Organisatie en doorzoeken van een gegevensverzameling

Masterproef

Nick Gaens

Promotor: Prof. dr. Jan Vandenbussche

Assessoren: Steven Maesen en Jelle Hellings

Universiteit Hasselt
Academiejaar 2013-2014

Contents

List of Figures

1 Inleiding

In deze thesis wordt er vertrokken vanuit een algemeen probleem: een gebruiker die een zoekactie wilt ondernemen en daarbij verschillende fases moet doorlopen vooraleer hij of zij tot het opstellen van een query komt. Er wordt daarbij een onderscheid gemaakt op basis van de algemeenheid en complexiteit van het vraagstuk waar de gebruiker een antwoord voor probeert te bekomen: het opstellen van een query voor een complex probleem van een algemenere aard is moeilijker dan een opstellen voor een ronduit simpel en specifiek probleem. Het bekomen van een query is een eerste probleem dat dus overwonnen moet worden.

Een tweede probleem houdt in dat het gestructureerd organiseren van een (digitale) gegevensverzameling, waarop de query uit te voeren is, een moeilijke opgave is. Neem bijvoorbeeld het brede scala aan real-world concepten als 'auto', 'dier, 'land', 'hotel' enzovoorts, of - op een kleinere schaal - alle computerbestanden die zich binnen een enkel bestandssysteem bevinden. Het is allesbehalve triviaal om een representatie te creëeren die zulke gegevens van een sterk uiteenlopende aard op een uniforme manier niet enkel op kan slaan (en weergeven) maar ook nog eens ondersteuning biedt aan het efficiënt doorzoeken van die gegevens, bijvoorbeeld met behulp van een grafische user interface of door middel van een tekstuele querytaal.

In dalende maat van abstractie komen in die context vier verschillende technologieën aan bod, die dus het representeren van sterk uiteenlopende data mogelijk maken, met daarbij ook een manier om de data te queryen. Als eerste komt de abstracte attribute-value dataspace aan bod met daarbij de zoektalen Boolean Search Language (BSL) en Associative Search Language (ASL), waarvan de laatste als uitbreiding op de eerste geldt. Vervolgens wordt NoSQL geïntroduceerd met daarbij de bespreking van de vier meest prominente, niet-relationele databasetechnologieën die onder die naam thuishoren. De opsomming beëindigen gebeurt met twee bestaande onderzoeksprojecten, namelijk Search Computing en iMeMex Personal Dataspace Management. Search Computing implementeert een graafrepresentatie van onderling verbonden real-world objecten. De aangeboden graaf kan vervolgens met behulp van een user interface gemanipuleerd worden om zo visueel queries op te stellen. Het iMeMex-project laat op zijn beurt toe om een traditioneel computerbestandssysteem voor te stellen door middel van een boomstructuur en te queryen met een eigen, gespecialiseerde querytaal, iQL. Voor zowel de voorgestelde NoSQL-technologieën, Search Computing als iMeMex wordt een brug geslagen naar de abstracte attribute-value dataspace: op welke manier kunnen ze elk gemodelleerd worden met attribute-value dataspaces en welke kenmerken van attribute-value dataspaces

belichamen ze op welke manier.

Eindigen doet deze thesis met een meer praktische invalshoek door op zoek te gaan naar een idee van de werking en performantie van de attribute-value dataspace-zoektaal ASL. Om dat te doen, zijn er benchmarks uitgevoerd: enerzijds zijn de uitvoeringstijden van ASL-expressies op gesimuleerde attribute-value dataspaces gemeten en anderzijds de uitvoeringstijden van equivalente SQL-queries op een traditioneel, relationeel model. Door onder andere het analyseren van visuele queryplannen zal getracht worden enkele in het oog springende resultaten te verklaren.

2 Zoekproces

Een gebruiker die (digitaal) zoekt naar informatie van eender welke aard ondergaat - vaak zonder besef daarvan - achtereenvolgens verschillende fasen [1] om uiteindelijk tot op het punt te geraken dat er daadwerkelijk een query ingevoerd wordt. Hieronder worden eerst die fasen voorgesteld, waarna een onderscheid gemaakt wordt tussen abstracte en concrete enerzijds en moeilijke en makkelijke queries anderzijds.

