectie 3.2 zullen we zien dat de gemiddelde NN-afstand $E[NN]$ al vlug 0.5 bereikt naarmate D stijgt. We moeten dus het volume gevormd door de query beperken zodat het volledig binnen R ligt. Veel van het volume kan dus wegvallen, waardoor tijdens de uitvoering van de query minder ruimte onderzocht moet worden dan het volledige volume gevormd door de query aangeeft. Afhankelijk van het volume dat wegvalt, moeten er dus minder bounding regio's geraadpleegd worden dan verwacht door een model dat niet met het clippen van het volume aan de randen van R rekening houdt. De kans dat er geclipt moet worden hangt samen met de kans dat een query-punt zich dicht bij de randen van R bevindt. Het volume dat zich binnen een

afstand x van de rand van R bevindt, kunnen we modelleren met de formule $1 - (1 - 2 \cdot x)^D$.



Figuur 5: Verloop van $1 - (1 - 2 \cdot x)^D$

Uit de grafiek in figuur 5 blijkt dat naarmate de dimensionaliteit D stijgt, er veel kans is dat een punt zich op 10% of 20% van de rand van R bevindt. Immers, bijna het volledige volume van R bevindt zich in de rand. Hoe meer men naar het midden van de ruimte gaat, hoe trager de kans stijgt. Er is dus ook meer kans dat de data- en query-punten zich in de rand van R bevinden. Dit kon men ook al merken in Voorbeeld 2 van sectie 2.3 waar men steeds een kleinere kans kreeg dat een punt binnen een hyperbol met straal 0.5 en centrum het centrum van R lag naarmate D steeg. We hebben dus twee feiten

- de NN-afstand in een Euclidische dataruimte bereikt al vlug een lengte van 0.5 naarmate D stijgt
- De kans dat een punt binnen 10% of 20% wordt vlug groot naarmate D stijgt

De gevolgen van deze twee feiten noemt men de *boundary effects*. Eén ervan is dat het clippen van volumes aan de rand van R steeds belangrijker wordt naarmate D_F stijgt.

In hoog-dimensionale ruimten is het dus ook belangrijk om rekening te houden met boundary effects. In laag- en medium-dimensionale ruimten zijn boundary effects niet echt van belang. Er is een uitzondering op het feit dat boundary effects een belangrijke rol spelen met betrekking tot NN-queries in hoog-dimensionale ruimten[1].

In sectie 2.1 hebben we eenvoudige voorbeelden vermeld van het volume van de Minkowski-Som en een algemene uitleg over de toepassing ervan. De

[1] In [9] toont men aan dat het boundary effect weinig invloed heeft op de performantie van NN-queries als

algemene uitleg hield echter geen rekening met het feit dat R begrensd is. Hier werken we een andere toepassing uit die hiermee wel rekening houdt. Als voorbeeld werken we namelijk het volume van de Minkowski-Som van een hyperbalk (als bounding regio) en een hyperkubus (in plaats van een query-bol). We houden hierbij dus rekening met het boundary effect dat de query-hyperkubus geclipt wordt aan de grenzen van de ruimte. We beginnen eerst met het simpelere geval waarin we geen rekening houden met boundary effects. Deze zou men dus kunnen gebruiken als men werkt met een lage of medium dimensionaliteit.

We willen de kans berekenen dat een query-hyperkubus met ribben van lengte l een bounding hyperbalk intersecteert in een D-dimensionale ruimte. We willen dus weten hoeveel procent van de mogelijke query-hyperkubussen de bounding hyperbalk zullen intersecteren. Elke query-hyperkubus kan uniek worden voorgesteld door een query-punt. Een query-punt is een welbepaald punt van een query-hyperkubus, bijvoorbeeld het punt met de hoogste waarden voor elke dimensie, het centrum, ... We nemen het centrum van de query-hyperkubus als query-punt. De query-hyperkubus en de bounding hyperbalk intersecteren elkaar als het centrum van de query-hyperkubus binnen het object B ligt, gevormd door elke zijde van de bounding hyperbalk te vergroten met lengte $l/2$. Het volume van het object B komt dus overeen met het volume van het object gevormd door de Minkowski-Som van de query-hyperkubus en de bounding hyperbalk.
We kunnen dan stellen dat de kans dat er een intersectie plaatsvindt, het volume van het object B is. Dit volume is:

$$\Pi_{j=0}^{D-1}(URC_j - LLC_j + l). \text{ Hierbij is respectievelijk}$$

URC de upper right corner of de vector van de bounding hyperbalk met voor elke dimensie de hoogste waarde en
LLC de lower left corner of de vector van de bounding hyperbalk met voor elke dimensie de kleinste waarde.
In bovenstaande formule houden we echter geen rekening met boundary effects, met name dat de uitbreiding van de bounding hyperbalk met de query-hyperkubus zich gedeeltelijk buiten de dataruimte kan bevinden en dus geclipt moet worden. We nemen aan dat de bounding hyperbalk zich volledig

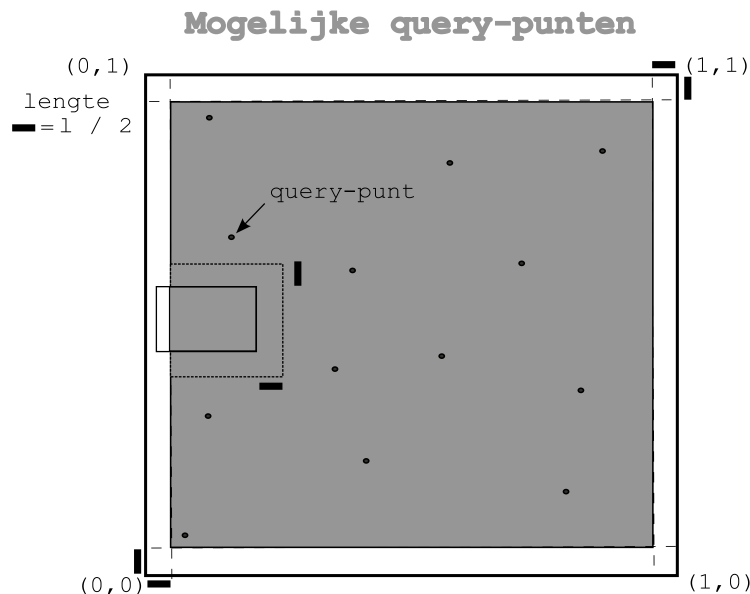
---

$$N \gg C_{eff} \cdot \left(\sqrt[D]{\frac{1}{C_{eff} \cdot Vol^{D}_{hyperbol}(\frac{1}{2})}} + 1 \right)^{D}$$

C_{eff} is het gemiddeld aantal feature-vectoren in de bounding regio van een indexknoop.

binnen de dataruimte bevindt en dat $0 \leq l \leq 1$. We houden alleen rekening met query-hyperbalken die zich volledig in de dataruimte bevinden.
We willen dus weten wat de kans is dat een specifieke bounding hyperbalk geïntersecteerd wordt door een query-hyperkubus met ribben van lengte l. Er is eveneens gegeven dat de bounding hyperbalk en de query-hyperkubus volledig in R liggen.

Ook hier willen we dus het volume berekenen bepaald door de query-punten die de Minkowski-Som intersecteren, maar nu zijn er dus beperkingen. Hoeveel procent van de *mogelijke* query-hyperkubussen intersecteren de bounding hyperbalk?



Figuur 6: geclipt Minkowksi-Som-volume van een hyperbalk en een hyperkubus

1. Ten eerste zullen we bekijken wat de mogelijke query-hyperkubussen/query-punten kunnen zijn. We weten dat query-hyperkubussen die zich niet helemaal binnen de dataruimte bevinden, niet meetellen. Hierdoor zullen de query-punten beperkter zijn in waarden. Namelijk, voor elke dimensie liggen de waarden van het query-punt in het interval $[\frac{l}{2}, 1-\frac{l}{2}]$. Dit betekent dat het totale mogelijke volume van de Minkowski-Som niet één is, maar $(1-l)^D$.

2. Voor elke dimensie moeten we er ook voor zorgen dat het beginpunt van de Minkowski-Som voor een dimensie niet kleiner is dan $l/2$ en het eindpunt niet groter dan $1 - l/2$. Immers, dit is het interval ($[l/2, 1 - l/2]$) waarin de query-punten kunnen liggen, want anders ligt de query-hyperbalk buiten de dataruimte. We beperken ons dus tot de mogelijke query-punten.

Het volume van de Minkowski-Som die rekening houdt met boundary effects is dan

$$P[hyperbalk](l) = \Pi_{j=0}^{D-1} \frac{(min(URC_j + \frac{l}{2}, 1 - \frac{l}{2}) - max(LLC_j - \frac{l}{2}, \frac{l}{2}))}{1 - l}$$

Men kan deze formule nog wat vereenvoudigen. In plaats van bovenstaande URC-gedeelte, kan men ook $min(URC_j, 1 - l)$ schrijven. Dit verandert niks aangezien men de verhoging van URC met $l/2$ weglaat en hierbij het minimum met evenveel verlaagd. Om hetzelfde resultaat te bekomen, moet men ook LLC met $l/2$ verlagen, zodat men voor LLC $max(LLC_j - l, 0)$ bekomt. Men bekomt dan de formule:

$$P[hyperbalk](l) = \Pi_{j=0}^{D-1} \frac{(min(URC_j, 1 - l) - max(LLC_j - l, 0))}{1 - l}$$

Deze laatste formule vindt men terug in [33].

3.2 NN-afstand

In dit hoofdstuk proberen we inzicht te krijgen in de manier waarop de NN-afstand evolueert naarmate het aantal dimensies D stijgt. Daartoe proberen we een formule te ontwikkelen voor de verwachte NN-afstand $E[NN]$. We houden eerst geen rekening met boundary effects, namelijk het clippen aan de rand van R, die een grote kans hebben om voor te komen in hoog-dimensionale ruimten.

Berekening $E[\boldsymbol{NN}]$ Bij het berekenen van $E[NN]$ is volgend gegeven belangrijk: gegeven een D-dimensionale ruimte met N punten en een D-dimensionale hyperbol Y met een straal r. Als er geen punt in de hyperbol ligt, dan is de NN-afstand groter dan r. Hoeveel kans is er dat de NN-afstand kleiner dan of gelijk is aan r, met andere woorden hoeveel kans is er dat er minstens één punt in Y ligt? Dit is de som van de kansen dat één punt erin ligt, twee punten erin liggen, ..., N punten in Y liggen. Dit

is het complement van de kans dat er geen punten in Y liggen, met andere woorden het complement van de kans dat N punten niet in E liggen. De kans dat N punten niet in Y liggen, kan men schrijven als $\left(1 - Vol_{Y(r)}\right)^N$. Het complement hiervan is dus de kans dat er minstens één punt in Y ligt. Dit is dus

$$\begin{aligned} P_{E[NN]\leq r}(r) &= 1 - \left(1 - Vol_{Y(r)}\right)^N \\ &= 1 - \left(1 - \frac{\sqrt{\Pi^D} \cdot r^D}{\Gamma\left(\frac{D}{2}+1\right)}\right)^N \end{aligned}$$

Deze functie is de *probabiliteit-distributiefunctie*.
De kans duidt aan in hoeveel procent van de gevallen Y minstens één punt zal bevatten. Dit is ook de kans dat $E[NN] \leq r$. Op deze laatste manier kunnen we de kans ook verwoorden als $P_{E[NN]\leq r}(r) = \int_0^r P_{E[NN]=x}(x)$. De afgeleide van de functie $P_{E[NN]\leq r}(r)$ geeft ons dan $P_{E[NN]=x}(x)$ waarbij x kan vervangen worden door de straal-variabele r. $P_{E[NN]=r}(r)$ is de *probabiliteit-dichtheidsfunctie*. Voor een bepaalde r specificeert deze formule hier voor welk percentage van de hyperbollen met straal r geldt dat $E[NN] = r$.

$$\begin{aligned} P_{E[NN]=r}(r) &= \frac{dP_{E[NN]\leq r}(r)}{dr} \\ &= \frac{D \cdot N}{r} \cdot \left(1 - \frac{\sqrt{\Pi^D} \cdot r^D}{\Gamma(\frac{D}{2}+1)}\right)^{N-1} \cdot \frac{\sqrt{\Pi^D} \cdot r^D}{\Gamma(\frac{D}{2}+1)} \end{aligned}$$

De afstand $E[NN]$ is dan het gemiddelde van alle afstanden, dus

$$\begin{aligned} E[NN] &= \int_0^\infty r \cdot P_{E[NN]=r}(r)dr \\ &= D \cdot N \cdot \int_0^\infty \left(\left(1 - \frac{\sqrt{\Pi^D} \cdot r^D}{\Gamma(\frac{D}{2}+1)}\right)^{N-1} \cdot \frac{\sqrt{\Pi^D} \cdot r^D}{\Gamma(\frac{D}{2}+1)}\right) dr \end{aligned}$$

Berekening $E[NN]$ met boundary effects Het opstellen van een formule voor de gemiddelde NN-afstand die wel rekening houdt met het clippen aan de rand van R volgt dezelfde logica als bovenstaande redenering. Het enige verschil is dat we de hyperbol Y moeten clippen aan de rand van R. Dit volume verschilt dus afhankelijk van de plaats van Y. We bereken daarom het gemiddelde volume van een hyperbol, dat we voorstellen met de volgende formule:

$$Vol_{Y(r)_{gemid}} = \iint_{(0,\ldots,0)}^{(1,\ldots,1)} \left(\iint_{(0,\ldots,0)}^{(1,\ldots,1)} 1_{d(v,w)\leq r}(v,w) \quad dv\right) dw \quad (v,w \in R)$$

In bovenstaande formule bereken we dus voor elke hyperbol $Y(r)$ met centrum w het volume/de kans. De integraal $Vol_{Y(r)_{gemid}}$ is moeilijk te berekenen. Een mogelijke oplossing hiervoor is gebruik te maken van numerieke integratie zoals de Montecarlo-methode [13]. De *Montecarlo-methode* is een numerieke integratie-methode gebaseerd op het volgende principe: het volume van een complex object komt overeen met de kans dat een willekeurig punt uit de dataruimte in het object ligt. Vanuit deze redenering kan een benadering van het volume van het object bekomen worden door willekeurig een aantal punten van de dataruimte te selecteren en te testen of deze zich in het object bevinden. We bekijken eerst hoe we deze methode kunnen toepassen per hyperbol $Y(r)$. Gegeven zijn dus een D-dimensionale hyperbol $Y(r)$ met een specifiek centrum w en w bevindt zich ergens binnen R. We willen dus het volume van $Y(r)$ weten. Het volume van Y(r) wordt voorgesteld door de integraal in v in de formule van $Vol_{Y(r)_{gemid}}$. We kunnen hiervoor de Montecarlo-methode gebruiken. Honderdduizend punten zouden hierbij voldoende nauwkeurigheid geven.
De volgende stap is om de straal van $Y(r)$ te variëren. De minimale straal van een hyperbol is nul en de maximale straal is de lengte van de diameter van R, $\sqrt{D}$. Immers, het geclipte volume van de hyperbol met $r \geq \sqrt{D}$ is altijd één. We delen het interval $]0, \sqrt{D}[$ op in aantal deelintervallen. De eindpunten van de deelintervallen kan men dan als eindpunt van de straal r nemen. Voor deze stralen bepaalt men dan het volume van de hyperbollen. Laten we het aantal verschillende stralen aanduiden met r_{aantal}. Hoe kleiner de deelintervallen, hoe hoger de nauwkeurigheid. Hoe hoger de dimensionaliteit, hoe groter de maximale lengte van r wordt.
Tenslotte kan men bovenstaande stappen herhalen voor alle D dimensies waarvoor men de integraal wil evaluëren. Voor een bepaalde dimensie D en een bepaalde straal r zoekt men dus telkens het gemiddelde volume. Men kan hiervoor het gemiddelde volume nemen van een aantal willekeurige hyperbollen die men kan bepalen met behulp van de Montecarlo-methode. We kunnen dan een tabel $Vol_{geclipt}$ opstellen die voor elk paar van dimensie en straal $Vol_{Y(r)_{gemid}}$ bevat. Voor een bepaalde dimensie en een bepaalde straal r zal een benaderend volume op plaats i staan in de reeks met stralen voor de bepaalde dimensie, waarbij i als volgt kan bepaald worden:

$$i = \left\lfloor \frac{r}{\sqrt{D}} \cdot r_{aantal} \right\rfloor$$

We hebben dus nu een formule voor het geclipte volume van een hyperbol en we kunnen deze ook in de praktijk gebruiken. Dus voor een bepaalde dimensionaliteit D geldt:

$$P_{E[NN]\leq r}(r) = 1 - (1 - Vol_{Y(r)_{gemid}})^N$$

Dan is de probabiliteit-dichtheidsfunctie:

$$P_{E[NN]=r}(r) = N \cdot (1 - Vol_{Y(r)_{gemid}})^{N-1} \cdot \frac{d\,Vol_{Y(r)_{gemid}}}{dr}$$

Vervolgens kunnen we $E[NN]$ formuleren als

$$E[NN] \tag{2}$$

$$= \int_0^{\infty} r \cdot P_{E[NN]=r}(r) \tag{3}$$

$$= \int_0^{\infty} r \cdot N \cdot \left(1 - Vol_{Y(r)_{gemid}}\right)^{N-1} \cdot \frac{d\,Vol_{Y(r)_{gemid}}}{dr} \tag{4}$$

$$= N \cdot \int_0^{\sqrt{D}} r \cdot (1 - Vol_{geclipt}[i])^{N-1} \cdot (\,Vol_{geclipt}[i] - Vol_{geclipt}[i-1]) \tag{5}$$

$$= N \cdot \frac{\sqrt{D}}{r_{aantal}} \cdot \sum_{i=1}^{r_{aantal}} i \cdot (1 - Vol_{geclipt}[i])^{N-1} \cdot (\,Vol_{geclipt}[i] - Vol_{geclipt}[i-1]) \tag{6}$$

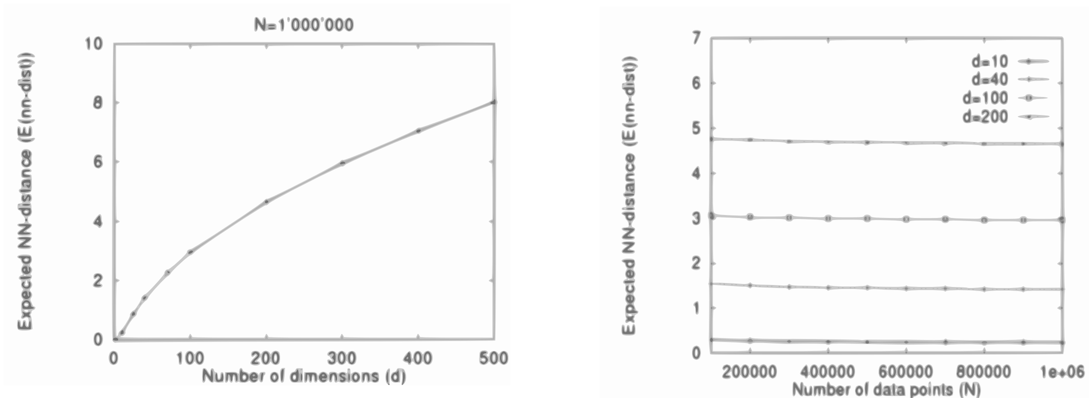
In de overgang van formule 4 naar formule 5 vervangen we ∞ door $\sqrt{D}$. Dit is toegestaan omdat het geclipte volume van een hyperbol met $r \geq \sqrt{D}$ constant is waardoor de afgeleide nul zal zijn. In diezelfde overgang vervangen we de afgeleide van $Vol_{Y(r)_{gemid}}$ door de discrete implementatie ervan. In de overgang van formule 5 naar formule 6 vervangen we de integraal door de discrete implementatie ervan, waarbij $r = i \cdot \sqrt{D}/r_{aantal}$.

In [26] gebruikt men bovenstaande theorie om $E[NN]$ te berekenen, rekening houdend met het clippen aan de rand van R. Men bekomt hierbij de grafieken in figuur 7.

Hieruit kan men de volgende besluiten trekken:

1. de NN-afstand neemt toe met een wortel-functie naarmate het aantal dimensies stijgt. Al vlug zijn de *NN*-afstanden groter dan de lengte van een ribbe van de dataruimte. De maximale afstand binnen een dataruimte van D dimensies, is de diameter van R in D dimensies. Deze diameter heeft lengte $\sqrt{D}$.

2. Het vermeerderen van het aantal punten in de dataruimte lijkt weinig invloed te hebben op de NN-afstand.



Figuur 7: Verloop van $E[NN]$ ten opzichte van D en N ([26])

In bovenstaande gevallen hebben we aangenomen dat de verschillende dimensies onafhankelijk zijn van elkaar, dat de data-distributie voldoet aan de eisen beschreven in sectie 2.2 en dat de data-distributie de query-distributie volgt.

Berekening $E[NN]$ met fractale dimensie (en boundary effects) Het kan echter zijn dat de verschillende dimensies niet onafhankelijk van elkaar zijn. We kunnen de $E[NN]$ afstand laten rekening houden met het feit dat de dimensies niet onafhankelijk van elkaar zijn. We kunnen hiervoor de powerlaw van de fractale dimensie gebruiken om de formule $P_{E[NN]\leq r}(r)$ te bekomen:

$$P_{E[NN]\leq r}(r) = 1 - \left(1 - \frac{\rho_F}{N} \cdot Y(r)^{\frac{D_F}{D}}\right)^N$$

Immers, zoals gezien in sectie 2.2, is het aantal punten binnen het volume afhankelijk van de fractale dimensie en niet van de Euclidische dimensie. De verdere uitwerking verloopt analoog als in bovenstaande gevallen.

3.3 Indexen die clusteren/partitioneren

Onder indexen die clusteren of partitioneren verstaan we indexen die gebruik maken van bounding regio's. Aan deze bounding regio's leggen we de volgende eisen op:

- Een bounding regio bevat minstens twee punten.

- Voor elke bounding regio i is er een MBR_i. De MBR kan gebruikt worden om te prunen en geeft een idee waar de punten in de bounding regio zich bevinden.

- De MBR van een bounding regio is convex.