2.1 Fasen

Zoals vermeld in [1], heeft een gebruiker vier fasen doorlopen voordat hij of zij een eerste resultaat ziet van de zoektocht naar de gewenste informatie. Die vier fasen zijn de volgende:

- **rondwandelen**: de gebruiker heeft nog geen intentie om naar informatie te zoeken;
- **verkennen**: de gebruiker beseft dat hij of zij informatie wil bekomen, maar weet nog niet hoe dat te doen;
- **zoeken**: de gebruiker bepaalt welke informatie moet bekomen worden en
- **vragen**: de gebruiker formuleert een vraag die de gewenste informatie moet bekomen.

Een grafische voorstelling van dit proces is weergegeven in 1. Merk ook op dat hoe 'smaller' een fase is weergegeven op die figuur, des te concreter de te realiseren acties van een gebruiker en diens denkpatronen zijn.

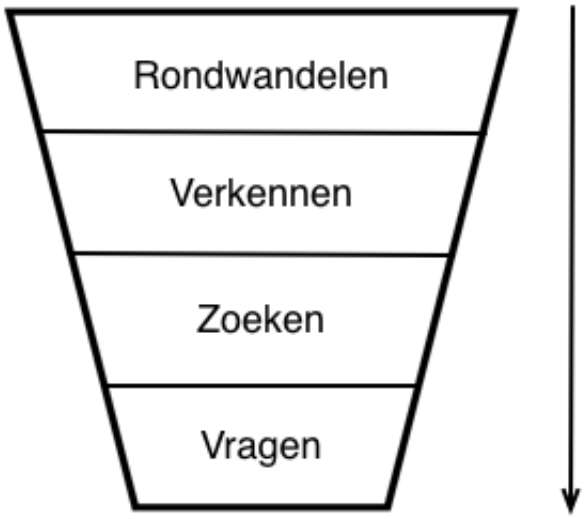

Figure 1 – De informatietrechter met daarbij een pijl die de volgorde aangeeft waarmee de vier fasen doorlopen worden.

2.2 Moeilijkheid van een query

Als een gebruiker op het punt gearriveerd is dat hij of zij een query moet opstellen, kan het zijn dat die query zowel erg abstract als moeilijk blijkt te zijn om te formuleren. Er valt dus een onderscheid te maken op basis van de abstractie en moeilijkheid van het vraagstuk dat de gebruiker voor zichzelf formuleert op het moment dat hij of zij beseft naar welke informatie te willen zoeken. Beschouw even de volgende vier voorbeeldvraagstukken:

- A (algemeen met lage complexiteit): "Leer iets bij over Hasselt";
- B (concreet met lage complexiteit): "Wat is de hoofdstad van België?";
- C (concreet met hoge complexiteit): "Hoeveel luchtvaartmaatschappijen vliegen er van Brussel naar New York?" en
- D (algemeen met hoge complexiteit): "In welke steden met meer dan 30000 inwoners bestaan er sterrenrestaurants?".

Als de twee aanwezige parameters, namelijk de complexiteit van de vraag en de algemeenheid van de doelstelling, op een assenstelsel geplot worden, met daarop aangeduid waar de vier voorgaande voorbeeldvraagstukkken zich in dat stelsel bevinden, dan levert dat een figuur op zoals weergegeven in figure 2. Merk aan die figuur op dat er een gekleurde zone afgebakend is door middel van een stippellijn. Die lijn representeert de notitiegrens [1]

en geeft een minimum aan voor zowel de complexiteit van de vraag als de algemeenheid van de doelstelling van de gebruiker waaraan voldaan moet worden, opdat die gebruiker meerdere queries zal moeten formuleren om zijn of haar antwoord te bekomen. Met andere woorden: de gebruiker zal tussentijdse resultaten moeten noteren.

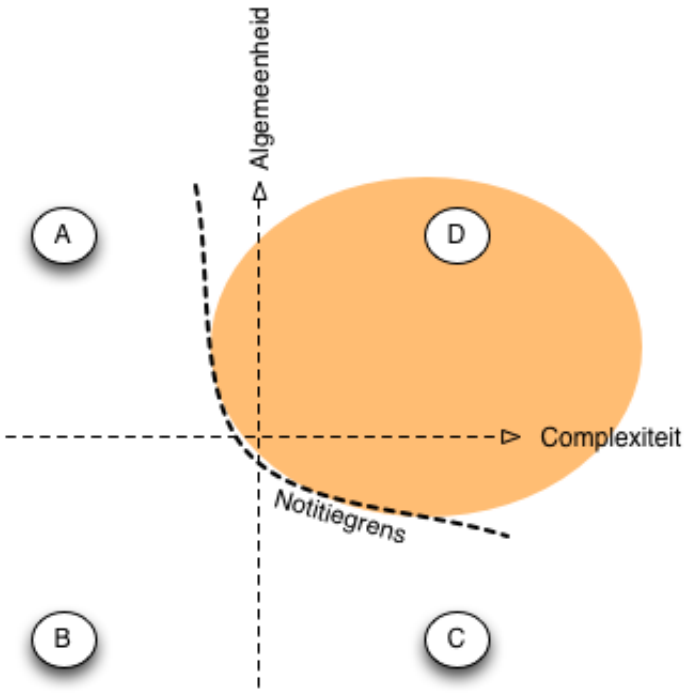

Figure 2 – Een assenstelsel met op de horizontale as de complexiteit van de in te voeren query en op de verticale as de algemeenheid of abstractie van het vraagstuk dat de gebruiker geformuleerd heeft. De vier omcirkelde letters geven queries aan die thuishoren in elk deel van het stelsel. Weergegeven met een stippellijn is de 'notitiegrens': een minimale combinatie van querycomplexiteit en vraagstuk-abstractie die er voor zorgt dat een gebruiker meerdere queries nodig heeft om zijn doel te bereiken en dus de resultaten van de tussentijdse queries zal moeten bijhouden of noteren.

3 Attribute-Value Dataspace

Een *attribute-value dataspace* [1] is een abstract datarepresentatiemodel waarbij de gegevens bestaan uit slechts eenvoudige attribute-value-paartjes die op hun beurt gebundeld worden in objecten. Beschouw daarbij alvast de volgende definities en notaties:

- een item i beslaat een attribute-value-paar en is van de vorm $(a,\ v)$;
- de verzameling van alle items is voorgesteld door I (ook wel het universum genoemd);
- een I-object o is een eindige, niet-lege verzameling van items uit I;
- de verzameling van I-objecten is voorgesteld door O en
- een I-dataspace D is een eindige verzameling van I-objecten.

Een grafische voorstelling van deze opbouw is terug te vinden in figure 3.

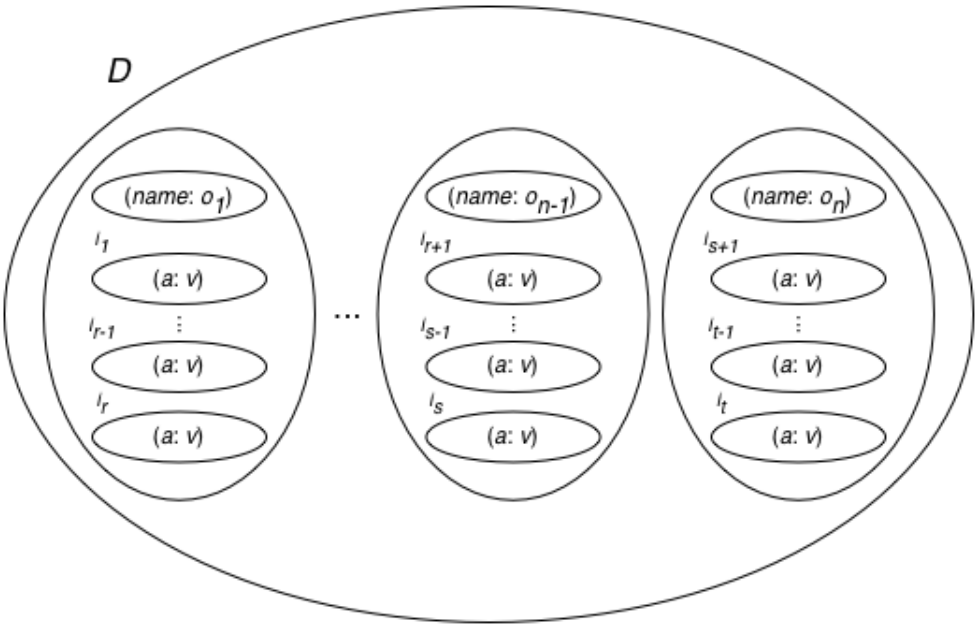

Figure 3 – Opbouw van een attribute-value dataspace.

Kenmerken Attribute-value dataspaces beschikken over de volgende typerende kenmerken:

[1] De termen *key-value dataspace* en *attribute-value dataspace* zijn gelijkwaardig, maar in deze thesis gaat de voorkeur uit naar de laatste.