Gebruikmakend van het feit dat de NN-afstand groeit naarmate de dimensionaliteit D stijgt, zullen we aantonen dat het gemiddeld aantal bounding regio's dat geraadpleegd wordt door indexen die aan bovenstaande voorwaarden voldoen, evolueert naar een lineaire scan.
Het aantal blokken dat geraadpleegd wordt tijdens een NN-query is naast de indexstructuur, ook afhankelijk van het zoekalgoritme. Er bestaat een algoritme dat voor een NN-query een minimaal aantal blokken raadpleegt. Minimaal wilt zeggen dat alleen die blokken geraadpleegd worden waarvan de MBR door de query-bol geïntersecteerd wordt. Het HS-algoritme [14] is het originele algoritme dat aan deze eigenschap voldoet. In sectie 5.1 geven we een incrementeel algoritme dat eveneens minimaal is. Omdat er een algoritme bestaat dat minimaal is, kunnen we dus een NN-query omzetten naar een range-query met als straal de NN-afstand.
Het aantal blokken dat opgevraagd wordt, is ook afhankelijk van de gebruikte indexstructuur. Eerst construeren we een formule die ons toelaat uit te drukken hoeveel kans er is dat een blok geraadpleegd wordt bij het uitvoeren van NN-queries, onafhankelijk van de gebruikte index. Laten we de query-bol met straal $E[NN]$ noteren als $Y(E[NN])$. De kans dat MBR_i geïntersecteerd wordt door een NN-query is dan $Vol_{MBR_i \oplus Y(E[NN])}$. Hierbij houden we dus geen rekening met het clippen aan de rand van R. Houden we wel rekening met het clippen aan de rand van R, dan wordt de formule:

$$P[MBR_i] = Vol_{((MBR_i \oplus Y(E[NN])) \cap R)}$$

We willen dit nu wat abstracter maken, zodat we meer inzicht krijgen hoe de kans dat een MBR geraadpleegd wordt, evolueert zonder de specifieke details van de MBR's van de regio's te kennen. We weten wel dat elke MBR minstens twee punten bevat. Laten we één punt hiervan a_i noemen en een andere b_i. Aangezien de MBR's convex zijn, weten we dat het lijnstuk $[a_i, b_i]$ zich binnen MBR_i zal bevinden. We vervangen dan de Minkowski-Som $Vol_{MBR_i \oplus Y(E[NN])}$ door de Minkowksi-Som van het lijnstuk $[a, b]$ en $Y(E[NN])$. We verlagen hierdoor eventueel de kans dat MBR_i geïntersecteerd wordt aangezien $Vol_{(([a_i,b_i] \oplus Y(E[NN])) \cap R)} \leq Vol_{((MBR_i \oplus Y(E[NN])) \cap R)}$. We kunnen voor bounding regio i nog een lagere kans bekomen door het lijnstuk te kiezen zodanig dat de Minkowski-Som minimaal is. De Minkowski-Som hangt af van de lengte van het lijnstuk en de positie ervan. Immers, we houden hier

rekening met het clippen van de Minkowski-Som aan de rand van R waardoor het volume van de Minkowski-Som kan verminderd worden. We nemen als lengte van het lijnstuk de gemiddelde afstand tussen twee punten $E[NN]$. Eén punt van het lijnstuk is a_i. Het andere punt c_i nemen we zodanig dat de Minkowski-Som geminimaliseerd wordt. Dus,

$$Vol_{([a_i,b_i]\oplus Y(E[NN]))\cap R} \geq Vol_{([a_i,c_i]\oplus Y(E[NN]))\cap R}$$
$$= min_{p_i \in oppervlak(Y(a_i,E[NN]))} Vol_{(([a_i,p_i]\oplus Y(E[NN]))\cap R}$$

Een ondergrens voor de kans dat een bounding regio geraadpleegd wordt, is dan

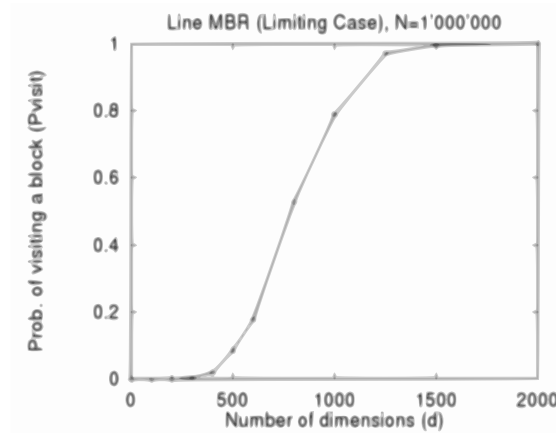
$$P[MBR_i] \geq Vol_{([a_i,c_i]\oplus Y(E[NN]))\cap R}$$

Omdat we het hier algemeen willen houden, weten we niet hoeveel bounding regio's er nu precies zijn. De variabele is hier het aantal dimensies. We zijn hier dus geïnteresseerd hoe het volume van de Minkowski-Som varieert ten opzichte van het aantal dimensies. Een ondergrens voor de gemiddelde kans dat een bounding regio geïntersecteerd wordt door een query-bol $P_{geraadpleegd}$ is dan

$$P[MBR] \geq \int_{a\in R} Vol_{([a,c]\oplus Y(E[NN]))\cap R} \tag{7}$$

$$(\,Vol_{([a,c]\oplus Y(E[NN]))\cap R} = min_{p\in oppervlak(Y(a,E[NN]))} Vol_{(([a,p]\oplus Y(E[NN]))\cap R)} \tag{8}$$

De integraal in formule 7 werd geplot met behulp van de Montecarlo-methode, waardoor men de grafiek in figuur 8 verkreeg.



Figuur 8: Plot van integraal in formule 7 ([26])

Hieruit kan men volgende feiten afleiden:

- Vertrekkende vanuit praktische ervaring dat een indexmethode maximaal 20% van de data mag bekijken om efficiënt te zijn, kunnen we besluiten dat er voor elke methode die clustert/partitioneert, er een dimensionaliteit is waarbij de lineaire scan beter presteert. Uit de grafiek blijkt deze grens rond 610 dimensies te liggen. Aangezien bovenstaande theorie een optimistische benadering is, moeten we verwachten dat het aantal dimensies lager ligt.

- Het aantal blokken dat geraadpleegd moet worden, stijgt naarmate de dimensionaliteit stijgt.

- Voor elke methode die clustert of partitioneert, bestaat er een dimensionaliteit vanaf waar alle blokken geraadpleegd moeten worden.

3.4 Samenvatting

We hebben een overzicht gegeven van de theorie die belangrijk is bij het optimaliseren van queries in verschillende opzichten, waaronder de indexen die men gebruikt en de opslag van de index. We hebben een verklaring gegeven voor de curse of dimensionality-feiten:

- degradatie van de performantie van indexen die clusteren/partitioneren naarmate D stijgt

- boundary-effects

Een belangrijk feit is dat men een onderscheid dient te maken tussen laag/ medium-dimensionale en hoog-dimensionale ruimten omwille van boundary effects. We hebben ook de verwachte $E[NN]$-afstand bekeken en merkten op dat deze groeit naarmate D stijgt. De formules voor de $E[NN]$-afstand zullen terugkomen in sectie 5.3 waarin we een kostmodel voor NN-queries bespreken. Tenslotte moeten we ook opmerken dat we de fractale dimensie in plaats van de Euclidische dimensie gebruiken in formules. De performantie van een index kan dus beter zijn dan een kostmodel, dat gebruikt maakt van de Euclidische dimensie, zou laten blijken aangezien de fractale dimensie lager dan de Euclidische kan zijn.

4 Indexen

We hebben in het vorig hoofdstuk een aantal theoretische aspecten besproken die gelden in multidimensionale ruimten. Nu zullen we de invloed hiervan op een aantal database-onderdelen bespreken.

Alhoewel we gezien hebben dat partitionerende indexen evolueren naar een lineaire scan, zullen we toch een aantal indexen bespreken. We kunnen proberen de index tot op een zo hoog mogelijke dimensie zo goed mogelijk te laten werken. We hebben afgesproken dat 20% van de datablokken maximaal geraadpleegd mogen worden om sneller te kunnen zijn dan een lineaire scan. Wegens de hoge dimensionaliteit en het feit dat afstandsberekeningen veel gebruikt kunnen worden, kan ook de CPU-kost een belangrijke factor spelen in de totale tijd om een query af te handelen. Naast de indexen die hier besproken zullen worden, zijn er nog andere indexen geschikt voor hoogdimensionale ruimten: hybrid-tree, cir-tree, ... Laag- en medium-dimensionale indexen worden hier niet besproken, maar onder deze vallen: R-tree, R*-tree [23], ...

De indexen moeten ons dus helpen om efficiënt queries te kunnen stellen over punten in een multidimensionale ruimte. We concentreren ons hierbij vooral op nearest-neighbour-queries (NN-queries). We maken onder andere een onderscheid tussen *Euclidische* en *metrische* indexen. Euclidische indexen worden opgebouwd met behulp van de waarden in de feature-vectoren. Metrische indexen worden geconstrueerd door de waarden van een afstandsfunctie $d(x, y)$, waarbij x en y objecten, feature vectoren, ... zijn, die voldoet aan minstens drie eigenschappen:

- symmetrie: $d(x, y) = d(y, x)$
- niet-negatief: $d(x, y) > 0 \Leftrightarrow x \neq y$ en $d(x, x) = 0$
- driehoeksongelijkheid: $d(x, y) \leq d(x, z) + d(z, y)$

Een metrische index kan voor niet niet-Euclidische ruimten gebruikt worden. Voorbeelden van metrische indexen zijn de M-tree [25], de MVP-tree [7], de M+-tree [40], ... We zullen hier echter enkel Euclidische indexen bespreken.

Vooraleer we bepaalde indexen bespreken, introduceren we enkele begrippen. Een *blok* is een eenheid voor een hoeveelheid opeenvolgende bytes. Een veel voorkomend begrip is *fan-out*. De fan-out is het percentage feature-vectoren dat in één blok van een bepaalde grootte kan. Hoe hoger de fan-out, hoe meer feature-vectoren in het blok kunnen, hoe minder overhead (navigatie-informatie, ...) er dus is. Hoe hoger de dimensionaliteit, hoe groter de vectoren zijn, wat afhankelijk van de methode een lagere fan-out zou kunnen betekenen.

4.1 SS-tree

De SS-tree [38] is een dynamische Euclidische indexstructuur om hoog-dimensionale ruimten te indexeren. Bij dynamische indexen hoeft men

de index niet helemaal opnieuw te bouwen telkens iets aan de index wordt toegevoegd. Dit is wel het geval bij statische indexen. Men stelt dat de query-performantie hetzelfde moet zijn als die van statische indexstructuren, maar updates mogen wel langere tijd in beslag nemen.

4.1.1 Structuur

M is het maximaal aantal kinderen dat een knoop kan bevatten. Voor een bepaalde SS-tree bepalen we ook m, waarbij $0 \leq m \leq M$, die het minimaal aantal kinderen voorstelt dat een knoop moet bevatten. Deze definities van m en M vinden we ook terug bij andere bomen zoals de R*-tree [23]. Men gebruikt sferische bounding regio's om de feature-vectoren in de multidimensionale ruimte te groeperen. In de originele paper gebruikt men slechts 1 soort knoop in de implementatie. Hierin slaat men de informatie op voor zowel de navigatieknoop als de dataknoop. *Navigatieknopen* gebruikt men om door de boomstructuur te navigeren. De *rootknoop* is de navigatieknoop die geen ouder heeft. *Dataknopen* bevatten de feature-vector(en). In de implementatie van de oorspronkelijke paper is de navigatieknoop de enige geïmplementeerde knoop. Voor de dataknoop gebruikt men dus ook de navigatieknoop, maar men gebruikt er slechts enkele velden van.

De navigatieknoop

- `childArray`: array van pointers naar kinderen
- `nrOfChildren`: aantal directe kinderen (aantal ingevulde pointers in `childArray`)
- `nrOfSubtreeChildren`: aantal kinderen/knopen in de subboom met als wortel deze navigatieknoop
- `height`: de hoogte tussen het niveau van deze knoop en het bladniveau
- `centroid`: het gemiddelde van de vectoren (centroid's of feature-vectoren) van de kinderen
- `radius`: de afstand van de `centroid` tot de feature-vector van het kind dat het verst afgelegen is van de `centroid` of een grotere afstand. Ook al is de afstand groter dan de afstand tot het verste punt, we zullen de `radius` nog kunnen gebruiken om de boom te prunen tijdens de query.
- `closestChild`: de info van het directe kind dat het dichtst bij de `centroid` is gelegen.

- `nrOfUpdates`: hoeveel keer deze knoop geüpdatet is.

Het veld `nrOfUpdates` wordt gebruikt om lazy recalculation te doen. Hiermee wordt bedoeld dat de `radius` en van een knoop niet bij elke update worden herberekend, maar telkens pas na een aantal updates. Dit vermindert uiteraard de CPU-kost.

De dataknoop

- `featureVector`: dit is de `centroid`-variabele in de navigatieknoop
- `data`: dit is de `closestChild`-variabele in de navigatieknoop
- `radius`: uitbreiding van het object in de multidimensionale ruimte. deze is nul in de implementatie van de oorspronkelijke paper.

4.1.2 Operaties

Invoegen Men gebruikt bij het invoegen de overflow-treatment-techniek *forced reinsert* die ook gebruikt werd bij de R*-tree [23]. Hierbij gaat men een vector niet onmiddellijk splitsen als deze vol is, maar men gaat $x\%$ van de elementen van de vector opnieuw invoegen in de boom. Uit de experimenten van de R*-tree-paper blijkt dat $x = 30$ goede resultaten levert. Forced reinsert heeft twee voordelen ten opzichte van het opsplitsen:

- Men probeert de overhead van het recursief opsplitsen te vermijden.
- De boomstructuur wordt geöptimaliseerd. Immers, de structuur van deze boom hangt af van de volgorde waarin elementen toegevoegd worden, waardoor vroegere beslissingen een negatief effect kunnen hebben op de structuur. Door het herinvoegen wordt dit probleem wat verminderd.

In algoritme 4.1 beschrijven we het forced reinsert-algoritme.
Men kan onder andere twee manieren onderscheiden waarop men de knopen in de reinsert-lijst terug kan invoegen. Men maakt hierbij gebruik van de afstanden berekend in stap 1 van het forced reinsert-algoritme.

- far reinsert: van de verste knoop tot de dichtsbijzijnde
- close reinsert: van de dichtsbijzijnde tot de verste knoop

algoritme 4.1 forced reinsert

Input: Zij A een knoop die we willen toevoegen aan knoop B. Knoop B heeft echter het maximaal aantal kinderen bereikt.

Output: knoop A is toegevoegd aan de boom

1: Berekenen de afstand van alle M+1 knopen ten opzichte van de `centroid` van B.
2: Rangschik de M+1 knopen volgens dalende afstand
3: Verwijder de eerste x procent van de knopen (deze hebben immers het meeste kan om ergens anders toegevoegd te worden). Pas `nrOfChildren` van de directe parent aan. Verander `nrOfUpdates`, `nrOfSubtreeChildren`, `centroid` en `radius` (zie verder hoe deze berekend wordt als er nog niet genoeg updates zijn) van alle ouders tot en met de rootknoop aan. Pas als een knoop uit alle parents is verwijderd, wordt deze aan de reinsert-lijst toegevoegd.
4: Voeg de knopen in de reinsert-lijst terug toe aan de boom.

Uit experimenten blijkt dat close reinsert beter is dan far reinsert. Het reinsertion-algoritme kan enkel aangeroepen worden op een knoop P als het hiervoor geen enkele keer op P is toegepast geweest of als de laatste overflow treatment van P een splitsing was. Men kan hiervoor een lijst van reinserted knopen bijhouden.

De `centroid` is het gemiddelde van de vectoren van de kinderen. De D-dimensionale `centroid` c van een knoop F met als kinderen k_j ($j \in \{1, \ldots, n\}$), die telkens een feature-vector of centroid vect hebben, wordt als volgt bepaald:

$$\forall i \in \{1, \ldots, D\} : \quad c_i = \frac{\sum_{j=1}^{n} k_j.vect_i \; \cdot \; k_j.\texttt{nrOfSubtreeChildren}}{\sum_{j=1}^{n} k_j.\texttt{nrOfSubtreeChildren}}$$

De updates van `nrOfChildren` en `centroid` in de ouders zijn simpele aanpassingen aan de waarden. Eerst berekent men het totaal aantal kinderen die verdwijnen en de verandering aan de `centroid` van de rechtstreekse parent. Vervolgens is `nrOfChildren` telkens een een aftrekking en is er een herberekening van de `centroid`.
De update van de `radius` is geen exacte berekening. We hebben vermeld dat de lengte van de straal minstens de afstand tot het verste punt is. De `radius` wordt in dit geval berekend als de som van de lengte van de oude `radius` en de afstand van de oude `centroid` tot de nieuwe `centroid`. Deze afstand is

minstens zo groot als de exacte afstand. Op deze manier hoeven we de dus niet de verst afgelegen knoop te bepalen, de vector ervan op te vragen en de afstand tot de nieuwe `centroid` te berekenen. Dit zal dus wel uiteindelijk gedaan worden als de knoop een aantal updates heeft ondergaan.

We beschrijven vervolgens het *invoeg-algoritme* 4.2 waarbij we knoop A toevoegen aan de boom.

algoritme 4.2 SS-tree: invoegen

Input: Zij A een knoop die we toevoegen aan de boom.
Output: knoop A is toegevoegd aan de boom

1: Voeg A toe aan de reinsert-lijst
2: **while not** reinsertlijst.leeg() **do**
3:    **if** boom.leeg() **then**
4:       maak een nieuwe navigatieknoop/rootknoop aan en voeg A eraan toe. Verwijder A uit de reinsert lijst
5:    **else**
6:       Daal de boom af totdat de ouder van A bereikt is. Van elke knoop die men onderweg tegenkomt (ook de ouder van A), moet `nrOfUpdates`, `nrOfSubtreeChildren`, `centroid` en `radius` aangepast worden. Men kiest telkens de knoop/subboom waarvan de `centroid` het dichtst bij de vector van A ligt[2].
7:       **if not** ouder.Vol() **then**
8:          Voeg de knoop toe en pas `nrOfChildren` aan
9:       **else**
10:          Voer het forced-reinsert-algoritme (4.1) uit indien dit toegelaten is. Indien forced reinsert niet toegelaten is, voer het splitsingsalgoritme (4.3) uit.
11:       **end if**
12:    **end if**
13: **end while**

4.2 SR-tree

4.2.1 Motivatie

De SR-tree [8] is een verbetering van de SS-tree. Een probleem bij de SS-tree is dat hij gebruikt maakt van bounding hyperbollen. In hoog-dimensionale ruimten nemen bounding hyperbollen meer ruimte in dan bounding hyperbalken, waardoor er dus meer kans is op overlappende gebieden. Dit heeft

algoritme 4.3 SS-tree: splitsen

Input: een navigatieknoop

Output: de kinderen/knopen van de navigatieknoop zijn verdeeld over twee nieuwe navigatieknopen en de oude navigatieknoop wordt verwijderd

1: We bepalen de dimensie met de grootste `variance`(variantie) waarbij we gebruik maken van de `centroid`'s van de kinderen. We gebruiken hiervoor de waarden van de punten voor de respectievelijke dimensie(s). We splitsen de dimensie met de grootste variantie in twee delen op. De splitslocatie in die dimensie wordt zo gekozen dat de som van de varianties van beide delen minimaal is. Voor beide kanten berekenen we dan de variantie van de punten die aan die respectievelijke kant zich bevinden. We proberen te zoeken naar het minimum voor de som van beide varianties. Het aantal mogelijke splitsingslocaties hangt af van M en m. Bijvoorbeeld, als $M = 2m$, dan zijn er maar twee mogelijkheden aangezien elk deel minstens m knopen moet hebben.

2: Twee nieuwe navigatieknopen worden aangemaakt en de knopen worden verdeeld over de navigatieknopen volgens de berekende splitsing

3: **if** rootknoop wordt gesplitst **then**

4: Er wordt een nieuwe rootknoop aangemaakt en de twee nieuwe knopen voegen we toe aan de nieuwe rootknoop.

5: **else**

6: De nieuwe knoop die het dichtst bij de parent van de oorspronkelijke knoop ligt, vervangt de vroegere knoop. De andere nieuwe knoop wordt aan de reinsert-lijst toegevoegd. Dit laatste valt niet onder het forced reinsert algoritme.

7: **end if**

algoritme 4.4 SS-tree: verwijderen

Input: knoop A met kinderen, m is het minimum aantal kinderen dat een knoop moet bevatten.

Output: een kind/knoop van A is verwijderd uit de boom

1: **if** A.`nrOfChildren` $> m$ **then**

2: Pas de betrokken waarden van A aan. De waarden van de parents van A worden al geüpdatet tijdens het zoeken naar de knoop die verwijderd dient te worden.

3: **else**

4: Deze situatie wordt underflow genoemd. Verwijder knoop A uit de boom, zijn kinderen worden aan de reinsert-lijst toegevoegd en de betrokken waarden van de parents worden geüpdatet.

5: **end if**

dan weer als gevolg dat er meer kans is dat het zoeken trager gaat verlopen, aangezien er meer knopen onderzocht moeten worden.
De SR-tree vertoont veel gelijkenis met de SS-tree en we zullen hier enkel de verschillen bespreken.

4.2.2 Structuur

De SR-tree maakt gebruik van meerdere soorten bounding regio's per knoop om het gebied ervan zo klein mogelijk te houden. Dit zorgt er voor dat de verschillende gebieden meer disjunct worden. De SR-tree maakt per knoop gebruik van een bounding hyperbalk en een bounding hyperbol. Het gebied van de knoop is dan de intersectie van de bounding hyperbalk en de bounding hyperbol. De bounding hyperbalk zou dus het gebied van de bounding hyperbol meer moeten beperken. Waarom eigenlijk een bounding hyperbol gebruiken? Bijvoorbeeld, bij NN-queries is het niet het volume wat vooral belangrijk is. De diameter van het gebied is hier belangrijker. Bij een bounding hyperbalk definiëren we de diameter als de diagonaal. Alhoewel een bounding hyperbalk in hoog-dimensionale ruimten minder volume inneemt, is zijn diameter groot. Daarentegen, verdelen bounding hyperbollen de ruimte in gebieden met een kleinere diameter, maar een groter volume.

De navigatieknoop

- alle elementen van de SS-tree
- bounding hyperbalk

De dataknoop

- alle elementen van de SS-tree

4.2.3 Operaties

Invoegen Het invoeg-algoritme van de SR-tree is hetzelfde als dat van de SS-tree, behalve dat als we een bounding hyperbol aanpassen/invoegen, we ook de bounding hyperbalk aanpassen/invoegen. De bounding hyperbol wordt hier op een andere manier aangepast dan bij de SS-tree. We gebruiken hier een andere methode omdat we ook rekening willen houden met de bounding hyperbalk van de kinderen om zo een kleiner gebied te bekomen.

De grenzen van de bounding hyperbalk zijn de kleinste en grootste waarde van elke dimensie van de punten binnen de bounding hyperbalk. Het centrum van de hyperbol wordt op dezelfde manier bepaald als bij de SS-tree.

De `radius` is het minimum van twee afstanden:

- De eerste afstand is de lengte van de `radius` zoals deze bij de SS-tree berekend wordt.
- De tweede afstand is de maximale afstand van het centrum van de navigatieknoop tot de bounding hyperbalk van de directe kinderen van de knoop. De afstand tot een bounding hyperbalk is hier de maximale afstand tot een punt van de bounding hyperbalk.

4.2.4 Performantie

Uit experimenten blijkt dat de SR-tree een betere performantie heeft dan de SS-tree. De volumes van de gebieden zijn gemiddeld kleiner dan bij de SS-tree en R*-tree [23]. De diameters zijn gemiddeld van ongeveer dezelfde lengte als die van de SS-tree. Echter, door het toevoegen van een bounding-hyperbalk kan men verwachten dat de fan-out gaat verslechteren. Er kunnen dus minder navigatieknopen en bladknopen in één blok gestoken worden, waardoor er een hogere kans is dat er meer blokken moeten gelezen worden dan bij een methode met een lagere fan-out. Dit heeft dan een negatieve invloed op de query-performance. Zo is de fan-out van de SR-tree 1/3 van die van de SS-tree en 2/3 van die van de R*-tree [23]. Uit de experimenten blijkt echter dat de SR-tree minder van schijf leest dan de SS-tree. De SR-tree moet meer navigatie-informatie lezen dan de SS-tree, maar de SR-tree moet echter minder data-informatie lezen en de reductie in data-informatie is groter dan de stijging van navigatie-informatie ten opzichte van de SS-tree. Uit de experimenten blijkt dat hoe minder uniform de datadistributie is, hoe beter de SR-tree is ten opzichte van de SS-tree. Men merkt een performantiestijging van rond de 40 procent, maar dit hangt dus af van de datadistributie. Bij uniforme data presteren beide indexen ongeveer even goed.