- Attribute-value dataspaces laten 'loosely structured data' toe, waarmee bedoeld wordt dat de objecten binnen een dataspace onderling helemaal niet met elkaar verbonden moeten zijn. Een kleine opmerking hierbij kan zijn dat elk item toch het *object-id* (bijvoorbeeld o_1) van het object waarin ze zich bevinden met zich meedraagt. Dus de vorm van een item is niet zozeer (a, v) zoals initieel vermeld, maar eerder (*object_id*, a, v). Het object_id wat betreft de opbouw van een attribute-value dataspace van minder belang en dus wordt het gemakkelijkheidshalve weggelaten in de notatie.

- Gegevens moeten niet voldoen aan een of ander beperkend schema vooraleer ze toegelaten worden tot een dataspace. Zulke schema's zijn bijvoorbeeld wel heel erg prominent aanwezig bij traditionele relationele databases, waarbij voor elk attribuut van een relatie op z'n minst nauwkeurig vastgelegd moet zijn welke naam het zal hebben van welk type het zal zijn (bijvoorbeeld een integer of een string). Het ontbreken van een schema resulteert bij attribute-value dataspaces in het aanbieden van een grote vrijheid en flexibiliteit wat betreft de potentiële inhoud ervan. Zo bestaat er bijvoorbeeld geen beperking op het al dan niet voorkomen van meerdere keys of values met dezelfde waarde binnen éénzelfde object.

- Minder prominent, maar daarom niet minder belangrijk, is dat attribute-value dataspaces gegevens die divers van aard zijn uit meerdere bronnen terzelfdertijd accepteren. Het maakt met andere woorden zowel niet uit wat voor gegevens in een attribute-value dataspace als van waar die gegevens afkomstig zijn. Deze eigenschap is eigenlijk een gevolg van beide voorgaande eigenschappen.

De bovenstaande kenmerken komen verderop in de thesis nog aan bod als ze getoetst worden aan concretere datarepresentatiemechanismen en technologieën.

3.1 Dataspace Query Languages

Zoeken in een attribute-value dataspace kan door het creëeren van een booleaanse expressie, zijnde een of meerdere keywords die aan elkaar geschakeld worden met behulp van booleaanse operatoren. In deze sectie introduceren we in die context de taal BSL, wat staat voor Boolean Search Language [5]. Deze introductie omvat een formele definitie van de syntax van BSL, de semantiek van een BSL-expressie en uiteindelijk enkele voorbeelden. Behalve BSL wordt verderop ook ASL, wat staat voor Associative Search Language, als uitbreiding van BSL op dezelfde manier voorgesteld.

3.1.1 BSL

BSL (of Boolean Search Language) is een dataspace querytaal waarmee booleaanse zoekexpressies geconstrueerd kunnen worden. De taal maakt van slechts twee bouwstenen gebruik: *keywords* en drie booleaanse operatoren zijnde 'and', 'or' en 'except'. In deze subsectie wordt vooreerst de syntax van BSL uit de doeken gedaan door die voor te stellen met behulp van *Backus-Naur Form*-syntaxgrammatica. Die voorstelling (zie figure 4) toont aan op welke manier keywords opgebouwd zijn alsook hoe expressies in BSL gevormd worden door de aaneenschakeling van zulke keywords met behulp van de beschikbare operatoren. Behalve de syntax komt ook de semantiek van BSL aan bod: er wordt gekeken naar wat een keyword uitdrukt en welke betekenis een volledige BSL-expressie met zich meedraagt.

3.1.1.1 Syntax De syntax van BSL kan voorgesteld worden met behulp van *Backus–Naur Form*-syntaxgrammatica [11] zoals weergegeven in figure 4.

```
<boolean_operator> ::= "and" | "or" | "except"
<attribute-list> ::= <attribute> | <attribute> "," <attribute_list>

<positive_key> ::= <attribute> | "{" <attribute_list> "}"
<negative_key> ::= "NOT" <positive_key>
<key> ::= <positive_key> | <negative_key> | "*"

<positive_value> ::= <attribute> | "{" <attribute_list> "}"
<negative_value> ::= "NOT" <positive_value>
<value> ::= <positive_value> | <negative_value> | "*"

<keyword> ::= "(" <key> ":" <value> ")" | "(" "*" ")"
<bsl_expression> ::= <keyword> | <keyword> <boolean_operator> <bsl_expression>
```

Figure 4 – Syntax van BSL vastgelegd in een Backus-Naur Form-syntaxbeschrijving.