4.3 VA-file (vector approximation-file)

4.3.1 Motivatie

We hebben vermeld dat uit praktische ervaring blijkt dat als een index meer dan 20% van de data opvraagt, een lineaire scan performanter is. Aangezien de NN(/range)-query-performantie van indexen die opdelingen van de data/ruimte maken altijd evolueren tot een lineaire scan (alle bounding regio's/feature-vectoren worden opgevraagd) naarmate de (fractale) dimensionaliteit stijgt, werd de VA-file [26] geïntroduceerd[3]. De VA-file is een kleine

[3]Een andere, nieuwere index die ook als doel heeft de lineaire scan te versnellen is de IQ-tree [32].

index/datastructuur waar we vlug doorheen kunnen scannen en is bedoeld om de lineaire scan te versnellen. Dit gebeurt door de ruimte te verdelen in regio's en van de info van die regio's approximaties te maken.

4.3.2 Structuur

De dataruimte wordt opgesplitst in 2^b hyperbalkvormige delen. We kunnen elk deel uniek adresseren met behulp van een bitstring van b bits. In de VA-file houden we voor elke vector bij in welk deel deze zich bevindt. Aan elke dimensie kennen we b_i bits toe, zodanig dat als we D dimensies hebben, $\sum_{i=1}^{D} b_i = b$. De waarde van b_i bevindt zich normaal tussen 3 en 7. Een dimensie D_i heeft dus 2^{b_i} delen. Er geldt dus ook dat $\prod_{i=1}^{D} 2^{b_i} = 2^b$. Per dimensie i worden de grenzen van de delen van i zo bepaald dat elk deel van i evenveel vectoren bevat.
De VA-file bevat voor elke feature-vector, het id van de feature-vector en een bitstring van lengte b die de positie van de feature-vector beperkt tot een hyperbalkvormige bounding region. Er moet ook voor elke dimensie opgeslagen worden waar de dimensie gesplitst is en begrensd wordt.

4.3.3 Operaties

Range/NN-queries worden uitgevoerd door heel de VA-file te scannen.

4.3.4 Performantie

De VA-file heeft een hoge fan-out. Afhankelijk van de precisie van de vectoren en de grote van de bitstring b, is er een overhead op het gebied van opslaghoeveelheid van 12.5%–25% ten opzichte van de werkelijke info.
In tegenstelling tot indexen die opdelingen maken of clusteren, maakt de VA-file gebruik van het feit dat punten verspreid liggen in hoog-dimensionale ruimte, met andere woorden dat er weinig kans is dat twee punten in eenzelfde deel liggen. Hoe komt dit? Doordat er maar een zeer kleine kans is dat twee punten in hetzelfde deel liggen, wordt een bitstring bijna een identificatiemiddel voor een vector. Hierbij komt dan nog het bovenstaande feit dat de VA-file maar 12.5%-25% van de werkelijke data is in omvang. Omwille van deze reden kan men dus spreken van het comprimeren van vector-data. Men moet dus zo'n 12.5%–25% van de data te bekijken om een vector misschien uniek te identificeren.

Wat opvalt bij de experimenten is dat de verhouding van het aantal vectoren of het aantal blokken ten opzichte van de dimensionaliteit dat bezocht wordt tijdens een k-NN-search lijkt op de omgekeerde logaritmische functie

die naar nul convergeert als het argument naar oneindig gaat. Natuurlijk is er nog de kost van het lezen van het hele index-bestand, maar deze is kleiner dan de werkelijke hoeveelheid data en de VA-file lijkt volgens experimenten bruikbaar en beter dan andere methoden zoals de X-tree, R*-tree [23] en de lineaire scan vanaf een dimensionaliteit van zes.

Als punten meer geclusterd liggen, dan kan de VA-file niet meer zo goed gebruiken maken van het feit dat punten in hoog-dimensionale ruimten zeer verspreid liggen. Veronderstel dat we twee punten hebben die gelegen zijn in een dataruimte waarin de punten uniform verdeeld zijn. Als deze twee punten beide voor een bepaalde dimensie zich in hetzelfde deel bevinden, dan is de kans niet zo groot dat dit ook zo is voor een andere dimensie. Bij geclusterde dataruimten, is deze kans veel hoger uiteraard (de opdelingen van de dimensies worden onafhankelijk van elkaar behandeld), waardoor er meer kans is dat twee punten in hetzelfde deel liggen en dus de benadering niet zo precies is, zodat veel meer vectoren opgezocht moeten worden. Een andere mogelijkheid is de bitstring te vergroten, maar dan wordt ook de VA-file groter, wat een negatief effect heeft op de query-performantie.
De VA-file is dus een tamelijk eenvoudige methode die veelbelovend lijkt voor het zoeken naar punten in een hoog-dimensionale ruimte.

4.4 A-tree (approximation tree)

4.4.1 Motivatie

De A-tree [41] is gebaseerd op de SR-tree en de VA-file.
Bij de SR-tree was de fan-out een negatief punt, maar voor de rest lijkt dit een goede structuur. Uit testen blijkt dat bij het zoeken in de SR-tree in hoog-dimensionale ruimten de bounding hyperbol niet zo veel gebruikt wordt om te prunen dan de bounding hyperbalk.

De VA-file gebruikte het principe van de approximatie/compressie met behulp van de bitcodering, wat een goede fan-out oplevert en goed werkt voor uniform verdeelde datapunten, maar wat minder voor geclusterde dataruimten. De meeste dataruimten in de praktijk zijn echter geclusterd, waardoor de VA-file dan waarschijnlijk iets minder presteert.

De A-tree neemt grotendeels de SR-tree over omdat de meeste niet-synthetische data niet uniform verdeeld is. Enkele veranderingen worden doorgevoerd om zo een betere index te bekomen. De A-tree maakt gebruik van VBR's (virtual bounding rectangles) om de fan-out te verbeteren. De VBR's zijn relatieve benaderingen van de bounding hyperbalken van knopen ten opzichte van de bounding hyperbalk van de ouder. Men spreekt hier

van relatieve approximatie. Bij de VA-file kon men waarnemen dat als de datadistributie niet-uniform was, de performantie erop achteruit ging. Dit was omdat de approximatie slechter werd. Door relatieve approximatie te gebruiken, verandert de approximatie afhankelijk van de data-distributie. Men probeert bij de SR-tree immers clusters te vormen die zo weinig mogelijk overlappen en de clusters te vormen volgens de data-distributie. Op deze manier wordt de approximatie-fout kleiner dan bij de VA-file. De VBR's nemen minder ruimte in beslag om op te slaan. Elke navigatieknoop bevat naast zijn eigen bounding hyperbalk, de VBR's van zijn kinderen. We kunnen de VBR's dan gebruiken om de boom vlugger te prunen. De performantie die verloren gaat omwille van de approximatie-fout, blijkt kleiner te zijn dan de voordelen ervan. Omdat de bounding hyperbol niet zoveel gebruikt wordt tijdens het zoeken, wordt de `radius` van de hyperbol verwijdert uit de navigatieknoop. De `centroid` blijft echter behouden voor het invoegen verwijder-proces.

4.4.2 Structuur

Constructie van een VBR Een bounding hyperbalk kan altijd voorgesteld worden door de twee eindpunten van een van zijn diameters. Een VBR is relatief ten opzichte van de bounding hyperbalk van de ouderknoop. Zij $a(a_1, \ldots, a_D)$ en $a'(a'_1, \ldots, a'_D)$ de twee eindpunten van de diameter van de bounding hyperbalk A van de ouderknoop. We kiezen de eindpunten van de diameter waarvoor geldt dat $a_i \leq a'_i$ voor $i \in \{1, \ldots, D\}$. Zij B de bounding hyperbalk van één van de kinderen. De twee eindpunten van de diameter van B noemen we b en b' en er geldt dat $b_i \leq b'_i$ voor $i \in \{1, \ldots, D\}$. We construeren nu een VBR voor B ten opzichte van A.

Voor elke dimensie delen we het lijnstuk $[a_i, a'_i]$ op in z gelijke delen, $z \in \mathbb{N}$ en $z \geq 1$. De waarden van het punt met de kleinste waarden van de VBR wordt berekend door de functie $Q_s(b_i)$ (startpunt) en de waarden van het punt met de grootste waarden door de functie $Q_e(b'_i)$ (eindpunt).
De bedoeling is b_i te benaderen door $Q_s(b_i)$, zodanig dat Q_s een compactere waarde oplevert en $Q_s(b_i) \leq b_i$.

$$Q_s(b_i) = a_i + \frac{(a'_i - a_i) \cdot h_s(b_i)}{z}$$

en

$$h_s(b_i) = \begin{cases} z - 1 & (b_i = a'_i) \\ \left\lfloor \frac{b_i - a_i}{a'_i - a_i} \cdot z \right\rfloor & \text{(in de andere gevallen)} \end{cases}$$

Er geldt dat $0 \leq Q_s(b_i) \leq z-1$. Immers, als b_i tussen $z-1$ en z ligt, wordt $h_s(b_i)$ naar $z-1$ afgerond en als $b_i = a'_i$, dan is $h_s(b_i) = z-1$, wat ook het maximum is van de tweede formule. We kunnen dus het getal $h_s(b_i)$ gebruiken om de waarde b_i te benaderen.

Voor de eindpunten, gebruiken we een andere formule. De bedoeling is b'_i te benaderen door $Q_e(b'_i)$, zodanig dat Q_e een compactere waarde oplevert en $b'_i \leq Q_e(b'_i)$.

$$Q_e(b'_i) = a_i + \frac{(a'_i - a_i) \cdot h_e(b'_i)}{z}$$

en

$$h_e(b'_i) = \begin{cases} 1 & (b'_i = a_i) \\ \left\lceil \frac{b'_i - a_i}{a'_i - a_i} \cdot z \right\rceil & \text{(in de andere gevallen)} \end{cases}$$

Er geldt dat $1 \leq Q_e(b'_i) \leq z$. We kunnen het getal $h_e(b'_i)$ gebruiken om b'_i te benaderen.
Er geldt dus dat $a_i \leq Q_s(b_i) \leq b_i \leq b'_i \leq Q_e(b'_i) \leq a'_i$ en de VBR van B wordt dan gedefinieerd als

$$\begin{aligned} VBR(\mathrm{B}) &= (v, v'), \text{ waarbij} \\ v &= (Q_s(b_1), \ldots, Q_s(b_D)) \text{ en} \\ v' &= (Q_e(b_1), \ldots, Q_e(b_D)) \end{aligned}$$

Er geldt dat $h_s(b_i) \in \{0, \cdots, z-1\}$ en $h_e(b'_i) \in \{1, \cdots, z\}$. Voor beide getallen zijn er dus maximaal z verschillende waarden. Met de z verschillende waarden kunnen we elke mogelijke waarde van de twee getallen identificëren. Om z verschillende waarden op te slaan hebben we $\lceil \log_2 z \rceil$ bits nodig. We hebben dus een compactere representatie van b_i en b'_i. Voor een VBR in D dimensies hebben we dus $2 * D * \lceil \log_2 z \rceil$ bits nodig. Deze bitcode voor een VBR(B) noemen we de *subspace code* van de VBR(B) ten opzichte van de bounding hyperbalk A van de ouder met radix z.
Naast bounding hyperbalken, kunnen we een VBR ook gebruiken om een datapunt te benaderen. Voor de VBR hebben we dan maar één vector nodig in plaats van twee. Bijvoorbeeld, de vector gedefiniëerd door $h_s(b_i)$. De tweede vector kan dan berekend worden uit de eerste vector, namelijk $Q_e(b'_i) = Q_s(b_i) + 1$.

Structuur van de index Omdat er toch tamelijk wat verschillen zijn tussen de SR-tree en de A-tree, beschrijven we de structuur volledig in plaats van enkel de verschillen te vermelden. Bij de A-tree wordt de navigatieknoop onderverdeeld in 3 types: de rootknoop, de intermediateknoop en de bladknoop.

De rootknoop

- een lijst van elementen. Een element bestaat hier uit een pointer naar een kind, de VBR van het kind ten opzichte van de hele dataruimte, het aantal feature-vectoren in de subboom waarvan het kind de wortel is en de `centroid` van de feature-vectoren in de subboom waarvan het kind de wortel is.

Een intermediateknoop zijn de knopen boven de bladknopen op de rootknoop na.
De intermediateknoop

- een bounding hyperbalk die de bounding hyperbalken van de kinderen omvat
- een lijst van elementen. Een element bestaat uit een pointer naar een kind, de VBR van de bouding hyperbalk van het kind ten opzichte van de bounding hyperbalk van de intermediateknoop, het aantal feature-vectoren in de subboom waarvan het kind de wortel is en de `centroid` van de feature-vectoren in de subboom waarvan het kind de wortel is.

Er is een bijectie tussen de dataknopen en de bladknopen. Bij elke dataknoop hoort juist één bladknoop en omgekeerd.
De bladknoop

- bounding hyperbalk H die de feature-vectoren van de bijhorende dataknoop omvat
- de pointer naar de bijhorende dataknoop
- Voor elke feature-vector van de bijhorende dataknoop een VBR ten opzichte van H

De dataknoop

- minimaal m en maximaal M feature-vectoren en telkens de bijhorende referentie naar de data die het beschrijft.

Bij de root- en intermediateknopen zit alles in eenzelfde blok, behalve eventueel de `centroid` en het aantal data-objecten in de subboom omdat het zoek-algoritme alleen de bounding hyperbalk, de pointer en de VBR gebruikt. Dit zorgt voor een hogere fan-out en snellere toegang tot de data die we nodig hebben tijdens het zoeken.

4.4.3 Operaties

Het invoegen en verwijderen van een knoop of feature-vector A gebeurt ongeveer hetzelfde als bij de SR-tree. Er zijn echter wat veranderingen. De `radius` hoeft dus niet meer geüpdatet te worden. Omwille van de VBR's zijn er nog enkele veranderingen. We bespreken hier enkel de verschillen:

Zij A een feature-vector of knoop die toegevoegd, verwijderd of waarvan de vector (feature-vector of `centroid`) veranderd wordt. Zij B de knoop waarvan A een kind of feature-vector is. De bounding rectangle van B moet dan eventueel aangepast worden. Er zijn dan twee mogelijkheden:

- Als de bounding rectangle van B niet veranderd hoeft te worden, dan moet enkel de VBR van A aangepast, toegevoegd of verwijderd te worden.

- Als de bounding rectangle van B veranderd, dan moeten minstens de VBR's van de directe kinderen van B aangepast worden en de VBR van B zelf. Er moet vervolgens gecontroleerd worden als de bounding rectangle van de ouder van B aangepast moet worden. Deze procedure moet op deze manier recursief toegepast worden.

4.4.4 Performantie

Omwille van de aanpassingen van de VBR's is er een kleine stijging in de kost van het toevoegen, verwijderen of updaten van een feature-vector of knoop ten opzichte van de SR-tree.
De A-tree en SR-tree vragen evenveel opslagkost vanaf 4 tot en met 32 dimensies. Hierna vraagt de A-tree minder opslagkost. Bij 64 dimensies blijkt experimenteel dat de A-tree 19.5% minder opslagkost vraagt dan de SR-tree. Voor niet-uniforme data is de A-tree sneller dan de SR-tree. Zo moet de A-tree bij 64 dimensies 77.3% minder blokken bekijken dan de SR-tree en 77.7% minder blokken dan de VA-file.

4.5 Tree Striping

4.5.1 Motivatie

We weten dat de performantie van multidimensionale indexen die partitioneren/clusteren daalt naarmate het aantal dimensies stijgt. Het probleem hierbij is niet het aantal punten zoals we konden afleiden uit de experimenten, maar het aantal fractale dimensies. Hoe kleiner de fractale dimensie, hoe beter de performantie. We zouden dus een multidimensionale index kunnen opsplitsen in meerdere indexen om de dimensionaliteit van elk index te beperken. Bij het opsplitsen in meerdere indexen, indexeert elke index één of meer dimensies van de oorspronkelijke Euclidische dimensies. De waarden van een bepaalde dimensie worden hierbij dus niet verdeeld over twee of meer indexen. We maken hierbij een afweging tussen hoeveel moeite het kost om een NN-query te doen op één multidimensionale index en hoeveel het kost om de resultaten van meerdere (multidimensionale) indexen samen te voegen tot één resultaat. Met andere woorden, hoe hoger de dimensionaliteit van een index, hoe meer kans dat de performantie van de index naar die van een lineaire scan evolueert. Echter, hoe hoger de dimensionaliteit van een index, hoe groter de selectiviteit van de index bij een query, hoe minder resultaten de index zal teruggeven bij een query, hoe minder het kost om de deelresultaten samen te voegen. De twee extremen hierbij zijn dan inverted lists of één multidimensionale index. Bij Inverted lists wordt voor elke dimensie een 1-dimensionale index aangemaakt. Dit zijn de principes waarop Treestriping [31] gebaseerd is.

4.5.2 Structuur

Gegeven zijn een D-dimensionale dataruimte R en N D-dimensionale featurevectoren v. Het j-de element van v duiden we aan met v_j.

Definitie Dimensie-toekenning. De *dimensie-toekenning* DT is een afbeelding $R^D \rightarrow (R^{D_0}, \cdots, R^{D_{k-1}})$. Het is dus een mapping van een D-dimensionale vector v naar k D_l-dimensionale vectoren w^l, waarbij $0 \leq l \leq k$, zodat de volgende condities gelden:

1. $\sum_{l=0}^{k-1} D_l = D$
2. $\forall j : 0 \leq j < D, \exists l : 0 \leq l < k, \exists i : 0 \leq i < D_l : v_j = w_i^l$ (alle D dimensies, komen voor in de opdeling)

3. $\forall l : 0 \leq l < k, \forall i : 0 \leq i < D_l, \exists j : 0 \leq j < D : w_i^l = v_j$ (alle dimensies D_l van alle vectoren w^l komen voor in de originele D dimensies)

De originele index heeft D dimensies. Er zijn k nieuwe indexen. Elke index heeft D_l dimensies.

Definitie Tree Striping. Geven een database DB van N D-dimensionale vectoren en een dimensie-toekenning DT. Een *Tree Striping TS* is gedefiniëerd als een vector van k D_l-dimensionale vectoren:

$$MIS^l = \{w^l\}, w \leq l \leq k, \text{met } w^l = DT^l(v), v \in DB$$

Als $k = D$, dan hebben we de inverted lists en als $k = 1$, dan hebben we één multidimensionale index.
Bijvoorbeeld, zij $D = 5$, $k = 2$ en $DT_{(on)even}$ de dimensietoekenning waarbij we aan de eerste index alle oneven dimensies toekennen en aan de tweede index alle even dimensies. Een feature-vector $(5, 0, 2, 3, 8)$ wordt dan geïndexeerd volgens $DT_{(on)even}$ door $(5, 2, 8)$ te plaatsen in de eerste index en $(0, 3)$ in de tweede index.
Om de globale kostformule in sectie 4.5.4 te vereenvoudigen, zullen we veronderstellen dat we aan elke index evenveel dimensies toekennen. Met behulp van het globale kostmodel in sectie 4.5.4 bepalen we dan een optimale k. Het getal k is een reëel nummer en daarom stellen we $k_{opt} = \lfloor k \rfloor$. De optimale dimensie voor elke deelboom is dan $D_{opt} = \lfloor \frac{D}{k} \rfloor$. Aangezien $k_{opt} \cdot D_{opt} \leq D$, moeten we de overige dimensies $D_{overig} = D - (k_{opt} \cdot D_{opt})$ verdelen over de k_{opt} deelbomen. Er geldt dat $D_{overig} < k_{opt}$. Immers, uit $D - (k_{opt} \cdot D_{opt})) >= k_{opt}$ kan men een contradictie afleiden. Er zullen bijgevolg D_{overig} deelbomen zijn met $D_{opt} + 1$ dimensies en $k_{opt} - D_{overig}$ deelbomen met D_{opt} dimensies.
Het kan voorkomen dat een bepaalde dimensie een grotere selectiviteit heeft dan een andere, met andere woorden dat deze minder elementen teruggeeft bij een query. Dit kan zijn omwille van de aard van de waarden. Bijvoorbeeld, de verzameling van mogelijke waarden is groot of bepaalde dimensies hebben een grotere selectiviteit omwille van de distributie van de waarden (afhankelijke variabelen hebben geen invloed op de performantie/kost van het zoeken). Bijvoorbeeld, in [3] toont men hoe men dimensies kan selecteren met behulp van de fractale dimensie. Om een kleinere resultaatset van een deelindex terug te krijgen, kan men het aantal dimensies dat men toekent aan een deelindex wijzigen om zo de dimensies met een hoge selectiviteit te groeperen.
Veronderstel dat de selectiviteit van alle dimensies even groot is of we weten er verder niks van, dan kunnen we de volgende optimale dimensietoekenning opstellen:

De optimale dimensie-toekenning TK_{opt} is een dimensie-toekenning TK waarvoor geldt:

- voor $0 \leq l < k_{opt}$, het aantal dimensies van elke deelindex is

$$D_l = \begin{cases} D_{opt} + 1 & \text{als } l < D_{overig} \\ D_{opt} & \text{in de andere gevallen} \end{cases}$$

- $0 \leq i < D_l$

- $DT^l(v)_i = w_i^l = \begin{cases} v_{(l \cdot (D_{opt}+1)+i)} & \text{als } l < D_{overig} \text{ (sequentiële dimensietoekenning)} \\ v_{D_{overig} \cdot (D_{opt}+1)+(l-D_{overig}) \cdot D_{opt}+i} & \text{in de andere gevallen} \end{cases}$

Als we wel meer informatie hebben over de selectiviteit van de dimensies, dan kunnen we de toekenning bekomen uit bovenstaande definitie aanpassen om hiervan gebruik te maken. We zouden zoveel mogelijk dimensies die een hoge selectiviteit hebben samen kunnen plaatsen in één boom. Men zou hiervoor dus eerst de dimensies op selectiviteit kunnen ordenen en dan de toekenning doen.

4.5.3 Operaties

Toevoegen en verwijderen Om een feature-vector toe te voegen, verdelen we de vector over de verschillende deelindexen volgens de dimensietoekenning. Bij het verwijderen van een feature-vector wordt de feature-vector verwijderd uit alle deelindexen.

Queryen Een algoritme zou kunnen zijn dat we de query opsplitsen in deelqueries waarbij elke deelquery een deelindex ondervraagt. We kunnen echter enkele opmerkingen maken om dit algoritme te verbeteren:

- als de query slechts een deel van het totaal aantal dimensies vereist om geëvalueerd te worden, query dan alleen de vereiste deelindexen.
- extra deelindexen queryen kan kostelijker zijn dan de voorlopige resultaten te controleren op de overige voorwaarden
- query eerst de deelindexen die de kleinste resultaatset teruggeven. Als men bij de dimensietoekenning de dimensies ordent op selectiviteit, kan men in deze situatie gebruik maken van deze ordening.