De opvallendste eigenschappen van deze syntaxbeschrijving zijn:

- de drie operatoren (*and*, *or* en *except*);
- de belangrijkste bouwsteen van BSL in de vorm van het *keyword* is in feite een key-value-constructie;
- de wildcard (notatie: $*$, zie hieronder) die in de key of value van een keyword kan voorkomen, of het keyword kan vervangen indien zowel de key als de value een wildcard zijn;

- een BSL-expressie is een (recursieve) aaneenschakeling van keywords door de drie operatoren.

Keyword Zoals hier net boven reeds vermeld staat, is het keyword één van de twee bouwstenen van BSL. De attribute-valueconstructie van een keyword bestaat uit een positieve of negatieve attribute en een positieve of negatieve value, gescheiden door een dubbelpunt. Stel even de volgende notatie:

- zij a een key en Σ een eindige verzameling van keys zodat dus $a \in \Sigma$;
- zij v een value en V een oneindig domein voor values zodat dus $v \in V$;
- zij $\mathrm{A} \subseteq \Sigma$ en $\mathrm{V} \subseteq V$;
- $\neg$ representeert de negatie.

Er zijn dan vier mogelijke vormen van attribute-value keywords die afgekort genoteerd worden als: $(a{:}\ v)$, $(a{:}\ \neg\mathrm{V})$, $(\neg\mathrm{A}{:}\ v)$ en $(\neg\mathrm{A}{:}\ \neg\mathrm{V})$. Een overzicht van hun volledige voorstelling:

$$
\begin{array}{rcl}
(a{:}\ v) & \text{met} & (a{:}\ v) := \{(a,\ v)\} \\
(a{:}\ \neg\mathrm{V}) & \text{met} & (a{:}\ \neg\mathrm{V}) := \{(a,\ v) \mid v \in V \text{ - } \mathrm{V}\} \\
(\neg\mathrm{A}{:}\ v) & \text{met} & (\neg\mathrm{A}{:}\ v) := \{(a,\ v) \mid a \in \textstyle\sum \text{ - } \mathrm{A}\} \\
(\neg\mathrm{A}{:}\ \neg\mathrm{V}) & \text{met} & (\neg\mathrm{A}{:}\ \neg\mathrm{V}) := \{(a,\ v) \mid a \in \textstyle\sum \text{ - } \mathrm{A},\ v \in V \text{ - } \mathrm{V}\}
\end{array}
$$

Een voorbeeld van elk van deze vier vormen, in dezelfde volgorde als hierboven:

(name: Nick)
(name: ¬{Nick, Bill, Steve})
(¬{name, occupation}: Nick)
(¬{name, occupation}: ¬{Nick, Bill, Software developer})

Wildcard Als aanvulling op de algemene vormen van een keyword (zie hier net boven) bestaan er situaties waarin a en/of v een lege verzameling voorstellen. In het geval dat slechts één van beide gelijk is aan $\emptyset$, wordt de notatie $*$ aangewend. Voorbeelden van het resultaat hiervan zijn de keywords ({name, occupation}: $*$) en ($*$: Bill). Als zowel a als v gelijk zijn aan $\emptyset$, noteren we niet ($*$: $*$), maar kortweg ($*$).

Operator De drie in BSL voorkomende operatoren (*and*, *or* en *except*) zijn binair van aard en ook rechttoe rechtaan wat betreft hun syntax. Daarom dat er enkel de onderstaande voorbeelden gegeven worden ter indicatie van hun syntax.

$$
\begin{aligned}
e_1 &:= \text{(name: } \neg\text{Nick) AND (setting: \{race, victory\})} \\
e_2 &:= \text{(setting: race) OR (team: Ferrari)} \\
e_3 &:= \text{(setting: victory) EXCEPT (team: Ferrari)}
\end{aligned}
$$

3.1.1.2 Semantiek

Keyword en wildcard De volgende definities leggen de semantiek van een keyword formeel vast:

- zij k een keyword, dan is k een deelverzameling van het universum I (zie ook de aanvang van dit hoofdstuk);
- zij i een item met $i \in I$, dan wordt gezegd dat i 'matcht' met k als $i \in k$;
- zij o een I-object, dan wordt gezegd dat o 'voldoet' aan k (genoteerd met $o \models k$) als $\exists\ i \in o$ waarvoor geldt dat 'i matcht met k' (oftewel $o \cap k \neq \emptyset$).