Het query-algoritme wordt beschreven in algoritme 4.5.

algoritme 4.5 Tree-Striping query-algoritme

```
IdSet Query(TreeStriping ts, Query query) {

IdSet resultset;
/* Sorteer de deelindexen volgens dalende selectiviteit, hierbij ook rekening
houdend welke dimensies in de query gebruikt worden. De functie geeft
terug hoeveel deelindexen we moeten queryen. */
aantalIndexen = ts.sorteerDalendeSelectiveit(query);
/* Bepaal voor alle nodige deelindexen een subquery. De eerste subquery
zou dus de grootste selectiviteit moeten hebben. */
for 0 ≤ i < aantalIndexen do
   Queries SubQueries[i] = query.MaakSubQuery(ts.DT[i]);
end for

/* Gebruik het globaal model en een model voor de lineaire scan om te
bepalen wat het voordeligst is */
if aantalIndexen > 0 then
   IdSet deelIndexResultsets[aantalIndexen];
   kostIndex = kostModelIndex(subQueries[0]);
   kostLineair = kostModelLineaireScan(subQueries[0]);
   for i=0; i < aantalIndexen AND kostIndex < kostLineair; ++i do
      deelIndexResultset[i] = ts.index[i].query(subQueries[i]);
      /* Sorteer volgens het id van de feature-vectoren */
      deelIndexResultset[i].sorteer();
      if i+1 < aantalIndexen then
         kostIndex = kostModelIndex(subQueries[i+1]);
         kostLineair = kostModelLineaireScan(deelIndexResultset[<i+1]);
      end if
   end for
   resulset.intersect(deelIndexResultset, aantalIndexen);
   /* Lineaire scan was goedkoper dan overige deelindexen te queryen */
   if i < aantalIndexen then
      database.load(resultset);
      /* verwijder de feature-vectors die niet voldoen aan de query */
      resultset = query.voeruit(resultset);
   end if
end if
return resultset;

}
```

4.5.4 Performantie

Nu moeten we onderzoeken of deze methode eventueel een voordeel oplevert. Hiervoor hebben we dus het volgende nodig:

- een kostmodel voor het queryen van de verschillende indexen
- een kostmodel voor het samenvoegen van de resultaten

Kostmodel queryen We tonen hier de berekening voor de verwachte kost van een query waarbij het gevraagde gebied geen hyperbol is, maar een hyperkubus K. De ribben van K hebben een lengte l. Dit model houdt geen rekening met boundary effects, het MBR-effect (sectie 5.3 en de fractale dimensie. Het is dus hoofdzakelijk nuttig voor gebruik in laag- en medium-dimensionale ruimten. Men kan echter dit model vervangen door een meer nauwkeurig model voor hoog-dimensionale ruimten.

Gegeven is dat er N feature-vectoren zijn en C_{eff} feature-vectoren per bounding regio. Als een waarde van elke dimensie vier bytes inneemt en er voor elke feature-vector een id is van 4 bytes, dan kunnen we C_{eff} schatten als

$$C_{\mathit{eff}} = \frac{\mathit{page\text{-}size} \cdot \mathit{gebruikte\ opslag}}{4 \cdot (D+1)}$$

We stellen de bounding regio's voor als hyperkubussen. Het gemiddelde volume van een bounding regio O kunnen we dan ruw voorstellen als N/C_{eff}. We houden hierbij dus geen rekening met het MBR-effect. De lengte van een ribbe van de bounding regio is dan $\sigma = \sqrt[D]{N/C_{\mathit{eff}}}$. Dan kunnen we zonder rekening te houden met boundary effects specifiëren dat

$$\begin{aligned} Vol_{O \oplus K} &= (\sigma + l)^D \\ &= \left(\sqrt[D]{\frac{N}{C_{\mathit{eff}}}} + l \right)^D \end{aligned}$$

Het aantal geïntersecteerde bounding regio's A kunnen we dan modelleren als

$$\begin{aligned} A &= \frac{N}{C_{\mathit{eff}}} \cdot Vol_{O \oplus K} \\ &= \left(1 + l \cdot \sqrt[D]{\frac{N}{C_{\mathit{eff}}}} \right)^D \end{aligned}$$

Kostmodel samenvoegen van resultaten Vervolgens moeten we nog een kostmodel hebben voor het samenvoegen van de resultaten van de verschillende indexen. Uit de constructie van de verschillende bomen blijkt dat elke index alle feature-vectoren bevat. Elke deelquery op een deelboom geeft een aantal feature-vectoren terug die eraan voldoen. Vervolgens moet de doorsnede van de feature-vectoren genomen worden. Feature-vectoren in de doorsnede zijn de vectoren die aan alle deelqueries voldoen en zitten dus in het resultaat.
Om de feature-vectoren vlug te kunnen filteren, sorteren we ze eerst op hun id. Om een gebrek aan geheugen te voorkomen, gebruikt men de multi-way merge-sort. De kost van deze operatie is $2 \cdot B \cdot \log_M(B)$, waarbij B het aantal blokken is dat gesorteerd moet worden en M het aantal beschikbare cache-pagina's voorstelt. Zij $|ERS|$ het uiteindelijk aantal resultaten. Hoeveel tussenresultaten $|TRS_i|$ kunnen we dan van index i verwachten als deze D_i dimensies indexeert? Wegens de distributie van de punten (zie sectie 2.2) is dan $|TRS_i|/N = q^{D_i} = (|ERS|/N)^{D_i/D}$. We hebben afgesproken dat een feature-vector-id 4 bytes inneemt. Het aantal blokken B dat gesorteerd moet worden, is dan

$$B = \frac{4 \cdot (N \cdot q^{D_i})}{\textit{page-size}}$$

Om de tussenresultaten van de verschillende deelindexen samen te voegen, vermeldt men dat de resultaten van elk deelindex nog één extra keer overlopen moeten worden. Dit is geen precieze aanpak van de situatie. Bijvoorbeeld, als er maar één deelindex is, dus $k = 1$, hoeven de tussenresultaten niet samengevoegd en gesorteerd te worden. Om in de andere gevallen te bekomen dat de tussenresultaten maar één keer hoeven gelezen te worden, moet er in het geheugen plaats zijn voor k blokken, anders moet er tussendoor ook naar het opslagmedium geschreven worden.

Globaal kostmodel Als we opsplitsen in k indexen, dan is het globale kostmodel:

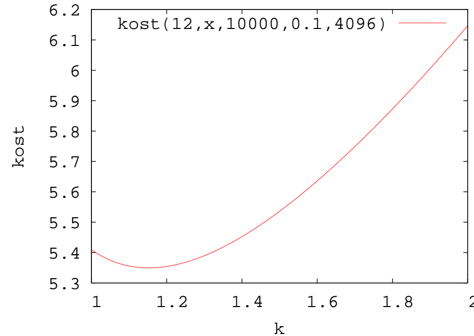
$$\textit{Kost}(D_i, N, q) = \sum_{i=1}^{k} \left(\left(1 + q \cdot \sqrt[D_i]{\frac{N}{C_{\textit{eff}}(D)}} \right)^{D_i} + \frac{4 \cdot (N \cdot q^{D_i})}{\textit{page-size}} \cdot \left(1 + 2 \cdot \log_M \left(\frac{4 \cdot (N \cdot q^{D_i})}{\textit{page-size}} \right) \right) \right)$$

We kunnen dit vereenvoudigen door de D dimensies gelijkmatig te verdelen over k indexen. Elke index i indexeert dan $D/k(= D_i)$ dimensies. Het kost-

model evolueert dan tot:

$$Kost(D,k,N,q) = k \cdot \left(\left(\left(1 + q \cdot \left(\frac{N}{C_{\mathit{eff}}\left(\frac{D}{k}\right)} \right)^{\frac{k}{D}} \right)^{\frac{D}{k}} + \frac{4 \cdot (N \cdot q^{\frac{D}{k}})}{\mathit{page\text{-}size}} \cdot \left(1 + 2 \cdot \log_M \left(\frac{4 \cdot (N \cdot q^{\frac{D}{k}})}{\mathit{page\text{-}size}} \right) \right) \right) \right)$$

In het laag-/medium-dimensionale geval maken we de veronderstelling dat we alle D dimensies opsplitsen. Dit volgt uit de berekening van σ, waarbij we veronderstellen dat de lengte in alle dimensies even groot is. Hierom nemen we voor N een waarde groter dan 2^D. In figuur 9 is de grafiek van het kostmodel getekend voor de volgende parameters: $\mathit{page\text{-}size} = 4kb$, $\mathit{gebruikteopslag} = 90\%$, $D = 12$, $N = 10000$, $M = 10$ en de query-hyperbol heeft een straal met lengte 0.1.



Figuur 9: grafiek kostmodel

Een optimale k bevindt zich in deze situatie rond 1.2. Eén index is dus hier een optimale keuze volgens bovenstaand kostmodel.

Experimentele resultaten Uit experimenten met een globaal kostmodel waarbij het query-kostmodel aangepast is voor hoog-dimensionale ruimten, blijkt dat het aantal k's bepaald door het kostmodel overeenkomt met de optimale k die bepaald werd door k te variëren. Het blijkt ook dat Tree Striping de performantie van het queryen met meer dan 100 procent kan versnellen ten opzichte van de performantie van de twee uitersten, inverted lists en één multidimensionale index. Ten opzichte van één multidimensionale

index werd een performantiestijging tot en met 310 procent waargenomen. In vergelijking met inverted lists werd een performantiestijging tot en met 12300 procent waargenomen.

4.6 Bulk Loading

Motivatie We zijn er vanuit gegaan dat insert- en verwijder-operaties niet zo snel hoeven te gebeuren ten opzichte van het zoeken in de index. We hebben tot nu toe ook aangenomen dat als we iets toevoegen, we alleen maar beschikken over de info van één vector. Het kan echter zijn dat we zicht hebben op alle info die in een lege index geplaatst moet worden. Een voorbeeld hiervan is het toevoegen van de eerste data. Wegens het weinig aantal data-elementen zal hier een lineaire scan sneller zijn dan een index. We kunnen dan Bulk Loading gebruiken om, gebruikmakend van deze extra info, een goede index sneller te bouwen. De indexen zorgen ook voor een goede query-performantie, plaatsgebruik, ...

Er bestaan meerdere bulkloading-technieken, maar we beschrijven hier een algemene techniek die op heel veel indexen kan toegepast worden. Het algoritme uit [33] en [2] bestaat uit volgende delen:

1. Bepalen van de hoogte h van de index en de daarmee gepaard gaande fan-out f
2. De splitsstrategie
3. Opdelen van de data in f delen
4. Herhalen van stappen 2 en 3 voor elk van de f kinderen totdat we een data-knoop bekomen
5. Tijdens de recursie de knopen naar de schijf schrijven.

Het globaal algoritme is beschreven in algoritme 4.6. We bouwen de index bottom-up op.

4.6.1 Bepalen van de hoogte h en de fan-out f

We gebruiken beschikbare data die niet meer verandert om de structuur van de index vooraf te bepalen. De statische data bestaat uit het aantal feature-vectoren N die we willen indexeren, de dimensionaliteit D, de gemiddelde capaciteit en hoeveel we van die capaciteit gemiddeld willen gebruiken, aangeduid met $capaciteit_{gebruikt}$.

algoritme 4.6 Bulkload

Input: $C_{max,nav}, C_{max,data}, capaciteit_{gebruikt}, N,$ feature-vectoren

Output: index voor de feature-vectoren

```
Indexeer() {
Bepaal h met behulp van formule 11
/* We construeren de boom in pre-orde. */
maakKnoop(h, 0, N);
}
```

algoritme 4.7 maakKnoop

Input: $C_{max,nav}, C_{max,data}, capaciteit_{gebruikt}, h,$ een opeenvolgende reeks feature-vectoren

Output: een boom voor de reeks feature-vectoren

```
/* We doen alsof de feature-vectoren in een array zitten en beginPos
is een beginlocatie en aantal is een aantal vectoren zodat [beginPos–
beginPos+aantal-1] een interval vectoren definiëert. */
maakKnoop(h, beginPos, aantal) {
if 1 == h then
  dataknoop aanmaken en wegschrijven naar het opslagmedium en geef
  MBR door aan de ouder
  return
end if
/* navigatieknoop aanmaken */
/* Bepaal aantal kinderen voor de navigatieknoop */
f = fanout(h, N);
/* Bepaal de gewenste verdeling van de feature-vectoren over de kinderen
met behulp van de splittree */
SplitTree st = createSplitTree(h, beginPos, aantal, f);
/* Split feature-vectoren op volgens de Splittree */
splitsVectoren(h, beginPos, aantal, st);
/* feature-vectoren zijn verdeeld over kinderen, schrijf navigatieknoop weg
*/
schrijf navigatieknoop naar het opslagmedium
}
```

algoritme 4.8 splitsVectoren

Input: $C_{max,nav}, C_{max,data}, h,$ een opeenvolgende reeks feature-vectoren, SplitTree

Output: de reeks feature-vectoren zijn verdeeld onder de kinderen van een knoop gebruikmakende van de SplitTree

/* Doorloop de splittree in pre-orde en splits de vectoren volgens de info van elke navigatieknoop van de splittree */
splitsVectoren(h, beginPos, aantal, SplitTree st) {
if is_blad(st) **then**
  /* (beginPos,aantal) feature-vectoren zijn voor het kind van deze bladknoop */
  maakKnoop($h - 1$, beginPos, aantal);
  return
end if
Bepaal $N_{max,links}$ en $N_{max,rechts}$
/* splits feature-vectoren in twee delen */
aantalVectorenLinks = bipartionering(beginPos, aantal, st.splitDimensie, $N_{max,links}$, $N_{max,rechts}$);
splitsVectoren(h, beginPos, aantalVectorenLinks, st.linkerkind);
splitsVectoren(h, beginPos+aantalVectorenLinks, aantal-aantalVectorenLinks, st.rechterkind);
}

Het maximaal aantal feature-vectoren per dataknoop is

$$C_{max,data} = \left\lfloor \frac{dataknoopcapaciteit}{sizeof(feature\text{-}vector)} \right\rfloor$$

Het gemiddeld aantal feature-vectoren per dataknoop in de index is

$$C_{eff,data} = capaciteit_{gebruikt} \cdot C_{max,data}$$

Analoge definities kunnen we voor navigatieknopen definiëren.
Het maximaal aantal kinderen per navigatieknoop is

$$C_{max,nav} = \left\lfloor \frac{navigatieknoopcapaciteit}{sizeof(info\ over\ kind)} \right\rfloor$$

Het gemiddeld aantal kinderen per navigatieknoop is

$$C_{eff,nav} = capaciteit_{gebruikt} \cdot C_{max,nav}$$

In een boom met een hoogte h, is dan het maximaal aantal feature-vectoren gelijk aan

$$C_{max,index}(h) = (C_{max,nav})^{h-1} \cdot C_{max,data} \tag{9}$$

Analoog is in een boom met een hoogte h, het gemiddeld aantal feature-vectoren

$$C_{eff,index}(h) = (C_{eff,nav})^{h-1} \cdot C_{eff,data} \tag{10}$$

In deze formule is $(C_{eff,nav})^{h-1}$ het aantal dataknopen. Bijgevolg, om te bekomen dat alle N feature-vectoren in de boom opgenomen kunnen worden, moet de boom een minimale hoogte h hebben waarbij,

$$h = \left\lceil \log_{C_{eff,nav}} \left(\frac{N}{C_{eff,data}} \right) \right\rceil + 1 \tag{11}$$

Formule 11 kan men bekomen uit formule 10 waarbij $C_{eff,index}(h) = N$. Men kan er uit afleiden dat we alleen voor bepaalde waarden van N de beschikbare capaciteit voor honderd procent kunnen gebruiken.
Uit bovenstaande gegevens kunnen we vervolgens de fan-out van de root-node bepalen. De fan-out is het aantal kinderen dat de root-node nodig heeft om alle N feature-vectoren te kunnen opslaan, waarbij we aannemen dat de boomstructuur van elk kind gevuld is met $C_{eff,index}(h-1)$ feature-vectoren. De maximale fan-out is $C_{max,nav}$ De fan-out van de root-knoop is

dan:

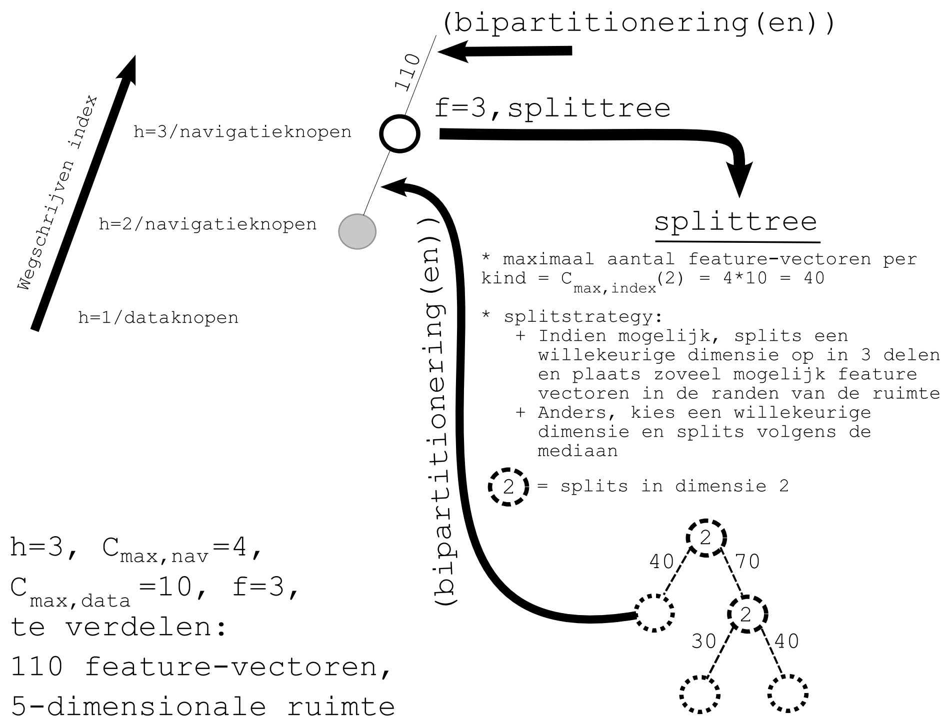
$$fanout(h, N) = \left\lceil \frac{N}{C_{eff,index}(h-1)} \right\rceil$$

De maximale fan-out van een navigatieknoop is $C_{max,nav}$. Met deze fan-out-formule kunnen we voor elke navigatieknoop I, gegeven het aantal feature-vectoren op te slaan in de subboom met I als wortel, bepalen hoeveel kinderen I moet hebben om alle feature vectoren te kunnen opslaan. Op deze manier probeert men het gewenste capaciteitsgebruik te benaderen. De fout die de afronding naar boven veroorzaakt, zal groter zijn voor navigatieknopen bovenaan in de index dan onderaan, aangezien h dan groter is. De fout wordt dan gecompenseerd op een niveau lager waar er dan een lager dan gemiddeld capaciteitsgebruik zal zijn als men niet afrond naar boven. Omdat onderaan in de index de afronding minder erg is, zal daar telkens het capaciteitsgebruik goed benaderd worden. Als we voor een navigatieknoop de fan-out f bepaald hebben, dan moeten we de beschikbare data in f kinderen onderbrengen. Hoe we de data over de f kinderen verdelen, wordt in een eerste stap bepaald door een split-tree op te stellen.

4.6.2 Splittree

Als we met behulp van de fanout-formule bepaald hebben hoeveel kinderen een navigatieknoop I heeft, dan moeten we nog de feature-vectoren verdelen over de kinderen. Hiervoor kan men een algoritme implementeren dat de splittree opstelt. Deze splittree vertelt hoe de feature-vectoren verdeeld worden over de f kinderen. Het algoritme om de splittree op te stellen ligt niet vast. De splittree moet aan bepaalde voorwaarden voldoen. Een splittree is een binaire boom met f bladeren, waarin elke navigatieknoop info bevat over hoe de feature-vectoren, behorende tot het gebied van de navigatieknoop, in twee delen gesplitst moet worden. Elke navigatieknoop van de splittree kan bijvoorbeeld informatie bevatten in welke dimensie gesplitst wordt en hoeveel feature-vectoren aan elke kant van de splitsing zich bevinden. We moeten er op letten dat elk kind niet meer feature-vectoren toegekend krijgt dan de subboom waarvan het kind de wortel is, kan bevatten. Dit kunnen we bepalen met formule 9. Als de splittree wordt opgesteld voor een navigatieknoop op niveau h, dan is de maximumcapaciteit van een bladknoop van de splittree $C_{max,index}(h-1)$. Een bladknoop van de splittree komt immers overeen met een kind van de respectievelijke navigatieknoop. De info die men nodig heeft om te bepalen waar men splitst, hangt af van het geïmplementeerde algoritme. Het kan echter zijn dat niet alle feature-vectoren in het geheugen

passen. In plaats van de beslissingen te baseren op alle feature-vectoren, kan men een sample nemen dat wel in het geheugen past. Men vermijd hier best random zoeken. Zo kan men bijvoorbeeld opeenvolgende blokken van drie plaatsen in de dataset nemen. De manier waarop de data uiteindelijk opgesplitst wordt zal ervoor zorgen dat er niet teveel feature-vectoren aan een bepaald kind worden toegekend. Het algoritme dat de splittree opstelt noemt men de *splitstrategy*. In figuur 10 kan men een splittree zien voor een navigatieknoop op niveau 3 (=h). We weten dat we 110 feature-vectoren moeten verdelen over de kinderen van de navigatieknoop. We berekenen dan de fan-out waarvan we veronderstellen dat deze 3 (=f) is. Vervolgens stellen we de splittree op van de navigatieknoop volgens de splitstrategy, rekening houdend met het maximaal aantal feature-vectoren dat een kind van de navigatieknoop kan bevatten.



Figuur 10: splittree voor een navigatieknoop op niveau 3

4.6.3 Opsplitsen van de data

De volgende stap is het opsplitsen van de data gebruikmakende van de gegevens in de splittree. Om het opsplitsen eventueel te versnellen, gebruikt

men niet de waarden van elke navigatieknoop, die vertellen hoeveel feature-vectoren elk kind zal krijgen. In plaats hiervan, gebruikt men het interval waarin de mogelijke verdelingen van de feature-vectoren zich bevinden. We weten dat elk van de f kinderen maximaal $C_{max,index}$ feature-vectoren kan bevatten. Dit maximum en het totaal aantal feature-vectoren stelt ons in staat een minimum te berekenen. We kijken hiervoor naar het aantal bladeren in de splittree aan de twee kanten van de opsplitsing. Zij l het aantal bladeren aan de linkerkant van de opsplitsing en r het aantal bladeren aan de rechterkant. De splittree is opgesteld voor een navigatieknoop op niveau h van de index. Het maximaal aantal feature vectoren dat we aan de linkerkant van de opsplitsing kunnen hebben is dus $N_{max,links} = l \cdot C_{max,index}(h-1)$. Analoog kunnen we voor de rechterkant schrijven dat $N_{max,rechts} = r \cdot C_{max,index}(h-1)$. De mogelijke aantallen feature-vectoren die men aan de linkerkant kan toekennen, worden gegeven door het interval $[N - N_{max,rechts}, N_{max,links}]$. Als het aantal voor de linkerkant niet in dit interval ligt, dan kunnen niet alle feature-vectoren geïndexeerd worden in de vooropgestelde structuur. Als we een kleinere *gebruikteopslag* gebruiken, dan rekenen we op minder elementen per knoop. We krijgen dan een hogere h en een kleinere fan-out is dan mogelijk per knoop. Het maximaal aantal feature-vectoren per subboom blijft echter hetzelfde. De splittree kan dan flexibeler opgesteld worden. We weten nu hoe we de feature-vectoren van een navigatieknoop onder zijn kinderen willen verdelen. Nu beschrijven we het bipartitioneringsalgoritme dat de werkelijke opsplitsing doet. Dit algoritme is gebaseerd op het quicksort-algoritme en de volgende drie randvoorwaarden:

- Binnen elk van de f delen hoeven de feature-vectoren niet gesorteerd te worden.
- De recursie stopt als de pivot-waarde zich binnen het interval $[N - N_{max,rechts}, N_{max,links}]$ bevindt.
- De pivot-waarden worden gekozen volgens de positie van de splitsingslocatie in de splittree en niet om de data specifiek in het midden te delen.