Beschouw ter voorbeeld de volgende vier keywords:

$$
\begin{aligned}
k_1 &:= \text{(name: Michael Schumacher)} \\
k_2 &:= \text{(name: } \neg\{\text{Michael Schumacher, Lewis Hamilton}\}) \\
k_3 &:= (\neg\{\text{setting, team}\}\text{: } *) \\
k_4 &:= (*)
\end{aligned}
$$

Dan is de 'betekenis' van die keywords als volgt:

k_1 := Valideer alle objecten met een key gelijk aan 'name' en een value gelijk aan 'Michael Schumacher'.

k_2 := Valideer alle objecten met een key gelijk aan 'name', maar met een value die zowel verschillend is van 'Michael Schumacher' als 'Lewis Hamilton'.

k_3 := van 'setting' als 'team' en met eender welke value.

k_4 := Valideer alle objecten met een key die zowel verschillend is Valideer alle objecten met eender welke key en eender welke value.

Merk op dat keywords op zich impliciete operatoren kunnen bevatten. Zo bevat het k_2 een impliciete 'and', want er wordt bedoeld: "... verschillend is van 'Michael Schumacher' **en** verschillend van 'Lewis Hamilton'.". Ook k_3 bevat zo'n impliciete 'and'.

De semantiek van een wildcard is aan de hand van k_3 en k_4 ook meteen duidelijk: een wildcard doet niets meer dan geen restrictie opleggen aan de key en/of value van een keyword.

Operator De semantiek van de drie operatoren (and, or en except) is als volgt:

$$\begin{aligned}(e_1 \text{ AND } e_2)(D) &:= e_1(D) \cap e_2(D)\\ (e_1 \text{ OR } e_2)(D) &:= e_1(D) \cup e_2(D)\\ (e_1 \text{ EXCEPT } e_2)(D) &:= e_1(D) - e_2(D)\end{aligned}$$

Het valt op dat er een overeenkomst bestaat tussen de operatoren en de verzamelingenleer anderzijds. Zo *mappen* de operatoren and, or en except respectievelijk op de doorsnede, unie en het verschil.

3.1.1.3 Voorbeelden Het is nuttig om enkele totaalvoorbeelden van BSL-expressies te leveren en die eens op een (kleine) voorbeelddataspace in te laten werken. Om dat te doen wordt enerzijds de voorbeelddataspace D geïntroduceerd, zoals weergegeven in figure 5.

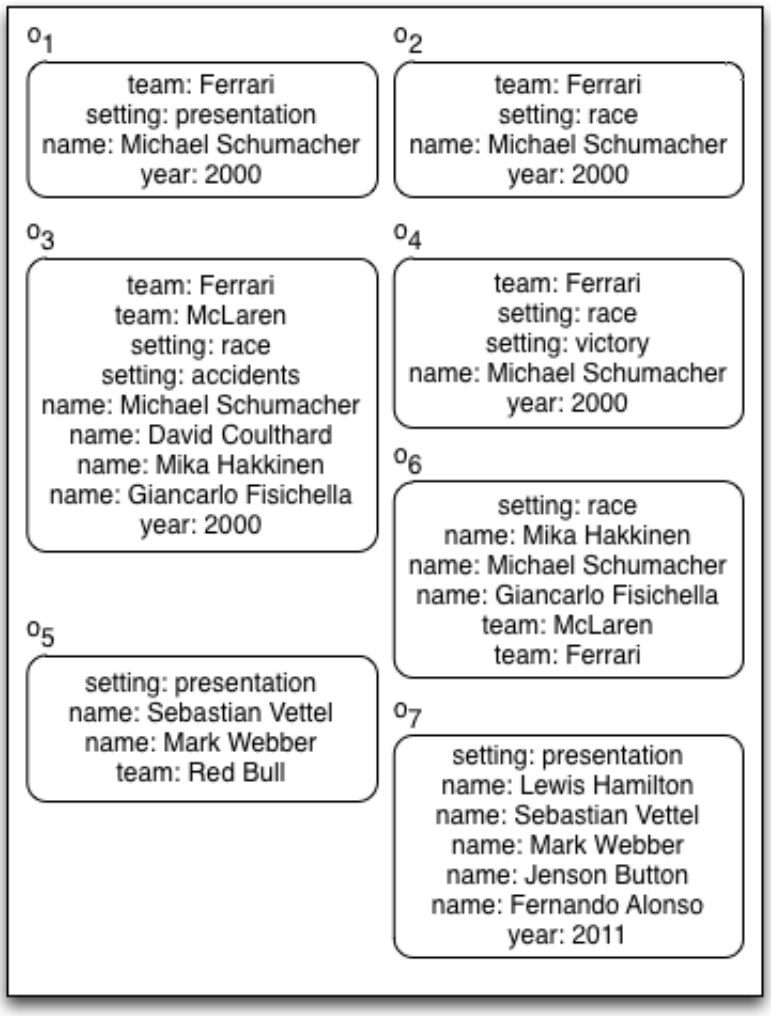

Figure 5 – Een attribute-value dataspace D met daarin objecten o_1 tot en met o_7.