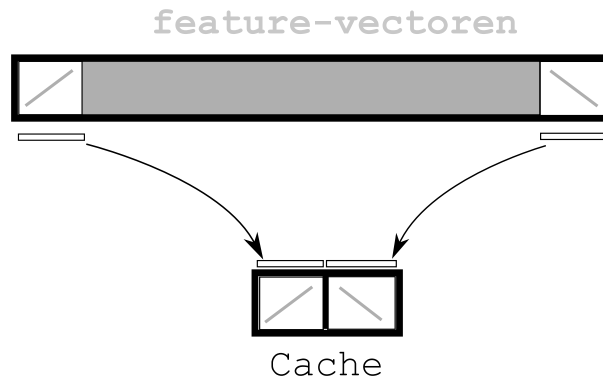
Er zijn twee versies van dit algoritme: een interne versie als alles in het geheugen kan geplaatst worden en een externe versie als het algoritme niet volledig in het geheugen kan uitgevoerd worden.

De *interne versie* is het quicksort-algoritme met bovenstaande drie aanpassingen. Kort samengevat ziet dit algoritme er als volgt uit. Men neemt drie feature-vectoren. Aangezien we ordenen op de splitsingsdimensie, bekijken we alleen de waarden van deze specifieke dimensie. We nemen de middelste waarde van de drie feature-vectoren als pivot-waarde. De feature-vector

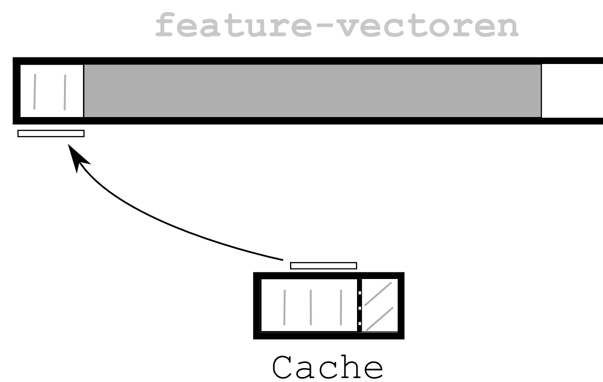
waartoe deze pivot-waarde behoort, noemen we het bisectie-punt. Vervolgens wordt gebruikmakende van het quicksort-algoritme de data geordend zodanig dat alle waarden groter dan de pivot-waarde rechts van de pivot staan en alle waarden kleiner dan de pivot-waarde links ervan. Als het bisectie-punt in het interval ligt, dan stoppen we de recursie. Anders passen we bovenstaande procedure opnieuw toe op het deel dat het interval bevat. De uiteindelijke splitsingslocatie specifieert hoeveel punten aan elk deel kan toegekend worden.

Bij de *externe versie* moeten we bijkomende maatregelen nemen aangezien niet alle data in het geheugen wordt geladen. Zoals in het splittree-gedeelte, gebruiken we ook hier een steekproef van de data. We voeren het bovenstaande interne algoritme erop uit op de sample om een geschikte pivot-value te bepalen. Vervolgens delen we de data op ten opzichte van deze pivot-waarde. We willen weten hoeveel data links en rechts van de pivot-waarde zich bevinden. We vervoeren hierbij data naar en van de cache in blokken met als grootte de helft van de cachegrootte. De data is sequentieel opgeslagen. De cache wordt eerst gevuld met de eerste en de laatste blok feature-vectoren (Figuur 11). Op de cache wordt vervolgens het intern algoritme uitgevoerd waardoor de data dus in twee delen wordt verdeeld. Van het grootste deel schrijven we opeenvolgende feature-vectoren vanaf het bisectiepunt en ter grootte van een blok terug naar het opslagmedium (Figuur 12). We kiezen het deel met de meeste feature-vectoren omdat deze tenminste de grootte van een blok heeft. Als het linkerdeel het grootst is, dan schrijven we de feature-vectoren ter grootte van een blok naar de zojuist gelezen blok aan de linkerkant. Als het rechterdeel het grootst is, dan schrijven we de feature-vectoren ter grootte van een blok naar de zojuist gelezen blok aan de rechterkant. Als we links, respectievelijk rechts hebben teruggeschreven, dan laden we de volgende blok aan de linkerkant, respectievelijk de rechterkant in de lege plaats in de cache (Figuur 13). De cache is op dit moment dan weer volledig gevuld. We hebben immers maar feature-vectoren ter grootte van één blok uit de cache weggeschreven. Van de oude data weten we wel al in welk deel het zich zal gaan bevinden. We voeren deze procedure uit totdat we alle blokken ingeladen hebben. Immers, dan is de data opgesplitst ten opzichte van de pivot-waarde. Als het bisectiepunt zich in het interval bevindt, dan stoppen we de recursie, anders voeren we bovenstaande procedure uit op het deel waarin zich het interval bevindt.

Deze procedure wordt uitgevoerd voor elke navigatieknoop van de splittree. Elk kind van de index krijgt hierdoor dus feature-vectoren toegewezen.



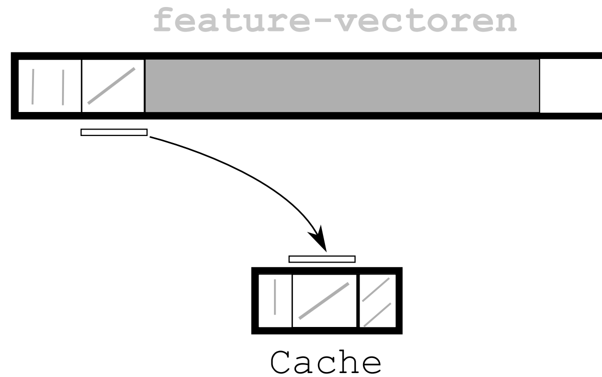
Figuur 11: Voor het sorteren: eerste blokken worden ingeladen



Figuur 12: Eerste blokken zijn gesorteerd

4.6.4 Bouwen van index en schrijven naar schijf

Als de recursie terugkeert van een kind in de index, dan krijgen we de bounding box van het kind en het adres van het kind op het opslagmedium terug. Als van alle kinderen van een knoop de recursie is teruggekeerd, dan kunnen we een navigatieknoop aanmaken, met de teruggekregen info over de kinderen, . . . en kunnen we deze naar het opslagmedium schrijven. De bounding boxen van de kinderen worden gecombineerd tot één omvattende bounding box. Deze omvattende bounding box en het adres van de nieuwe navigatieknoop worden doorgegeven naar de eventuele bovenliggende navigatieknoop aangezien de recursie dan terugkeert. Op het onderste niveau worden de dataknopen weggeschreven.
De index wordt op deze manier in pre-orde naar het opslagmedium geschre-



Figuur 13: Nieuwe blok wordt ingeladen

ven. Dataknopen die in de ruimte dicht bij elkaar liggen, zullen ook op het opslagmedium dicht bij elkaar liggen. De dataknopen bevinden zich dus op een geclusterde manier op het opslagmedium, wat het queryen bevordert.

4.6.5 Performantie

In [2] toont men aan dat de index-constructie een gemiddelde tijdscomplexiteit van $O(N \cdot \log(N))$ heeft, tenzij de tijdscomplexiteit van de splitstrategy groter dan $O(N)$ is. Men berekent ook het verschil in random-seek-operaties tussen deze bulkload-methode (waarbij de feature-vectoren symmetrich gesplitst worden) en de X-tree. Zij C_{cache} het aantal feature-vectoren die in cache passen, dan is *verbetering* het aantal random seeks dat de bulkload-methode minder verricht dan bij gebruik van de X-tree, waarbij

$$verbetering \approx \frac{C_{cache}}{4 \cdot \log_2\left(\frac{N}{C_{cache}}\right) \log_{C_{max,nav}}\left(\frac{N}{C_{cache}}\right)}$$

In [2] vergelijkt men de bulkload-methode experimenteel met de X-tree. Men initialiseert een X-tree met 16-dimensionale feature-vectoren. Het aantal vectoren varieert men van honderdduizend tot en met twee miljoen. De tijd om de X-tree te construeren varieerde van 965 seconden tot en met 393310 seconden (4 dagen en 13 uren). Als men dezelfde initialisaties uitvoert met behulp van de bulkload-methode, varieert de constructietijd (waarbij men de data symmetrisch opdeelt) van 26 tot en met 668 seconden.

4.7 Samenvatting

We hebben een overzicht gegeven van metrische indexen en Euclidische hoogdimensionale indexen. De Euclidische indexen zijn in meer detail besproken. We hebben gezien hoe de SR-tree een verbetering is op de SS-tree door een hyperbalk, naast de hyperbol bij te houden als bounding regions, om zo het volume van de bounding region te verkleinen. Dit leverde echter een slechtere fan-out op. De VA-file, die gebruikt maakt van "compressie", heeft een goede fan-out, maar werkt niet goed op niet-uniforme data, terwijl de SR-tree wel goed werkt op niet-uniforme data. We combineerden de SR-tree en de VA-file om de A-tree te bekomen. Hierin werd de structuur van de SR-tree grotendeels overgenomen en werd er gebruikt gemaakt van VBR's die gebaseerd zijn op de "compressie" van de VA-file. We weten echter dat omwille van the "Curse of Dimensionality" de performantie van indexen die clusteren/partitioneren eveoluëert naar die van een lineaire scan. We introduceren hiervoor Tree Striping, waarmee we deze evolutie proberen te verhinderen door een afweging te maken tussen meerdere indexen te gebruiken en de kost van het samenvoegen van de resultaten van de verschillende deelindexen. Uit experimenten blijkt dat Tree Striping eventueel voor heel wat extra performantie kan zorgen. We waren er echter vanuit gegaan dat het toevoegen van nieuwe elementen in de indexen niet snel hoefde te gaan, maar dat het queryen wel snel moest gaan. Om vanuit grote hoeveelheden data een index op te bouwen, vermelden we een algemene bulkloadtechniek. Hierbij bepalen we eerst de gewenste eigenschappen van de index en bouwen dan de index bottom-up op. We merken in een vergelijking met de X-tree op dat de bulkloadtechniek een goede index veel sneller kan opbouwen als er een grote hoeveelheid data moet geïndexeerd worden.

5 Queryen

We hebben nu bepaalde datastructuren om multidimensionale info op te slaan. Nu moeten we nog algoritmen hebben die ons toelaten deze structuren efficiënt te queryen. We zullen ons hierbij concentreren op de NN- en range-query. Bij multimediadatabases zijn incrementele algoritmen belangrijk vanwege de interactiviteit-vereiste die een gebruiker van de database kan vragen. Een gebruiker zoekt bijvoorbeeld naar de dichtstbijzijnde stad waar er een 4-sterren hotel is, met een kamer vrij voor de komende nacht. Men weet dus niet hoeveel data-objecten(steden en hotels) men gaat moeten doorlopen. Men zou eerst de vijf dichtstbijzijnde steden kunnen opvragen en dan de 10 dichtstbijzijnde steden, . . . totdat men een stad vindt waar een

hotel is die aan de gewenste eisen voldoet. Met een niet-incrementeel algoritme moet men echter alle oude resultaten opnieuw vinden, terwijl een incrementeel algoritme de oude resultaten niet meer opvraagt.

5.1 Single step incrementeel *k*-NN/range-algoritme

Met een single-step-algoritme bedoelt men dat men alleen werkt met de feature-vectoren. Dit is niet het geval bij multi-step-algoritmen (zie sectie 5.2). Van het algoritme dat we hier beschrijven [15] kan bewezen worden dat het optimaal is in het aantal blokken als in de index-hiërarchie geldt dat de wortels van de subbomen een minimumafstand tot de *objecten* in de betreffende subboom kunnen geven. Uit deze voorwaarde blijkt dat de minimumafstand belangrijk is en dat de maximumafstand niet noodzakelijk is om optimaliteit te bekomen. Verder nemen we dus aan dat de objecten onderverdeeld zijn in een hiërarchie en dat elk object maximaal één ouder heeft en maximaal één keer voorkomt. Het algoritme kan echter aangepast worden zodat de laatste twee voorwaarden niet vereist zijn.

Algoritme We geven alle verschillende typen knopen van een hiërarchie een nummer t. Een knoop van een bepaald type t duiden we aan met e_t. Alhoewel een feature-vector geen knoop hoeft te zijn, kennen we er het type e_0 aan toe. Veronderstel dat we een incrementele NN-query stellen over een verzameling feature-vectoren O, geördend in een hiërarchie H, met als query-punt q. Het incrementele NN-algoritme beschreven in algoritme 5.1 is optimaal in het aantal blokken.
We maken nog enkele opmerkingen in verband met de priorityqueue. Als de knopen/feature-vectoren in de queue gelijke afstanden hebben, geven we eerst voorkeur aan knopen met een lager type-nummer. We kiezen bijvoorbeeld feature-vectoren voor knopen. Als de type-nummers ook gelijk zouden zijn, dan geven we de voorkeur aan knopen dieper in de hiërarchie H. Hiermee trachten we zo vlug mogelijk knopen te vinden die in elkaars omgeving liggen.

Variaties Men kan het algoritme laten stoppen als men k resultaten heeft om zo een k-NN-query te beantwoorden. Een andere mogelijkheid is te stoppen als een bepaalde afstand tot q is overschreden. Men kan ook zoeken naar de verst verwijderde objecten door de afstandsfunctie van de priorityqueue aan te passen.

algoritme 5.1 incrementeel (k-)NN-algoritme

```
Queue(Vector q, Index H) {
/* knopen en feature-vectoren worden geördend van klein naar groot vol-
gens hun afstand tot q */
Queue prior = new PriorityQueue(e_t k, afstand d(q,k));
e_t dichtstbijQ = H.wortelVanIndex;
prior.enqueue(dichtstbijQ,0);
while not prior.isLeeg() do
   dichtstbijQ = prior.dequeue();
   /* dichtstbijQ is een feature-vector */
   if 0 == dichtstbijQ.t then
      rapporteer dichtsbijQ als volgende dichtsbijzijnde feature-vector
   else
      for i=0; i < dichtstbijQ.aantalKinderen; ++i do
         prior.enqueue(dichtstbijQ.child[i], d_t(q, dichtstbijQ.child[i]));
      end for
   end if
end while

}
```

Performantie Omwille van de extra flexibiliteit die dit algoritme geeft (incrementeel), zou men eventueel kunnen verwachten dat het wat slechter presteert dan andere algoritmen als men over de specifieke computationele kost spreekt met gebruik van een bepaalde index. In een vergelijking met methoden voor een vaste k, blijkt echter dat deze methode minstens even goed en veelal beter is op computationeel vlak voor verschillende indexen.

5.2 Optimal MultiStep k-NN algoritme

Motivatie Naast de single-step-algoritmen, zijn er dus ook multi-step-algoritmen. De bedoeling hiervan is dat men een afstandsfunctie kan definiëren over de werkelijke waarden van een object, dus niet over de feature-vectoren en dat de NN-query dan versneld wordt door gebruik te maken van de feature-vectoren van de data-objecten. Het gebruik van een multistep-algoritme kan voorkomen omdat men een complexe afstandsfunctie wil gebruiken of dat de dimensionaliteit van de feature-vectoren, vereist voor de complexe afstandsfunctie, misschien zo hoog zou zijn dat men niet efficiënt kan zoeken in de multidimensionale ruimte. Als alternatief voor k-multistep zouden we metrische indexen kunnen gebruiken waarbij we de afstandsfunctie op de werkelijke objecten gebruiken. Het nadeel hiervan is dat de afstandsfunctie vast ligt en niet door de gebruiker kan aangepast worden. Dit is wel mogelijk met het multistep-algoritme.

Algoritme Bij het zoeken in de multidimensionale ruimte wordt een vereenvoudigde afstandsfunctie gebruikt in plaats van de complexe functie. De vereenvoudigde afstandsfunctie noemt men de filter-afstandsfunctie of feature-afstandsfunctie. Als we filteren met de filter-functie in de multidimensionale ruimte willen we zeker zijn dat er geen objecten verloren gaan die in het resultaat zouden zitten als we de complexe afstandsfunctie op de echte objecten gebruikten. De enige voorwaarde om dit te bekomen is dat de filter-afstandsfunctie aan de *lower bounding-eigenschap* moet voldoen:

Definitie Lower bounding-eigenschap. Zij $d_{filter} : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}_0^+$ een filter-afstandsfunctie en $d_o : O \times O \rightarrow \mathbb{R}_0^+$, waarbij O het universum objecten voorstelt waarvoor d_o gedefiniëerd is, een object-afstandsfunctie. Zij $F : O \rightarrow \mathbb{R}^D$ een functie die elk element $o \in O$ afbeeldt op een D-dimensionale feature vector $F(o) \in \mathbb{R}^D$. De *lower bounding-eigenschap* geldt voor d_{filter} en $d_o \Leftrightarrow \forall o_1, o_2 \in O : d_{filter}(F(o_1), F(o_2)) \leq d_o(o_1, o_2)$.

In algoritme 5.2 beschrijven we een multi-step-algoritme dat geen objecten weglaat die in het eindresultaat zouden moeten zitten als de

filter-afstandsfunctie aan de lower bounding-eigenschap voldoet. Men kan erin twee soorten stappen onderscheiden: filter-stappen (met de feature-vectoren en de filter-afstandsfunctie) en verfijningsstappen (de objecten en de object-afstandsfunctie).

---

algoritme 5.2 niet-r-optimaal k-multistep-algoritme

Input: objecten en hun respectievelijke feature-vectoren, d_{filter}, d_o, k-NN-algoritme, voorbeeldobject en zijn feature-vector (=query-punt)

/* filter-stap */

1: we gebruiken de feature-ruimte en bepalen de k nearest neighbours volgens de afstandsfunctie d_{filter}

/* verfijningsstap */

2: bepaal het object met de grootste NN-afstand, gebruikmakende van de afstandsfunctie d_o, uit de verzameling bestaande uit de objecten van de feature-vectoren die we hebben bekomen in stap 1.

/* filterstap */

3: wegens de lower bounding eigenschap weten we dat een range-query met als straal, de NN-afstand volgens de afstandsfunctie d_o van het object verkregen uit de tweede stap, ook de echte k-NN-objecten bevat.

/* verfijningstap */

4: orden de objecten bekomen uit de derde stap van klein naar groot volgens hun NN-afstand, gebruikmakende van de afstandsfunctie d_o en kies de eerst k objecten (en ook de objecten met dezelfde range als het k-de object)

Aangezien de berekeningen op echte objecten zoveel tijd innemen, kan men vermoeden dat een optimaal algoritme bij een filterstap de berekeningen op de echte objecten zoveel mogelijk beperkt. Dit betekent dan dat de afstand in de tweede stap van algoritme 5.2 liefst de afstand tot de echte k-de-nearest neighbour zou zijn. We noemen een algoritme *r-optimaal* als het in de (eerste) filterstap zo weinig mogelijk kandidaten teruggeeft. Hoeveel objecten men terugkrijgt in de derde stap hangt af van de mate waarin de simpelere afstandsfunctie d_{filter} de objecten op dezelfde manier ordent als de afstandsfunctie d_o. Als dit de omgekeerde volgorde is, terwijl men dus nog altijd voldoet aan de lower bounding property, dan zullen bij die NN-query alle objecten met de afstandsfunctie D_o vergeleken worden. Als het exact dezelfde volgorde is, dan heeft men ook onmiddellijk exact de juiste elementen met de range-query van stap 3. Voor algoritme 5.3 kan men bewijzen dat het r-optimaal is en dat het ook geen objecten weglaat die in het resultaat zouden moeten zitten op voorwaarde dat de filter-afstandsfunctie aan de lower bounding-eigenschap voldoet.

algoritme 5.3 r-optimaal k-multistep-NN-algoritme

```
Objects k-multistep-NN(index H, Object q, k) {

algoritme ranking = index.increm_ranking(F(q), d_filter) ;
sorted_list result = new sorted_list(key, Object);
d_max = ∞;

while o = ranking.volgende() EN d_filter(o,q) ≤ d_max do
   if d_o(o,q) ≤ d_max then
      result.insert(d_o(o,q), o);
   end if
   if result.lengte ≥ k then
      d_max = result[k].key;
   end if
   /* verwijder alle elementen van result waarvoor geldt dat key > d_max */
   for i=0; i < result.lengte; ++i do
      if result[i].key > d_max then
         result.verwijder(i);
      end if
   end for
end while
/* geef alle elementen van result terug waarvoor geldt dat key ≤ d_max */
for i=0; i < result.lengte: ++i do
   if result[i].key > d_max then
      result.verwijder(i);
   end if
end for
return  result;

}
```

In algoritme 5.3 is ranking een incrementeel single-step NN-zoekalgoritme. We kunnen voor ranking bijvoorbeeld het algoritme vermeld in sectie 5.1 gebruiken. We overlopen de werking van algoritme 5.3. Het algoritme zoekt naar de k-de nearest neighbour in de verzameling objecten O en gebruikmakende van de afstandsfunctie d_o. Als het algoritme dit object heeft gevonden, dan heeft het ook alle objecten gevonden die zich dichter bij het voorbeeldobject bevinden dan de k-de nearest neighbour. Men moet dan nog, gebruikmakende van d_o, alle objecten vinden die op eenzelfde afstand van het voorbeeldobject liggen als de k-de nearest neighbour. Als men ook deze objecten heeft gevonden, dan heeft men alle objecten gevonden die het antwoord op de query vormen en dus kan het algoritme stoppen. We gaan nu iets meer in detail.
Het incrementeel k-NN-algoritme reikt de objecten volgens hun d_{filter}-afstand, van klein naar groot, aan, aan het multi-step-algoritme. In het begin wordt de lijst gevuld met de eerste k kandidaten, aangereikt door het incrementeel k-NN-algoritme. Dan wordt d_{max} gelijkgesteld aan het element met de grootste key/object-afstand. Dit is hetzelfde als in het eerste multi-step algoritme. De vraag is nu als de k-de nearest neighbour volgens de afstandsfunctie d_o in de gesorteerde lijst gaat voorkomen gedurende de loop van het algoritme. De grootste afstand volgens afstandsmaat d_o in de gesorteerde lijst is ook altijd groter of gelijk aan de afstand tot de k-de-nearest neighbour omdat de object-afstand van het k-de object van de lijst aan d_{max} wordt toegekend. Met behulp van het incrementeel zoekalgoritme en de filter-afstand proberen we zo vlug mogelijk de juiste elementen in de gesorteerde lijst te krijgen. Stel dat de sorted list nog niet het k-de-NN element volgens D_o bevat, dan is er een element met een afstand, gebruikmakende van d_o, kleiner dan de huidige d_{max}. Ook de feature-afstand van dit element zal kleiner zijn dan d_{max}. Aanegezien het incrementeel k-NN-algoritme de objecten, gebruikmakende van d_{filter}, van klein naar groot aanreikt, zullen we het k-de-NN-element volgens d_o nog tegenkomen. Het k-de-NN-element volgens d_o zal dus in de gesorteerde lijst voorkomen in de loop van het algoritme. Stel dat het k-de-NN-element in de lijst zit. We merken op dat de object-afstand van het k-de-NN-element de minimale waarde voor d_{max} is. Uiteindelijk zullen alle objecten met een kleinere object-afstand dan het k-de-NN-element ook in de gesorteerde lijst voorkomen aangezien hun feature-afstand kleiner is dan de object-afstand van het k-de-NN-element. Als dit gebeurt is, dan is d_{max} gelijk aan de object-afstand van het k-de-NN-element. Vervolgens zal de inhoud van de eerste k-1 elementen van de gesorteerde lijst niet meer veranderen. Feature-afstanden kunnen wel nog kleiner zijn dan d_{max}. Zodra de feature-afstand groter wordt dan d_{max}, dan zullen de afstanden volgens d_o ook groter zijn dan d_{max} wegens de lower bounding-eigenschap en kunnen we

dus stoppen.

5.3 Kostmodel voor NN- en range-queries

We hebben nu algoritmen gezien waarmee we (k-)NN- en range-queries verrichten. We zouden deze queries zo optimaal mogelijk willen uitvoeren. Een eerste voorwaarde is dat het algoritme zelf optimaal. Een tweede factor is het zoeken tussen de feature-vectoren. Als we gebruik maken van één multidimensionale index, kunnen we kiezen tussen het gebruik van die index of een lineaire scan. We hebben gezien dat de performantie van een index naar die van een lineaire scan kan evolueren. Het is daarom interessant om te weten hoeveel blokken men waarschijnlijk gaat raadplegen bij een NN- of range-query gebruikmakende van een index. Immers, zo kan men beslissen als het beter is om een lineaire scan te gaan doen in plaats van een index te gebruiken. Een kostmodel kan ons ook misschien een inzicht geven waarom een index niet goed functioneert.
Men heeft tijdens het deel over indexen zich misschien afgevraagd als er kostmodellen voor de indexen zijn. Deze zijn er naar mijn weten tot nu toe nog niet. De kostmodellen zijn meestal wat algemener, maar kunnen, gebruikmakend van principes gehanteerd in de index(en), toch redelijk accurate resultaten opleveren. Om zo precies mogelijke resultaten te bekomen zullen we in onderstaand kostmodel [1] rekening houden met boundary effects (zie sectie 3.1) en de fractale dimensie (zie sectie 2.2).

5.3.1 Kostmodel

We beginnen met de situatie waarbij we geen rekening houden met boundary effects en de fractale dimensie. Vooraleer we de kans op intersectie van een hyperbalk/hyperkubus bepalen, geven we eerst inzicht in hoe een D-dimensionale hyperkubus zich gedraagt.

Beschrijving D-dimensionale hyperbalk Een D-dimensionale hyperbalk heeft $2 \cdot D$ $(D-1)$-dimensionale zijkanten. We beschouwen hier het geval waarbij de zijkanten evenwijdig zijn met de assen. De hyperbalk is in elke dimensie $i \in \{1, \ldots, D\}$ begrensd door een minimale waarde m_i en een maximale waarde M_i. De intersectie van twee $(D-1)$-dimensionale zijkanten geeft telkens een $(D-2)$-dimensionaal object. Dit $(D-2)$-dimensionaal object bestaat uit $(D-3)$-, $\ldots$, 0-dimensionale objecten. Immers, twee verschillende $(D-1)$-dimensionale zijkanten hebben beide voor één dimensie een constante waarde, die de intersectie ervan ook zal bevatten. De waarden van de overige $(D-2)$ dimensies van het intersectie-object blijven variabel.

Dit $(D-2)$-dimensionaal object bestaat uit $(D-3)$-dimensionale objecten, ... Dit kan men als volgt bekijken. Men bekomt het $(D-3)$-dimensionaal object door het $(D-2)$-dimensionaal object te intersecteren met een andere $(D-1)$-dimensionale zijkant. Het $(D-2)$-dimensionale object staat per definitie loodrecht op de $(D-1)$-dimensionale zijkant. Het $(D-3)$-dimensionale object verschilt alleen van het $(D-2)$-dimensionale object doordat het voor een nieuwe dimensie een constante waarde krijgt. Men bekomt vanuit dit $(D-3)$-dimensionaal object opnieuw het $(D-2)$-dimensionaal object door de nieuwe constante waarde te vervangen door het respectievelijke continuë interval voor die dimensie. We hebben hierboven gezien dat het $(D-2)$-dimensionale object loodrecht stond op de $(D-1)$-dimensionale zijkant.

Elke $(D-1)$-dimensionale zijkant A wordt geïntersecteerd door $2 \cdot D - 1$ andere zijkanten. Dit kan men afleiden uit het feit dat voor de dimensie waar A een constante waarde voor heeft, geen andere zijkant een andere zijkant tegenhoudt die constante waarde te bereiken voor die dimensie. Alleen de parallelle zijkant zal de constante waarde niet bereiken. Vanuit dezelfde redenering (dimensie begrensd door) kan men afleiden dat bovenstaand $(D-3)$-dimensionaal object, zal geïntersecteerd worden door een $(D-1)$-dimensionale zijkant die er loodrecht op staat zodat men een $(D-4)$-dimensionaal object bekomt ...

We kunnen dus besluiten dat de intersecties van twee $(D-1)$-dimensionale zijkanten $(D-2)$-dimensionale objecten zijn, die bestaan uit $(D-3)$, $(D-4)$, ...-dimensionale objecten. Een $(D-3)$-dimensionaal object bekomt men door een $(D-2)$-dimensionaal object te intersecteren met een $(D-1)$-dimensionale zijkant waaruit het $(D-2)$-dimensionale object niet ontstaan is, ...

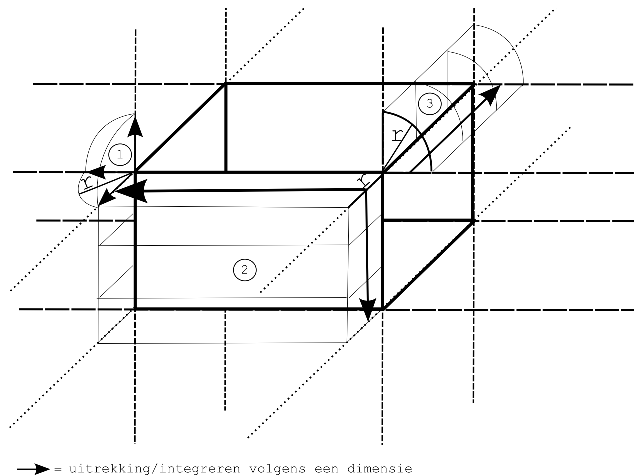
Volume Minkowski-Som hyperbalk en hyperbol Zoals vermeld in het Minkowski-Som gedeelte (zie sectie 2.1) kan men zich het volume van de Minkowski-Som van een hyperbalk en een hyperbol voorstellen door het centrum van de hyperbol door het oppervlak van de hyperbalk/hyperkubus te bewegen. Stel nu dat we de hyperkubus opsplitsen in de verschillende uitbreidingen die ontstaan door de intersectie van twee zijkanten, drie zijkanten, ... Dit zijn dus respectievelijk $(D-2)$-dimensionale objecten, $(D-3)$-dimensionale objecten, ... We hebben ook de $(D-1)$-dimensionale zijkanten zelf. Met ander woorden, de uitbreidingen zijn de objecten gevormd door de gehele uitbreiding op te splitsen volgens de vlakken die we bekomen door de $(D-1)$-dimensionale zijkanten van de hyperbalk als vlakken te beschouwen. In figuur 14 kan men dit zien voor een 3-dimenionsale hyperbalk en een 3-dimensionale hyperbol.

We bekijken hoe de Minkowski-Som/de hyperbol-uitbreiding inwerkt op deze verschillende objecten. Bij de $(D-1)$-dimensionale zijkanten splitst de zijkant de hyperbol doormidden. Het volume van de zijkant moet dan vermenigvuldigd worden met de straal van de hyperbol. Een $(D-2)$-dimensionaal object is dus een intersectie van twee $(D-1)$-dimensionale objecten die loodrecht op elkaar staan. De hyperbol-uitbreiding wordt dus bij een $(D-2)$-dimensionaal-object gesplitst volgens twee dimensies. Voor beide dimensies wordt de hyperbol in twee gesplitst aangezien de zijkanten loodrecht op elkaar staan. Op de plaatsen waar drie zijkanten intersecteren is er dus een $(D-3)$-dimensionaal object dat invloed heeft op de grootte van de uitbreiding, ... Het $(D-3)$-dimensionaal object wordt alleen aan de uiteinden geïntersecteerd met andere zijkanten waarbij dan $(D-4)$-dimensionale objecten ontstaan, ... Nu tonen we hoe het volume berekend wordt voor elk van de objecten. Voor een $(D-1)$-dimensionale zijkant hebben we het al vermeld. Hoe berekenen we nu het volume van de andere uitbreidingen? Bij een $(D-2)$-dimensionaal object zijn er voor elk punt van het object telkens twee dimensies met een constante waarde. Deze twee dimensies zijn de dimensies waar twee $(D-1)$-dimensionale zijkanten een constante waarde hebben. De constante waarden zijn een uiterste grens voor de hyperkubus in die dimensies. De andere dimensies van het $(D-2)$-dimensionale object hebben waarden die telkens reiken van de kleinste $(D-1)$-dimensionale zijkant voor die dimensie tot de grootste. We clippen de uitbreiding ook door de overige $(D-2)$ zijkanten. Immers, de uitbreiding daar wordt door de overige objecten bepaald. Hoe ver reikt nu de uitbreiding telkens? De verste afstand van de hyperkubus wordt bereikt als de straal van de hyperbol loodrecht op een zijkant staat. Bij intersecties is het de straal vanaf het intersectieobject dat de grootste uitbreiding geeft.
We hebben dus gezien dat elke uitbreiding overeenkomt met een $(D-i)$-dimensionaal object waarbij $i \in \{1, \ldots, D\}$. Dit object wordt gevormd door de intersectie van i zijkanten. Een zijkant deelt de hyperbol-uitbreiding telkens in tweeën aangezien de zijkanten loodrecht op elkaar staan. De uitbreiding wordt verder geclipt door de overige $D-i$ zijkanten aangezien daar de uitbreiding door een ander object bepaald wordt. We beschouwen voor een bepaald $(D-i)$-dimensionaal object de uitbreiding voor een punt van dat object. In i dimensies wordt de hyperbol in tweeën gesplitst en in de overige $D-i$ dimensies wordt de hyperbol helemaal geclipt. De straal van de uitbreiding is de straal r van de hyperbol. Deze uitbreiding is dus een hyperbol C in i dimensies waarvan het volume gedeeld moeten worden door 2^i. De totale uitbreiding van het $(D-i)$-dimensionaal object is dus de uitrekking van C volgens de $D-i$ dimensies. Dit kan men ook in figuur 14 waarnemen. Voor het 2-dimensionaal object gebeurt de uitrekking in één dimensie,

voor de 1-dimensionale zijkant in twee dimensies en voor het 0-dimensionaal punt in drie dimensies. In het geval van een hyperkubus waarbij de ribben een lengte a hebben, kunnen we de volume-uitbreiding voor een object A als volgt beschrijven. Veronderstel dat A een k-dimensionaal object is. A heeft dus $D - k = i$ constante waarden. A wordt dus gevormd door de intersectie van $D - k = i$ zijkanten. De volume-uitbreiding is dan:

$$a^k \cdot (\frac{1}{2^{D-k}} \cdot \frac{\sqrt{\Pi^{D-k}}}{\Gamma(\frac{D-k}{2}+1)} \cdot r^{D-k})$$

De drie mogelijke gevallen van uitbreiding in het 3-dimensionale geval, namelijk het volume van de bol wordt gehalveerd in één, twee of drie dimensies, zijn te zien in figuur 14.



Figuur 14: de drie mogelijk typen van uitbreiding in het 3-dimensionale geval

We weten nu hoe de uitbreiding van een object bepaald wordt. Nu moeten we nog weten hoeveel k-dimensionale objecten er dus zijn. k-dimensionale objecten zullen voor $D - k$ dimensies een constante waarde hebben. Er zijn hier twee mogelijke constante waarden voor elke dimensie i: m_i en M_i. m_i staat voor de kleinste waarde van de hyperbalk in dimensie i en M_i voor de grootste waarde. De waarden van de overige dimensies liggen afhankelijk van de dimensie in het interval $[m_i,M_i]$. De intervallen $[m_i,M_i]$ stellen we voor door *. Het aantal mogelijke combinaties van k plaatsen waar een $*$ staat in de D-dimensionale vector stelt dan het aantal verschillende k-dimensionale objecten voor. Dit is een combinatie. Immers, als we de k sterren zouden nummeren, dan maakt de volgorde waarin ze staan niks uit. Het stelt

telkens hetzelfde interval voor, voor een bepaalde dimensie. Voor elke combinatie moet men ook rekening houden met de plaatsen waar een constante waarde kan staan. Men heeft dus $D - k$ constante waarden. Er zijn telkens twee mogelijkheden: m_i of M_i. Dus heeft men 2^{D-k} mogelijkheden voor de constante waarden per $*$-combinatie. De volgende formule geeft dus weer hoeveel $(D - k)$-dimensionale objecten er zijn in de hyperbalk:

$$\binom{D}{k} \cdot 2^{D-k}$$

Een combinatie van k elementen uit een verzameling van D elementen wordt berekend door:

$$\binom{D}{k} = \frac{D!}{k! \cdot (D-k)!}$$

Het volume van de Minkowski-Som van de hyperkubus K met ribben van lengte a en de hyperbol Y $Vol_{K \oplus Y}$is dan;

$$Vol_{K \oplus Y} = a^D + \sum_{0 \leq k < D} \binom{D}{k} \cdot 2^{D-k} \cdot a^k \cdot \left(\frac{1}{2^k} \cdot \frac{\sqrt{\Pi^{D-k}}}{\Gamma(\frac{D-k}{2} + 1)} \cdot r^{D-k}\right) \quad (12)$$

$$= \sum_{k=0}^{D} \binom{D}{k} \cdot a^k \cdot \frac{\sqrt{\Pi^{D-k}}}{\Gamma(\frac{D-k}{2} + 1)} \quad (13)$$

We kunnen deze formule nu aanpassen voor een hyperbalk in plaats van een hyperkubus door het volume van een hyperkubus a^k te vervangen door het volume van de respectievelijke hyperbalk. Voor elk object moet dus het volume bepaald worden. Het is echter kostelijk om deze berekening uit te voeren. Als benadering hiervoor gebruiken we een hyperkubus met hetzelfde volume als de hyperbalk. Dit geeft een fout, maar uit testen blijkt dat deze fout klein genoeg is om weinig effect te hebben op het resultaat.

Range-query We zullen nu een formule $A[hyperkubus](r)$ ontwikkelen die het verwachte aantal bounding regio's geeft die we zullen moeten onderzoeken om de range-query te beantwoorden. Hiervoor moeten we de kans $P[hyperkubus](r)$ weten dat een hyperkubus geïntersecteerd wordt. Gegeven het gemiddeld aantal feature-vectoren C_{eff} per bounding regio, kunnen we het gemiddeld aantal geïntersecteerde bounding regio's als volgt berekenen:

$$A[hyperkubus](r) = \frac{N}{C_{eff}} \cdot P[hyperkubus](r)$$

De bounding regio is een D-dimensionale hyperkubus en de range-query wordt voorgesteld door een D-dimensionale hyperbol met straal r. Als we veronderstellen dat de data- en query-distributie voldoet aan de eisen beschreven in sectie 2.2, dan is de formule die de kans geeft dat een hyperkubus K geïntersecteerd wordt door een hyperbol Y gelijk aan $Vol_{K \oplus Y}$. De variabele die we nog niet kunnen invullen is de lengte van een ribbe van de hyperkubus, namelijk a. We weten dat het gemiddelde volume van een bounding regio gelijk is aan N/C_{eff}. Hieruit volgt dat de gemiddelde lengte van de zijkant van een bounding regio $\sqrt[D]{C_{eff}/N}$ is. Een bounding regio die deze lengte voor de ribben heeft, is echter geen minimale bounding regio (MBR) die we in de indexen gebruiken. Immers, de gemiddelde afstand per dimensie tussen twee punten is $1/C_{eff} \cdot \sqrt[D]{C_{eff}/N}$. Dus de gemiddelde lengte van elke ribbe van een MBR is

$$a = (1 - \frac{1}{C_{eff}}) \cdot \sqrt[D]{\frac{C_{eff}}{N}}.$$

Dit verschil in lengte noemt men het *MBR-effect*.
Als N en C_{eff} een vaste waarde hebben en we D naar oneindig laten gaan, zien we dat het MBR-effect groot is aangezien $0 \leq C_{eff}/N \leq 1$.
Als N en D een vaste waarde hebben en we laten C_{eff} stijgen, zal het MBR-effect ook steeds groter worden.
Als D en C_{eff} een vaste waarde hebben en we laten N stijgen, zal het MBR-effect steeds meer invloed hebben op het gemiddelde volume van een bounding regio.
Hieruit volgt dat

$$P[\mathit{hyperkubus}](r) = \sum_{k=0}^{D} \binom{D}{k} \cdot ((1 - \frac{1}{C_{eff}}) \cdot \sqrt[D]{\frac{C_{eff}}{N}})^k \cdot \frac{\sqrt{\Pi^{D-k}} \cdot r^{D-k}}{\Gamma(\frac{D-k}{2} + 1)}$$

Bijgevolg is

$$A[\mathit{hyperkubus}](r) = \frac{N}{C_{eff}} \cdot P[\mathit{hyperkubus}](r)$$

NN-query Gebruikmakend van de formule voor range-queries, zullen we een formule opstellen die het gemiddeld aantal intersecties met bounding regio's geeft voor een NN-query. Formules voor de verwachte NN-afstand werden al uitgewerkt in sectie 3.2.

Met deze gegevens kunnen we het gemiddeld aantal intersecties als volgt bepalen:

$$A[\mathit{hyperkubus}]_{NN} = \int_0^\infty A[\mathit{hyperkubus}](r) \cdot P_{E[NN]=r}(r)$$

5.3.2 Kostmodel met boundary effects

De vorige formules passen we nu aan om rekening te houden met boundary effects en de fractale dimensie. We beginnen eerst met boundary effects.

Volume Minkowski-Som hyperbalk en hyperbol We bepalen eerst hoe we het volume van de Minkowski-Som, geclipt aan de randen van de dataruimte R, berekenen. We nemen aan dat de hyperkubus zelf volledig binnen R ligt. We moeten dus het volume van de uitbreiding clippen. In de formule komt dit neer op het clippen van het volume van de hyperbol. Het clippen van het volume van een hyperbol doen we op dezelfde manier als in sectie 3.2. Namelijk, we stellen een tabel $Vol_{geclipt,o}$ op, op dezelfde wijze als de tabel $Vol_{geclipt}$ van sectie 3.2, maar we nemen enkel hyperbollen waarvan het centrum de oorsprong van R is, in plaats van hyperbollen met een willekeurige locatie. Gegeven een dimensie D en een straal r, kan men dus met de tabel $Vol_{geclipt,o}$ het geclipte volume van een hyperbol met als centrum de oorsprong van R, bepalen. Het object waardoor de hyperbol geclipt wordt is ook niet R. Wat het object is dat clipt, bepalen we hierna. We zullen het relatieve volume van de hyperbol bepalen door zowel de hyperbol als het clipping-object gelijkmatig te vergroten totdat het clipping-object overeenkomt met R. Het relatieve volume blijft hierdoor hetzelfde. Het volume van de hyperbol en het clipping-object is vergroot met een eenzelfde vergrotingsfactor. Vanuit deze situatie kunnen we het volume van de hyperbol opzoeken in de tabel $Vol_{geclipt,o}$. Dit volume vermenigvuldigen we dan met het inverse van de vergrotingsfactor van het volume van de hyperbol om het werkelijke volume te bekomen.