Anderzijds maakt een virtuele functie zijn opwachting, namelijk *apply*. Deze functie is gedefinieerd als *apply*(e, D), waarbij e een BSL-expressie voorstelt en D een dataspace, en 'voert e uit op D'. Het resultaat van *apply* is een verzameling van *object ID's* (zie section §3) uit D die 'voldoen' aan e (zie section 3.1.1.2). Stel daarbij de volgende gelijkheid, namelijk dat apply(e, D) = apply(e_1, D) $\cap$ apply(e_2, D) indien e een BSL-expressie is met $e = e_1$ *AND* e_2.

Beschouw de volgende BSL-expressies:

$$
\begin{aligned}
e_1 &:= \text{(name: Schumacher) AND (setting: \{race, victory\})} \\
e_2 &:= \text{(name: } \neg\text{Vettel)} \\
e_3 &:= \text{(setting: victory) EXCEPT (team: Ferrari)}
\end{aligned}
$$

Dan zijn de resultaten als volgt:

$$\begin{aligned} apply(e_1,\, D) &= o_1 \\ apply(e_2,\, D) &= \{o_1,\, o_3,\, o_4, o_5\} \\ apply(e_3,\, D) &= o_3 \end{aligned}$$

3.1.1.4 Uitdrukbaarheid BSL-queries zijn additief van aard wat wil zeggen dat ze object per object kunnen geëvalueerd worden. Neem even de volgende query in de context van de voorbeelddataspace uit figure 5 als voorbeeld: "Verkrijg alle objecten die een unieke 'name' hebben.". Deze query is niet additief en bijgevolg niet definieerbaar in BSL. De taal beschikt dus niet over de mogelijkheid om relaties tussen verschillende objecten uit te drukken, want of een object al dan niet deel uitmaakt van de resultaten van een BSL-expressie hangt enkel en alleen af van dat object zelf en niet van de aan- of afwezigheid van andere objecten. Met andere woorden: in een BSL-expressie kunnen geen joins voorkomen. De ietwat minimalistische syntax van BSL gaat dus ten koste van uitdrukkingskracht die bij standaard select-project-join queries bijvoorbeeld wel sterk aanwezig is.

3.1.2 ASL

In het voorgaande stuk wordt BSL geïntroduceerd als zijnde een querytaal voor attribute-value dataspaces en zoals in section 3.1.1.4 vermeld, ontbreekt het de taal aan uitdrukkingskracht, namelijk door het onvermogen om objecten te joinen. In die context ziet Associative Search Language (of ASL) [5] het levenslicht. ASL is immers een extensie van BSL in die zin dat het een extra operator die toelaat om aan elkaar gerelateerde objecten te verkrijgen, toevoegt aan BSL. De operator in kwestie heet *link* en laat dus toe om impliciete joins uit te drukken. Net als bij de introductie van BSL zullen ook de syntax en semantiek achtereenvolgens toegelicht worden, maar dan hier enkel voor de link-operator, omdat dat de enige toevoeging is van ASL bovenop BSL. Eindigen doen we met enkele voorbeelden.

3.1.2.1 Syntax Vermits ASL louter een uitbreiding van BSL is, neemt ASL de syntax van BSL uiteraard over. Dit betekent dat het BNF-diagram uit figure 4 ook geldt voor ASL. De extra syntax die ASL introduceert, wordt voorgesteld door het BNF-diagram in figure 6.

```
<eqrel> ::= "eq-attr" | "eq-val" | "eq"
<link> ::= "link(" <keyword> <eqrel> <keyword> ")(" <asl_expression> ")"
<asl_expression> ::= <bsl_expression>
                   | <link>
                   | <asl_expression> <boolean_operator> <asl_expression>
```

Figure 6 – Syntax van ASL in een BNF-diagram dat als uitbreiding van het diagram in figure 4 moet gezien worden.