Range-query We hebben dus nu een methode om het Minkowksi-Som-volume te clippen. Om het te clippen weten we best iets over de posities van de bounding regio's zodat we weten hoe we ongeveer moeten clippen. We moeten ook nog de variabele a in de Minkowski-Som-formule bepalen die rekening houdt met het feit dat de dataruimte begrensd is. We nemen hiervoor aan dat D' van de D dimensies in tweeën gesplitst worden. De overige $D - D'$ dimensies worden niet gesplitst. Deze aanname is mogelijk omdat we voor een hoge D en $N < 2^D$ anders (veel) lege bounding regio's hebben. Wegens het MBR-effect geldt dus dat in de D' dimensies waar gesplitst wordt $a_{split} = 0.5 \cdot (1 - 1/C_{eff})$. In de $D - D'$ dimensies waar niet geplitst wordt geldt dan $a_{unsplit} = 1 - 1/C_{eff}$. Als een dimensie niet gesplitst is, dan is er wegens het MBR-effect aan beide uiteinden van die dimensie

telkens a_{unKant} plaats, waarbij

$$\begin{aligned} a_{unKant} &= \frac{1 - (1 - \frac{1}{C_{eff}}))}{2} \\ &= \frac{1}{2} \cdot \frac{1}{C_{eff}} \end{aligned}$$

Als een dimensie gesplitst is, dan is er tussen beide bounding regio's een afstand $a_{middenLeeg}$ die gelijk is aan de afstand tussen twee punten, dus

$$a_{middenLeeg} = 0.5 \cdot \frac{1}{C_{eff}}$$

Dan, als een dimensie gesplitst is, dan is er aan beide uiteinden van die dimensie telkens a_{spKant} plaats, waarbij

$$\begin{aligned} a_{spKant} &= \frac{1 - 2 \cdot a_{split} - a_{middenLeeg}}{2} \\ &= \frac{\frac{1}{C_{eff}} - 0.5 \cdot \frac{1}{C_{eff}}}{2} \\ &= \frac{1}{4 \cdot C_{eff}} \end{aligned}$$

Laten we nu het volume van de Minkowski-Som van een bounding regio beschouwen. Om de berekening iets te vereenvoudigen, veronderstellen we een aantal dingen. We nemen aan dat de plaatsen waar de afstand tussen de bounding regio en R klein is, namelijk a_{unKant} en a_{spKant}, volledig opgevuld wordt door de Minkowski-Som. Dit kunnen we aannemen als een geldige aanname als de straal een tamelijk grote kans heeft om zo groot te zijn als deze waarden. Als we de situatie bekijken voor $C_{eff} = 1$, dan is $a_{unKant} = 1/2$ en $a_{spKant} = 1/4$. We hebben gezien dat voor hoge dimensies de gemiddelde NN-afstand al vlug 0.5 is, dus in hoog-dimensionale ruimten is dit een mogelijke aanname. Het betekent wel dat dit model niet zo correct zal zijn voor laag- en medium-dimensionale ruimten. Niet gesplitste dimensies hebben een lengte één en de uitbreiding volgens deze dimensies hoeft dus niet opgenomen te worden in de Minkowksi-Som. Immers, het volume van de tot dan berekende Minkowski-Som wordt dan vermenigvuldigd met één. De enige plaats waar dan nog geclipt moet worden is in de dimensies die gesplitst zijn, in de richting van de andere bounding regio. We gebruiken hiervoor bovenstaande clippingmethode. We moeten hiervoor wel nog het clipping-object bepalen. De hyperbol moet dus in de gesplitste dimensies geclipt worden. Vanaf de

bounding regio tot de rand van R is er nog a_{empty} plaats, waarbij

$$\begin{aligned} a_{empty} &= a_{middenLeeg} + a_{split} + a_{spKant} \\ &= \frac{1}{2} + \frac{1}{4 \cdot C_{eff}} \end{aligned}$$

In de gesplitste dimensies mag de straal van de hyperbol dus niet groter zijn dan a_{empty}. De hyperbol manifesteert zich niet in niet-gesplitste dimensies. Het relatieve volume van de geclipte hyperbol is $Vol_{geclipt,o}(k, r/a_{empty})$. Hierbij wordt dus de straal vergroot zodat het clipping-object R wordt. Hierin is k het aantal dimensies waarin gesplitst wordt. Het clipping-object is dus een k-dimensionale hyperkubus met ribben van lengte a_{empty}. We vermenigvuldigen het volume met $(a_{empty})^k$ om het benaderende volume op echte schaal te verkrijgen.
De formule voor de gemiddelde kans dat een bounding regio geïntersecteerd wordt door een range query met straal r, is dan

$$P[hyperkubus](r) = \sum_{0 \leq k \leq D'} \binom{D'}{k} \cdot \left(\frac{1}{2} - \frac{1}{4 \cdot C_{eff}}\right)^{D'-k} \cdot \left(\frac{1}{2} + \frac{1}{4 \cdot C_{eff}}\right)^{k} \cdot Vol_{geclipt,o}\left(k, \frac{r}{a_{empty}}\right)$$

Nu moeten we het gemiddeld aantal bounding regio's die geïntersecteerd worden berekenen. Als er N/C_{eff} bounding regio's zijn en als dit getal een macht van twee is, dan zijn er $\log_2(N/C_{eff}) = D'$ splitsingen voor alle bounding regio's. Als het aantal bounding regio's niet gelijk is aan een macht van twee, dan zijn er $N_{D'}$ bounding regio's met $\lfloor \log_2 (N/C_{eff}) \rfloor$ splitsingen en $N_{D'+1}$ bounding regio's met $\lceil \log_2 (N/C_{eff}) \rceil$ splitsingen. Als we $\lfloor \log_2 (N/C_{eff}) \rfloor$ splitsingen hebben, dan hebben we $2^{\lfloor \log_2 \left(\frac{N}{C_{eff}}\right) \rfloor}$ bounding regio's. We moeten er in totaal N/C_{eff} hebben. Voor elke bounding regio die we nu nog eens splitsen, komen er twee in de plaats. Door een splitsing verhogen we dus het totaal aantal bounding regio's met één. Hoeveel van deze splitsingen moeten we doen om N/C_{eff} bounding regio's te krijgen? Dit is dus

$$\frac{N}{C_{eff}} - 2^{\left\lfloor \log_2 \left(\frac{N}{C_{eff}}\right) \right\rfloor}$$

Dit betekent dat het aantal bounding regio's die $D' + 1 = \left\lceil \log_2 \left(\frac{N}{C_{eff}}\right) \right\rceil$ splitsingen hebben, gelijk is aan

$$N_{D'+1} = 2 \cdot \left(\frac{N}{C_{eff}} - 2^{\left\lfloor \log_2 \left(\frac{N}{C_{eff}}\right) \right\rfloor}\right)$$

Bijgevolg is het aantal bounding regio's die $D' = \left\lfloor \log_2 \left(\frac{N}{C_{\mathit{eff}}} \right) \right\rfloor$ splitsingen hebben, gelijk aan

$$N_{D'} = \frac{N}{C_{\mathit{eff}}} - N_{D'+1}$$

Dus is het gemiddeld aantal bounding regio's die geïntersecteerd worden

$$\begin{aligned} A[\mathit{hyperkubus}](r) = N_{D'} \cdot P[\mathit{hyperkubus}] \left(\left\lfloor \log_2 \left(\frac{N}{C_{\mathit{eff}}} \right) \right\rfloor, r \right) + \\ N_{D'+1} \cdot P[\mathit{hyperkubus}] \left(\left\lceil \log_2 \left(\frac{N}{C_{\mathit{eff}}} \right) \right\rceil \right) \end{aligned}$$

NN-query Nu moeten we nog een formule bekomen voor NN-queries. In sectie 3.2 hebben we al gezien hoe we de probabiliteit-dichtheidsfunctie kunnen berekenen waarbij rekening wordt gehouden met boundary effects. De formule voor het gemiddeld aantal geïntersecteerde bounding regio's bij een NN-query, wordt dan

$$\begin{aligned} A[\mathit{hyperkubus}]_{NN} &= \int_0^\infty A[\mathit{hyperkubus}](r) \cdot P_{E[NN]=r}(r) \\ &= N \cdot \sum_{i=1}^{r_{aantal}} A[\mathit{hyperkubus}] \left(\frac{i \cdot \sqrt{D}}{r_{aantal}} \right) \cdot (1 - \mathit{Vol}_{geclipt,o}[i])^{N-1} \cdot \\ &(\mathit{Vol}_{geclipt,o}[i] - \mathit{Vol}_{geclipt}[i-1]) \end{aligned}$$

5.3.3 Kostmodel met boundary effects en fractale dimensie

Range-query We zullen weer eerst een formule opstellen voor range-queries. Zoals vermeld in sectie 2.2 moeten we bij range-queries veronderstellen dat ρ_F constant is voor het gedeelte van de dataruimte waar zich punten bevinden. Als we deze formules gebruiken voor de NN-query-formules, dan kan deze voorwaarde iets zwakker gemaakt worden zoals vermeld in sectie 2.2.

We berekenen eerst de verwachting dat een bounding regio geïntersecteerd wordt door een range-query rekening houdend met de fractale dimensie D_F. Hiervoor moeten we de lengte a bepalen van de ribbe van een hyperkubus, rekening houdend met de fractale dimensie. Dit kunnen we doen gebruikma-

kend van de powerlaw voor de fractale dimensie.

$$N_{hyperkubus} = \rho_F \cdot Vol_K^{\left(\frac{D_F}{D}\right)}, \text{ dus}$$

$$C_{eff} = \rho_F \cdot \left(\frac{a}{1 - \frac{1}{C_{eff}}} \right)^{D_F}$$

$$\Updownarrow$$

$$a = \sqrt[D_F]{\frac{C_{eff}}{\rho_F}} \cdot \left(1 - \frac{1}{C_{eff}}\right)$$

De lengte van de bounding regio waarbij rekening wordt gehouden met het MBR-effect is gelijk aan a. ρ_F (gemiddeld volume per punt) houdt echter geen rekening met het MBR-effect, vandaar dat we delen door $1 - 1/C_{eff}$.

Een tweede aspect waarbij we rekening moeten houden met de fractale dimensie, is het aantal splitsingen in het geval van een hoge fractale dimensie. We maken hier een onderscheid tussen een dataset met een lage/medium fractale dimensie en een met een hoge fractale dimensie omdat bij een laagdimensionale dataset we geen rekening hoeven te houden met boundary effects. Men neemt de volgende heuristiek om te bepalen vanaf welke dimensie we over een hoog-dimensionale dataset spreken:

$$D_F \geq D' = \left\lceil \log_2 \left(\frac{N}{C_{eff}} \right) \right\rceil$$

Als de fractale dimensie laag is, dan splitsen we in alle dimensies, maar we hebben gezien dat als D_F groot is, we enkel in D' dimensies splitsen, waarbij $D' < D$. Een aantal dimensies kunnen echter afhankelijk zijn van een deel van deze D' dimensies. Als we één van de D' dimensies splitsen, dan wordt het aantal punten dat zich aan elke kant van de splitsing bevindt, ook beïnvloedt door de afhankelijke dimensies. Het doel van op te splitsen in D' in plaats van D dimensies was dat er geen bounding regio's zouden zijn met zeer weining datapunten. Als de waarden in één dimensie afhankelijk zijn van de waarden in andere dimensies en indien we geen rekening houden met deze afhankelijkheid, dan kan het zijn dat er aan de ene kant van de splitsing zeer weinig punten zijn en aan de andere kant zeer veel. Omwille van de afhankelijkheid kunnen de lengten van de bounding regio's aan beide kanten van de opsplitsing in de afhankelijke dimensie ook kleiner worden dan één. Om ervoor te zorgen dat het volume van de hyperkubus een representatie is voor het aantal punten die erin liggen en omdat de hyperkubussen MBR's zijn, beschouwen we een afhankelijke dimensie als gesplitst. We nemen aan dat deze op dezelfde manier worden gesplitst als de D' dimensies die worden

gesplitst, waardoor we dus meer dan D' splitsingen krijgen. Zij het totaal aantal splitsingen D''. Het percentage dimensies die onafhankelijk zijn van andere dimensies is gelijk aan D_F/D. We kunnen het volgende opschrijven

$$\frac{D_F}{D} \cdot D'' = D'$$
$$\Downarrow$$
$$D'' = \frac{D' \cdot D}{D_F}$$

Merk op dat als $D' > D_F \Rightarrow D'' > D$.
De twee bovenstaande aanpassingen zijn gevolgen van de fractale dimensie op het vlak van de datapunten. We kunnen ze verwerken in het opbouwen van een formule voor de range-query waarbij we mogen aannemen dat voor datapunten dimensies kunnen afhangen van andere dimensies. We nemen voorlopig nog aan dat voor de query-punten alle dimensies onafhankelijk van elkaar zijn. In het laag-dimensionale geval bekomen we dan dan de volgende formule voor een range-query:

$$P[hyperkubus](r) = \sum_{k=0}^{D} \binom{D}{k} \cdot \left(\left(1 - \frac{1}{C_{eff}}\right) \cdot \sqrt[D_F]{\frac{C_{\mathit{eff}}}{\rho_F}} \right)^k \cdot \frac{\sqrt{\Pi^{D-k}} \cdot r^{D-k}}{\Gamma(\frac{D-k}{2}+1)}$$

In de hoog-dimensionale situatie geldt:

$$P[hyperkubus](r) = \sum_{0 \le k \le D''} \binom{D''}{k} \cdot \left(\frac{1}{2} - \frac{1}{4 \cdot C_{\mathit{eff}}}\right)^k \cdot \left(\frac{1}{2} + \frac{1}{4 \cdot C_{\mathit{eff}}}\right)^{D''-k} \cdot Vol_{geclipt,o}\left(D'', \frac{r}{a_{empty}}\right)$$

De volgende stap is toe te laten dat ook voor de query-punten dimensies afhankelijk kunnen zijn van andere dimensies. De query-punten hebben betrekking op de Minkowski-Som, dus we moeten bepalen welke aanpassingen we hier eventueel moeten doen. We moeten dus bepalen hoeveel punten werkelijk in de Minkowksi-Som liggen. We schakelen hiervoor over van de powerlaw zonder de fractale dimensies naar de powerlaw met de fractale dimensie. Dit wil dus zeggen dat we de fractal point density in plaats van N/C_{eff} en de fractale dimensie D_F in plaats van D gebruiken in de formules. Als we verder aannemen dat de data- en query-punten van dezelfde distributie genomen worden en dezelfde fractale dimensie geven, dan wordt de

formule voor de range-query in de hoog-dimensionale situatie:

$$P[hyperkubus](r) = \left(\sum_{0 \leq k \leq D''} \binom{D''}{k} \cdot \left(\frac{1}{2} - \frac{1}{4 \cdot C_{eff}} \right)^{k} \cdot \left(\frac{1}{2} + \frac{1}{4 \cdot C_{eff}} \right)^{D''-k} \cdot Vol_{Y(r)} \left(D'', \frac{r}{a_{empty}} \right) \right)^{\frac{D_F}{D}}$$

Om het verwachte aantal geïntersecteerde bounding regio's te bepalen, vermenigvuldigen we bovenstaande formules met N/C_{eff} aangezien het werkelijke aantal punten N is en elk bounding region gemiddeld C_{eff} feature-vectoren bevat.

NN-query In sectie 3.2 hebben we gezien hoe we de NN-afstand kunnen berekenen, rekening houdend met de fractale dimensie en in het hoog-dimensionaal geval ook met boundary-effects. De formule voor de range-query en de probabiliteit $P_{E[NN]=r}(r)$ kunnen we dan combineren om het verwachte aantal geïntersecteerde bounding regio's te bekomen voor een NN-query, rekening houdend met boundary effects en de fractale dimensie. Ook hier maken we een onderscheid tussen het laag/medium-dimensionaal geval en het hoog-dimensionaal geval aangezien boundary effects pas een voelbare invloed beginnen uit te oefenen in het hoog-dimensionaal geval. De formules kunnen op analoge manier als in bovenstaande secties geconstrueerd worden, namelijk:

$$A[hyperkubus]_{NN} = \frac{N}{C_{eff}} \cdot \int_{0}^{\infty} \left(A[hyperkubus](r) \cdot P_{E[NN]=r}(r) \right) dr$$

5.4 Samenvatting

We hebben k-NN- en range-query-algoritmen besproken die optimaal zijn in het aantal bounding regio's die ze bezoeken. Dit wil dus zeggen dat er niet meer bounding regio's bezocht worden dan de bounding regio's die geïntersecteerd worden door de hyperbol bepaald door de query. Bij een k-NN-query is de straal van de hyperbol de afstand van het query-punt tot de k-de nearest neighbour.
Om meer inzicht te krijgen in NN- en range-queries hebben we ervoor een kostmodel opgesteld dat we ook kunnen gebruiken om te bepalen of we beter een lineaire scan doen of niet. We moeten wel telkens een onderscheid maken tussen een hoog-dimensionale of laag/medium-dimensionale ruimte (De

dimensie die we hiervoor gebruiken is de fractale dimensie.). Dit onderscheid is nodig om te weten of we rekening moeten houden met boundary-effects in het kostmodel, respectievelijk er geen rekening mee moeten houden. Het kostmodel bevat ook redelijk wat integralen die kostelijk zijn om te berekenen. Echter, door te werken met discrete intervallen en voorberekende tabellen met volumes wordt het wel mogelijk om in de praktijk deze methode te gebruiken. De tabellen hoeven maar één keer aangemaakt worden. Er zijn echter enkele problemen met dit kostmodel:

- De tabellen bevatten slechts voor een beperkt aantal dimensies waarden.
- Om de tabellen te raadplegen moet men eventueel de data van het opslagmedium halen.
- Het model werkt in het discrete domein bij het bepalen van de integralen

We zouden dus liever een kostmodel hebben dat we vlug kunnen berekenen (geen integralen , . . .) en dat accuraat is. Een kostmodel dat we vlug kunnen berekenen noemen we een gesloten model. In [42] beschrijft men een gesloten kostmodel voor laag/medium-dimensionale ruimten. Er bestaat ook een gesloten kostmodel voor hoog-dimensionale ruimten [30], maar dit kostmodel geldt enkel als het query-punt op de diagonaal ligt van de dataruimte.

6 Concurrency

6.1 Motivatie en inleidende theorie

Concurrency is het uitvoeren van meerdere transacties op eenzelfde dataset/index op eenzelfde tijdstip. Op deze manier kan een set van transacties misschien sneller uitgevoerd worden. Dit kan natuurlijk voor een aantal problemen zorgen, zoals een transactie die een knoop aanpast en een andere transactie die tegelijkertijd de informatie van de knoop zit te lezen. Men hanteert hier het serialiseerbaarheidsprincipe om alles consistent te laten verlopen. Voor een transactie kan men een stappenplan opschrijven hoe de query uitgevoerd wordt. Het stappenplan van een transacties bestaat dus uit de lees- en schrijfoperaties waaruit de transactie is opgebouwd. Men kan de stappenplannen van meerdere queries samenvoegen tot één stappenplan. Een stappenplan is serialiseerbaar als het uiteindelijke resultaat hetzelfde is als het seriële stappenplan. In het seriële stappenplan voert men alle acties van een transactie A uit voor alle acties van een transactie B als een actie

van A voorafgaat aan alle acties van B. Om serialiseerbaarheid te bekomen, kan men locks gebruiken in een database om te verhinderen dat een transactie een database-element leest of schrijft. Er is echter nog een probleem in dit geval. We kunnen geen lock nemen op een element dat nog ingevoegd moet worden. We noemen dit het *phantom*-probleem. We geven hiervan een voorbeeld: Zij T1 een transactie die een lees-operatie uitvoert op objecten die voldoen aan een conditie C1. Zij T2 een transactie die uitgevoerd wordt terwijl T1 bezig is en die nieuwe elementen wil toevoegen die aan de conditie C1 voldoen. T2 eindigt voor T1. Het kan dus zijn dat T1 de objecten die T2 toevoegt, opneemt in zijn resultaat. Men maakt onderscheidt tussen twee soorten locks ([20]: sectie 3.1.1). Een lock waarvoor de gebruiker van de lock garandeert dat deze geen deadlock kan veroorzaken noemt men een *latch*. Een latch wordt gedurende een korte tijd gebruikt en kost niet veel. Een lock die een deadlock kan veroorzaken noemt men een *lock*. Een *lock* wordt bijgehouden door een lock-manager om deadlocks vast te stellen, terwijl de latches niet door de lock-manager worden bijgehouden.

Er bestaan twee groepen van lockingmethoden: *granular locking* in indexen en *predicate locking*.

Bij *granular locking* worden er bepaalde typen locks geplaatst op subbomen of knopen. In de context van multidimensionale ruimten betekent dit dat er een lock wordt geplaatst op een deel van de ruimte. Als een transactie een lock plaatst, dan is dit een teken dat de transactie iets wilt gaan doen in deze ruimte. Het kan echter zijn dat de transactie niet van plan is om iets te doen met heel de ruimte. Een deel van de ruimte wordt dus overbodig gelocked. Er is dus een foutenmarge.

Bij *predicate locking* worden de query-predicaten van de transacties onderzocht om na te gaan of deze kunnen worden uitgevoerd. Bij deze methode is er dus geen foutenmarge, wat dus voor meer concurrency kan zorgen. In het laag-dimensionale geval kiest men veelal voor granular locking omdat dit minder overhead met zich meebrengt.

6.2 Problemen en mogelijke oplossingen

We bespreken nu een aantal problemen in de context van multimediadatabases:

- In [4] stelt men een methode gebaseerd op granular locking voor om het phantom-probleem op te lossen in multidimensionale indexen. In de experimenten vergelijkt men de performantie ook met een methode gebaseerd op predicate locking. Men test tot en met vijf dimensies en merkt dat in de 5-dimensionale ruimte de predicate-locking-methode

de granular locking-methode benadert en zelfs beter begint te worden vanaf een bepaald aantal transacties die tegelijkertijd uitgevoerd worden(≈ 20). Dit is ook het geval in lager-dimensionale ruimten, maar dan is het aantal transacties die tegelijkertijd uitgevoerd worden (de multiprogramming level) hoger vooraleer predicate locking beter wordt dan granular locking. Men zou de evolutie van de foutenmarge van granular locking kunnen bekijken om hier misschien meer inzicht over te krijgen. Men stelt voor om een methode te bekijken/ontwikkelen die zowel granular locking als predicate locking gebruikt.

- In hoog-dimensionale ruimten komt het feit dat men best zo weinig mogelijk locks plaatst, meer tot uiting dan in laag- en medium-dimensionale ruimten, omwille van de curse of dimensionality. Immers, hoe hoger de dimensionaliteit, hoe meer kans er is dat een bounding regio bezocht wordt door een NN-query, ... Als op elke bounding regio een lock wordt geplaatst, dan stijgt het aantal locks naarmate de dimensionaliteit stijgt.

- In multimediadatabases kunnen transacties lang duren. Een voorbeeld hiervan is de bewerking van een groot aantal video-frames.

In [27] en [6] concentreert men zich op het tweede punt, namelijk om zo weinig mogelijk locks te plaatsen. Beide methoden zorgen voor de consistentie van de huidige data en dienen niet om het phantom-probleem op te lossen. In [27] probeert men het zoeken doorheen de index zo weinig mogelijk te blokkeren. Men beschrijft hiervoor een methode waardoor een MBR-update het zoeken niet kan blokkeren en men vermeldt ook hoe men het locken tijdens een splitsproces zo kort mogelijk kan houden. In [6] probeert men ook het zoeken zo weinig mogelijk te laten blokkeren. Hier beschrijft men ook een algoritme voor forced reinsert-operaties. Het feit dat we het zoeken zo snel mogelijk willen laten verlopen, is ook een opmerking die we maakten in het hoofdstuk over indexen. In beide papers beperken de testen zich tot maximaal 10-dimensionale feature-vectors.

7 Opslag

Bij het opslaan van data zijn er een aantal elementen die we nastreven:

- zo weinig mogelijk capaciteit verloren laten gaan

- zo snel mogelijk iets plaatsen op een opslagmedium

- zo snel mogelijk data halen van een opslagmedium

Bij het gebruik van een index, gaat er een deel van de opslagcapaciteit verloren. De hoeveelheid hangt af van de gebruikte index. Bijvoorbeeld, de VA-file neemt 20–25 procent opslagcapaciteit in ten opzichte van de werkelijke data. De hoofdbedoeling van dit deel is aan te tonen dat ook de manier waarop men iets opslaat belangrijk is bij het optimaliseren van queries en in deze formules speelt in deze context van feature-vectoren ook de dimensionaliteit een rol.

7.1 Dynamische blokgrootte

Naar mijn weten werd tot nu toe meestal een vaste blokgrootte in databases gekozen. We kunnen deze restrictie echter laten vallen. Uit het volgende blijkt dat een optimale blokgrootte van een aantal dynamische factoren afhangt [5]. Met optimaal wordt er een zo laag mogelijke kost bij het queryen bedoeld. De kost is hier hoe lang het waarschijnlijk duurt om een blok data op te vragen van de harde schijf. De volgende factoren spelen een rol bij het opvragen van een blok data:

- t_{zoeken}: de tijd voor de leeskop om een willekeurige plaats te bereiken
- de tijd die het duurt om de werkelijke data over te brengen naar het geheugen
- de kans dat de blok geraadpleegd wordt

We kunnen de kost $kost_{blok}$ om de multidimensionale punten van een indexblok op te vragen als volgt berekenen:

$$kost_{blok} = (t_{zoeken} + C \cdot t_{punt}) \cdot X$$

In bovenstaande formule is C het aantal punten in de blok, t_{punt} = (de tijd om een byte over te brengen) $\cdot$ *sizeof*(*punt*) en X is de kans dat de blok geraadpleegd wordt. Uit bovenstaande formule kennen we de waarden van alle variabelen behalve X. X is dus de waarde teruggegeven door een kostmodel. Er zijn echter verschillende kostmodellen voor verschillende typen queries. In de originele paper kiest men om het kostmodel X voor de NN-query met de maximum-afstandsmaat in plaats van de Euclidische afstandsmaat te gebruiken om zo de kost voor de berekening zo laag mogelijk te houden. De auteurs merken op dat in de praktijk de optimale grootte van een blok weinig verschilt als men de Euclidische afstandsmaat of de k-NN-query gebruikt. Merk

op dat aangezien we een optimale blokgrootte proberen te vinden voor bepaalde blokken, we meer specifieke info hebben dan in het geval van algemene kostmodellen.
We zullen het kostmodel voor de NN-query met de maximum-afstandsmaat uitwerken. We zullen rekening houden met de distributie van de punten en boundary-effects. We maken dus gebruik van de fractale dimensie, respectievelijk het clippen aan de randen van de dataruimte wat hier de bounding regio kan zijn die alles omvat. We bepalen eerst de fractale dimensie van de dataset, D_F. We bepalen dan de verwachte NN-afstand $E[NN]$ binnen het blok met behulp van de powerlaw van de fractale dimensie:

$$1 = \rho_F \cdot Vol_{Y(r)}^{\frac{D_F}{D}} \text{ en } Vol_{Y(r)} = (2 \cdot r)^D$$

$$\Updownarrow$$

$$E[NN] = r = \frac{1}{2 \cdot \sqrt[D_F]{\frac{1}{\rho_F}}}, \text{ waarbij}$$

$$\rho_F = \frac{C}{Vol_{MBR}^{\left(\frac{D_F}{D}\right)}}$$

Dit is een ruwe benadering van het exact statistisch model voor de $E[NN]$-afstand, dat zich in sectie 3.2 bevindt. Met behulp van de powerlaw kunnen we bepalen hoeveel punten zich in een bepaald volume bevinden. We combineren dit dan met de formule voor het volume van een hyperbol en stellen het aantal punten die in de hyperbol liggen gelijk aan één. Hieruit bepalen we dan de straal van de hyperbol. Waarom is dit niet statistisch correct? We berekenen de verwachte lengte van de straal aan de hand van het verwachte aantal punten in de hyperbol. Echter, de operatie van het bouwen van een verwachting is niet omkeerbaar. Met andere woorden, men houdt geen rekening met het feit dat de nearest-neighbour ,zelfs onder onafhankelijkheid van de dimensies en uniformiteit, van plaats tot plaats kan verschillen. Aangezien men onder het bepalen van de verwachting, het nemen van het gemiddelde verstaat, kan men dus niet de oude waarden uit het gemiddelde verkrijgen.
We specifiëren een MBR door de diagonaal met als eindpunten de vector met de kleinste waarden $(lb_0, \ldots, lb_{D-1})$ en de vector met de hoogste waarden $(ub_0, \ldots, ub_{D-1})$. De Minkowksi-Som zonder rekening te houden met het clippen is dan:

$$Vol_{MBR,2E[NN]} = \prod_{0 \le i < D} (ub_i - lb_i + 2 \cdot E[NN])$$

We kunnen de query echter ook clippen aan de randen van de bounding regio OBR die alle bounding regio's omvat. Zij ubO en lbO de eindpunten van de

diameter van de omvattende bounding regio met de hoogste, respectievelijk de kleinste waarden voor elke dimensie. Dan is de Minkowski-Som rekening houdend met het clippen:

$$Vol_{MBR,2E[NN],OBR} = \prod_{0 \leq i < D} (min\{ub_i + E[NN], ubO_i\} - max\{lb_i - E[NN], lbO_i\})$$

Als we veronderstellen dat de query-distributie, de datadistributie volgt, dan krijgen we als kans dat de MBR geïntersecteerd wordt:

$$X = \frac{\rho_F}{N} \cdot Vol_{MBR,2E[NN],OBR}^{(\frac{D_F}{D})}$$

Dit is dus het gewone volume, rekening gehouden met boundary-effects en waarop de powerlaw werd toegepast. Men krijgt het verwachte aantal punten dat zich binnen de Minkowski-Som bevindt en dat aantal deelt men door het totaal aantal punten in de ruimte, N.
We zouden graag hebben dat de kost om een blok te benaderen zo laag mogelijk was. We zoeken hiervoor naar het minimum van de functie $kost_{blok}$. Bij het zoeken naar een minimum is het belangrijk dat er maar één minimum is, met andere woorden, het lokale minimum is het globale minimum. We moeten hiervoor bewijzen dat de afgeleide van de kostfunctie monotoon stijgend is. Dit wil dus zeggen dat de tweede afgeleide positief moet zijn. Men heeft dit aangetoond voor een vereenvoudigde versie van de kostfunctie. Namelijk, het kostmodel voor X is gebaseerd op de aanname van uniformiteit en onafhankelijkheid. Dit wil zeggen dat men de Euclidische dimensie heeft gebruikt in plaats van de fractale dimensie. Men heeft ook geen rekening gehouden met boundary effects. De vereenvoudigde versie van de kostfunctie $kost_{blok}$ waarvoor men dit heeft aangetoond is

$$(\sqrt[D]{\frac{1}{C}} + 1)^D \cdot (t_{zoeken} + C \cdot t_{punt})$$

De kost van een blok kan veranderen als een feature-vector wordt toegevoegd of verwijderd. Wij bekijken hier lokale optimalisatie, namelijk als in een boundary regio een punt wordt verwijderd of toegevoegd. Het is ook niet nuttig om bij elke toevoeging of verwijdering het kostmodel te evalueren. Dit zou immers kostelijk zijn. We kunnen echter proberen te schatten wanneer het nuttig zou zijn om het kostmodel te evalueren. Dit kan onder andere door de evaluatie te doen als een bounding regio een aantal keer is aangepast. Men zou het ook kunnen combineren met index-eigenschappen. Bijvoorbeeld, als elke bounding regio minstens m vectoren moet hebben, kan men het model gaan evalueren als het aantal kort bij m komt. We nemen aan

dat de bounding regio's disjunct zijn. We kunnen dus een afweging maken tussen twee kleinere bounding regio's MBR_1 en MBR_2 en een grotere bounding regio MBR_0, die beiden omvat. Een MBR_i bevat C_i vectoren en heeft een probabiliteit X_i om geïntersecteerd te worden. De kostfunctie is dan:

$$\delta_{kost} = (t_{zoeken} + C_1 \cdot t_{punt}) \cdot X_1 + (t_{zoeken} + C_2 \cdot t_{punt}) \cdot X_2 - (t_{zoeken} + C_0 \cdot t_{punt}) \cdot X_0$$

Als $\delta_{kost} > 0$ is, dan is een NN-query, gebruikmakend van de maximum-afstandsmaat, lokaal kostelijker met de twee aparte bounding regio's dan als men de twee bounding regio's vervangt door één grote. Het is dan beter om de twee samen te voegen tot één. We merken op dat de twee bounding regio's langs elkaar liggen. Als de kostfunctie δ_{kost} negatief is, dan is het dus beter om te bounding regio op te splitsen in twee kleinere bounding regio's. Waar te gaan splitsen kan men eventueel bepalen door naar een minimum voor de functie $(t_{zoeken} + C_1 \cdot t_{punt}) \cdot X_1 + (t_{zoeken} + C_2 \cdot t_{punt}) \cdot X_2$ te zoeken. Men moet dan wel eerst een functie-onderzoek doen om te bekijken als het lokale minimum van deze functie ook een globaal minimum is.

7.2 Locality of restructuring

Door voortdurend data te verwijderen en toe te voegen op een opslagmedium, kan het zijn dat er plaatsen tussen opeenvolgende datablokken zijn die geen data bevatten. De stelling [5] die we hier aantonen heeft betrekking op de vraag hoeveel bytes er verloren zijn gegaan over een bepaald interval van bytes waarbij de verloren bytes opgevat kunnen worden als de plaatsen tussen de blokken met data. De gegevens hier hebben betrekking op harde schijven. We bekijken het plaatsen van data op een schijf in het algemeen. We weten dat de data verspreid is over een interval $i_{volledig}$ van b bytes en dat $data$ procent van het interval ook werkelijk data bevat. We willen dat minstens $data_{min}$ procent van $i_{volledig}$ data bevat. Stel dat we nu s bytes willen toevoegen en dat er geen s opeenvolgende niet-gebruikte bytes zijn in $i_{volledig}$. Indien mogelijk, kunnnen we $i_{volledig}$ uitbreiden met s bytes. Veronderstel dat hierdoor $data < data_{min}$ wordt. Om aan de voorwaarde $data \geq data_{min}$ te voldoen, moeten we de data in $i_{volledig}$ herstructureren. We willen het interval dat geherstructureerd moet worden liefst zo klein mogelijk houden. Met andere woorden, wat is het kleinste interval dat we moeten herstructureren om s bytes opeenvolgend op te slaan?

Stelling Locality of restructuring. *Gegeven een interval $i_{volledig}$ van b bytes op een opslagmedium. Zij data hoeveel procent van $i_{volledig}$ data bevat. We stellen de eis dat $data \geq data_{min}$, waarbij $data_{min}$ een percentage is dat*

we zelf bepalen. Zij echter $data \leq data_{min}$[4]*. We willen s bytes toevoegen aan het opslagmedium. Dan geldt er dat er* een *interval van lengte* l *in* $i_{volledig}$ *bestaat dat s ongebruikte bytes bevat, waarbij*

$$l = \frac{s}{1 - data_{min}}$$

Bewijs. We bewijzen de stelling door middel van contradictie. We nemen aan dat alle intervallen van lengte l in $i_{volledig}$ minder dan s bytes bevatten. Het aantal vrije bytes in $i_{volledig}$, aangeduid door e, kan dan begrensd worden door:

$$e < \frac{b}{l} \cdot s$$

Bij definitie geldt:

$$data = 1 - \frac{e}{b}$$

Uit de twee bovenstaande feiten volgt:

$$data > 1 - \frac{s}{l}$$

Uit de definitie van l volgt:

$$data_{min} = 1 - \frac{s}{l}$$

Uit de laatste twee feiten volgt dan dat:

$$data > data_{min}$$

Dit laatste is een contradictie met het gegeven $data \leq data_{min}$ □

Het bepalen van waar en hoeveel vrije bytes er zijn, kan men versnellen door er info over bij te houden in een datastructuur.

7.3 Samenvatting

We hebben besproken hoe het queryen kan geoptimaliseerd worden door de blokgrootte van de blokken, waarin de indexdata zich bevindt, tijdens de levensduur van de index aan ta passen met behulp van een kostmodel. Het gegeven kostmodel kan men vervangen door een ander kostmodel. Tenslotte vermeldden we een stelling die ons helpt om een zo klein mogelijk interval te bepalen, waarin data geherstructureerd moet worden, met het doel de ruimte op de schijf zo efficiënt mogelijk te gebruiken.

[4] We kunnen $data$ verminderen tot $data_{min}$ door $i_{volledig}$ uit te breiden of door $data_{min}$ gelijk te stellen aan $data$. In beide gevallen blijven we aan de eis $data \geq data_{min}$ voldoen.

8 Besluit

We zien tegenwoordig dat steeds meer databases behalve de atomische datatypen, ook meer complexere datatypen ondersteunen. Denk hierbij maar aan GIS, XML, ... Men gebruikt echter nog niet veel multidimensionale indexen. Met vooruitgangen in het multimediadatabase-domein komt hier in de nabije toekomst misschien verandering in. Het multimediadatabase-domein is een omvangrijk domein waarin onderzoeksdomeinen zoals databases, multimedia, logica, gebruikersinterfaces, ... verenigd worden. Tijdens het zoeken naar informatie over multimediadatabases is het mij opgevallen dat er weinig bronnen zijn die de theorie en de technieken die men kan aanwenden bij het ontwikkelen van een multimediadatabase, groeperen en er een overzicht van geven. Daarom is er in dit proefschrift getracht een overzicht te geven van een deel van de technologieën die gebruikt kunnen worden bij het ontwikkelen van multimediadatabases. We hebben de technologieën ingedeeld op de volgende manier: indexen, algoritmen, concurrency en opslag.

We hebben de structuur en de operaties van de SS-tree beschreven. Dit is een index die niet gedetailleerd uitgelegd is in de originele paper en waarop een aantal andere indexen verder bouwen. Omwille van de "Curse of Dimensionality" weten we dat de performantie van indexen die partitioneren/clusteren evolueert naar die van een lineaire scan naarmate de dimensionaliteit stijgt. We hebben hierom een index/datastructuur bekeken met als doel de lineaire scan te versnellen. Er is ook getracht het nut van multidimensionale indexen te tonen, door een vergelijking te maken met inverted lists. We hebben namelijk beschreven hoe men een multidimensionale ruimte kan laten indexeren door een optimaal aantal indexen, waarbij de dimensies verdeeld worden over de indexen. Op deze manier trachten we ook de "Curse of Dimensionality" zo veel mogelijk uit te stellen. Uiteindelijk beschrijven we ook nog een bulkload-methode waarmee men de index-constructietijd kan versnellen als men veel feature-vectoren in één keer kan inladen. Er zijn nog heel wat andere indexen die andere technieken aanwenden en/of in dit proefschrift beschreven technieken optimaliseren om een zo goed mogelijke performantie te bekomen. Naast het optimaliseren van de zoeksnelheid, kan men ook het plaatsgebruik verbeteren.

De performantie van de indexen wordt in de originele papers experimenteel gemeten en niet theoretisch. We hebben wel een algemeen kostmodel gegeven dat de performantie van NN/range-queries op multidimensionale indexen bepaalt. Het is echter geen model dan men vlug kan berekenen voor een bepaalde input. Men stelt voor om tabellen te gebruiken om de berekeningen vlug te kunnen uitvoeren. Het kan dan echter zijn dat men de gegevens van deze tabellen van een opslagmedium moet inladen, wat een vertraging

veroorzaakt. Momenteel is er nog geen volledig kostmodel dat de kost van een NN/range-query in een hoog-dimensionale ruimte vlug kan berekenen. In [42] stelt men een kostmodel voor laag- en medium-dimensinale ruimten voor, dat men vlug kan berekenen. Voor het uitvoeren van (k-)NN-queries beschrijven we een optimaal incrementeel single-step-algoritme dat ook kan gebruikt worden voor range-queries en een r-optimaal k-multistep-algoritme.

Om de performantie/interactiviteit te verhogen, kan men concurrency toepassen. Omdat bij de indexen hoofdzakelijk de performantie van het zoeken belangrijk is, proberen we dit ook hier te optimaliseren. We vermelden twee methoden waarvan één rekening houdt met forced reinsert.

Tenslotte vermelden we nog kort waarom de opslag van de indexen wordt beïnvloed door de dimensionaliteit.

Een aantal onderdelen van een database die niet aanwezig zijn in dit proefschrift, zijn onder andere buffer management [12] [24] en logische optimalisatie van queries [36]. Quality-of-service [29] [37] is een aspect van multimediadatabases dat hier niet besproken is. Denk hierbij maar aan het over een netwerk afleveren van video's, presentaties, ... die een bepaalde quality-of-service moeten hebben.

Er is dus al heel wat onderzoek verricht in het domein van multimediadatabases, maar er zijn nog onopgeloste problemen zoals een kostmodel voor NN-queries in hoog-dimensionale ruimten dat efficiënt kan worden berekend. De vraag naar multimediadatabases zal misschien stijgen door de steeds grotere hoeveelheid data en de verscheidenheid aan data op het internet, bij politiediensten, ... Dit zou dan de algemene interesse in dit domein kunnen vergroten en de vooruitgang ervan versnellen.

Referenties

[1] Christian Böhm. A cost model for query processing in high-dimensional data spaces. *ACM Trans. Database Syst.*, 25(2):129–178, 2000.

[2] Christian Böhm and Hans-Peter Kriegel. Efficient bulk loading of large high-dimensional indexes. In *Int. Conf. on Data Warehousing and Knowledge Discovery (DaWak)*, pages 251–260, 1999.

[3] Leejay Wu Caetano Traina Jr., Agma J. M. Traina and Christos Faloutsos. Fast feature selection using fractal dimension. In *XV Brazilian Symposium on Databases (SBBD)*, pages 158–171, 2000.

[4] Kaushik Chakrabarti and Sharad Mehrotra. Efficient concurrency control in multidimensional access methods. In *Proc. of the 1999 ACM SIGMOD Int. conf. on Management of data*, pages 25–36, 1999.

[5] Christian Böhm en Hans-Peter Kriegel. Dynamically optimizing high-dimensional index structures. In *Proc. 7th Int. Conf. on Extending Database Technology(EDBT)*, pages 36–50, 2000.

[6] Seok II Song en Jae Soo Yoo. An efficient concurrency control algorithm for high-dimensional index structures. *Journal of Database Management*, 15(3):60–72, 2004.

[7] Tolga Bozkaya en Meral Özsoyoğlu. Distance-based indexing for high-dimensional metric spaces. In *Proc. of the 1997 ACM SIGMOD int. conference on Management of data*, pages 357–368, 1997.

[8] Norio Katayama en Shin'ichi Satoh. The sr-tree: An index structure for high-dimensional nearest neighbour queries. In *Proc. of the 1997 ACM SIGMOD Int Conf. on Management of Data*, pages 369–380, Tucson, Arizon USA, 1998.

[9] Sproull R. F. Refinements to nearest neighbour searching in k-dimensional trees. *Algorithmica*, 6(4):579–589, 1991.

[10] Christos Faloutsos and Ibrahim Kamel. Beyond uniformity and independence: Analysis of r-trees using the concept of fractal dimension. In *Proc. of the thirteenth ACM SIGACT-SIGMOD-SIGART symp. on Principles of database systems (PODS)*, pages 4–13, 1994.

[11] Bernd-Uwe Pagel Flip Korn and Christos Faloutsos. On the 'dimensionality curse' and the 'self-similarity blessing'. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 13(1):96–111, 2001.

[12] Sreenivas Gollapudi and Aidong Zhang. Buffer management in multimedia database systems. In *ICMCS*, pages 186–, 1996.

[13] Kalos M. H. and Whitlock P. A. Monte carlo methods, 1986.

[14] Gísli R. Hjaltason and Samet H. Ranking in spatial databases. In *Proc. 4th Int. Symp. on Large Spatial Databases*, pages 83–95, 1995.

[15] Gísli R. Hjaltason and Hanan Samet. Incremental similarity search in multimedia databases. Technical report, TR 4199, Department of Computer Science, University of Maryland, 2000.

[16] Jon Louis Bentley Jerome H. Friedman and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1975.

[17] Tamer Kahveci and Ambuj Singh. Efficient index structures for string databases. In *Proc. of the 27th Int. Conf. on Very Large Databases*, pages 351–360, 2001.

[18] Chang-Ryong Kim and Chin-Wan Chung. A multi-step query processing in large image data using histogram intersection. Technical report, CS/TR-2000-153,Department of Electrical Engineering and Computer Science,KAIST, 2000.

[19] James J. Little and Zhe Gu. Video retrieval by spatial and temporal structure of trajectories. In *Proc. 13th Int. Symp. on Storage and Retrieval for Image and Video Databases (SPIE)*, pages 544–553, 2001.

[20] David B. Lomet and Betty Salzberg. Concurrency and recovery for index trees. *VLDB J.*, 6(3):224–240, 1997.

[21] Guojun Lu. *Multimedia Database Management Systems.* Artech House, 1999.

[22] Hans-Peter Kriegel Mihael Ankerst and Thomas Seidl. A multistep approach for shape similarity in image databases. *IEEE Transactions on knowledge and data engineering*, 10(6):996–1004, 1998.

[23] Ralf Shneider Norbert Beckmann, Hans-Peter Kriegel and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.

[24] Banu Özden, Rajeev Rastogi, and Abraham Silberschatz. Buffer replacement algorithms for multimedia storage systems. In *ICMCS*, pages 172–180, 1996.

[25] Marco Patella Paolo Ciaccia and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of the 23rd VLDB Conference*, pages 426–435, 1997.

[26] Hans-Jörg Schek Roger Weber and Stephen Blott. A quantative analysis and performance study for simiality-search methods in high-dimensional spaces. In *Proceedings of the 24th VLDB Conference*, pages 194–205, 1998.

[27] Young Ho Kim en Jae Soo Yoo Seok II Song. An enhanced concurrency control scheme for multidimensional index structures. In *7th Int. Conf. on Database Systems for Advanced Applications (DASFAA)*, 2001.

[28] B. Narahari S.R. Subramanya, Rahul Simha and Abdou Youssef. Transform-based indexing of audio data for multimedia databases. In *Int. Conf. on Multimedia Computing and Systems (ICMS)*, pages 211–218, 1997.

[29] Richard Alan Staehli. *Quality of Service Specification for Resource Management in Multimedia Systems*. PhD thesis, Oregon Graduate Institute, 1996.

[30] Daniel A. Keim Stefan Berchtold, Christian Böhm, Florian Krebs, and Hans-Peter Kriegel. On optimizing nearest neighbour queries in high-dimensional data spaces. In *Proc. IEEE Int. Conf. on Database Theory (ICDT)*, pages 435–449, 2001.

[31] Daniel A. Keim Stefan Berchtold, Christian Böhm, Hans-Peter Kriegel, and Xiaowei Xu. Optimal multidimensional query processing using tree striping. In *Proc. 2nd Int. Conf. on Data Warehousing and Knowledge Discovery(DaWaK)*, pages 244–257, 2000.

[32] H.V. Jagadish Hans-Peter Kriegel Stefan Berchtold, Christian Böhm and Jörg Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *16th Int. Conference on Data Engineering(ICDE)*, pages 577–588, 2000.

[33] Christian Böhm Stefan Verchtold and Hans-Peter Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 216–230, 1998.

[34] V.S. Subrahmania. *Principles of Multimedia Database Systems*. Morgan Kaufmann, 1998.

[35] George Tzanetakis and Perry Cook. Musical genre classification of audio signals. *IEEE Transactions on speech and audio processing*, 10(5):293–302, 2002.

[36] Wolf-Tilo Balke Ulrich Güntzer and Werner KießBling. Optimizing multi-feature queries for image databases. In *Proc. VLDB 2000*, pages 419–428, 2000.

[37] Jonathan Walpole, Ling Liu, David Maier, Calton Pu, and Charles Krasic. Quality of service semantics for multimedia database systems. In *DS-8*, pages 393–412, 1999.

[38] David A. White and Ramesh Jain. Similarity indexing with the ss-tree. In *ICDE*, pages 516–523, 1996.

[39] Angeline Wong, Leejay Wu, Philip B. Gibbons, and Christos Faloutsos. Fast estimation of fractal dimension and correlation integral on stream data. *Inf. Process. Lett.*, 93(2):91–97, 2005.

[40] Jeffrey Xu Yu Xiangmin Zhou, Guoren Wang and Ge Yu. M^+-tree:a new dynamical multidimensional index for metric spaces. In *Proc. of the Fourteenth Australasian database conference on Database technologies 2003 (ADC)*, volume 17, pages 161–168, 2003.

[41] Shunsuke Uemura Yasushi Sakurai, Masatoshi Yoshikawa and Haruhiko Kojima. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *Proc. of the 26th Int. Conference on Very Large Data Bases(VLDB)*, pages 516–526, 2000.

[42] Dimitris Papadias Yufei Tao, Jun Zhang and Nikos Mamoulis. An efficient cost model for optimization of nearest neighbour search in low and medium dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering*, 16(10):1169–1184, 2004.