Zoals ingeleid vormt de link-operator de enige nieuwigheid en het is dan ook die operator die in het vervolg van deze sectie zal besproken worden.

Link Stel om aan te vangen deze veralgemeende vorm van de link-operator:

$$\text{link } (k_1 \text{eqrel } k_2)(e)$$

met daarbij:

- k_1 en k_2 zijnde keywords zoals gedefinieerd in section 3.1.1.1;
- e zijnde tot en met een volledige ASL-expressie (het is dus perfect mogelijk om ASL-expressies te nesten);
- *eqrel* (wat staat voor 'equality relation') zijnde een uitdrukking van een gelijkheid-scriterium waaraan k_1 en k_2 moeten voldoen.

Eqrel

Zoals de inleiding van ASL reeds aangeeft, laat de link-operator toe om een impliciete join uit te drukken. Het is dit eqrel-element dat daarbij een *join condition* specifieert tussen de keywords k_1 en k_2. Puur syntactisch leidt dat in een key-value dataspace tot drie mogelijke constanten die als waarde voor eqrel kunnen gelden, want er bestaan maar drie manieren om twee keywords inhoudelijk met elkaar te vergelijken:

1. op basis van de inhoud van de key (*eq-attr*);
2. op basis van de inhoud van de value (*eq-val*) en
3. beide (*eq*).

Stel twee keywords $k_1 = (a{:}\ v)$ en $k_2 = (b{:}\ w)$, dan kan het bovenstaande uitgedrukt worden met deze drie algemene vormen:

$$
\begin{aligned}
(a\text{: } v) \text{ eq-attr } (b\text{: } w) &\iff a = b \\
(a\text{: } v) \text{ eq-val } (b\text{: } w) &\iff v = w \\
(a\text{: } v) \text{ eq } (b\text{: } w) &\iff a = b \text{ en } v = w
\end{aligned}
$$

3.1.3 Semantiek

Link en eqrel Neem opnieuw even de algemene vorm van de link-operator, zijnde link $(k_1 \text{eqrel } k_2)(e)$. De betekenis hiervan is als volgt: "Valideer alle objecten *o* waarvoor er een object *o'* bestaat zodat *o'* in het antwoord van *e* zit én zodat *o* en *o'* *gelinkt* zijn door de eqrel. Twee objecten (*o* en *o'*) zijn door een eqrel gelinkt als:

- $\exists$ (a, v) $\in$ o: (a, v) voldoet aan k_1 én
- $\exists$ (a', v') $\in$ o': (a', v') voldoet aan k_2 én
- de eqrel tussen (a, v) en (a', v') geldt, wat wil zeggen dat er aan de gelijkheidsvoorwaarde(n) van eq-attr, eq-val of eq voldaan is.

Stel ter voorbeeld hiervan het volgende.

$$e := \text{link((name: Nick) eq-val (name: } *))((\text{setting: race}))$$

Dan dient *e* gelezen te worden als: "Verkrijg alle objecten *o* uit een dataspace met een attribute gelijk aan 'name' en een value gelijk aan 'Nick' waarvoor er minstens één ander object *o'* bestaat met ook diens attribute gelijk aan 'name' en diens value gelijk aan de value van o (hier: 'Nick'). Ook moet *o'* over een attribute beschikken die gelijk is aan 'setting' en diens value gelijk aan 'race'.".

3.1.4 Voorbeelden

Beschouw opnieuw de voorbeelddataspace uit figure 5 en de virtuele functie *apply* van de vorm apply(*e*, *D*) zoals beschreven in section 3.1.1.3. Stel de volgende vier expressies als zijnde ASL-expressies

$$
\begin{aligned}
e_4 &:= \text{link((name: } *) \text{ eq-val (name: } *))((\text{setting: } \{\text{race, victory}\})) \\
e_5 &:= \text{link((}*\text{: Ferrari) eq-attr (}*\text{: Red Bull))((name: } \neg\text{Mika Hakkinen))} \\
e_6 &:= \text{link((}*\text{) eq (}*\text{))((}*\text{))} \\
e_7 &:= \text{link((}*\text{: Michael Schumacher) eq-attr (}*\text{: Mark Webber))(} \\
&\qquad \text{link((}*\text{: race) eq-attr (}*\text{: presentation))((year: } \neg 2007)))
\end{aligned}
$$

De resultaten zijn dan als volgt:

$$
\begin{aligned}
apply(e_4,\ D) &= \{o_2,\ o_3,\ o_4,\ o_6\} \\
apply(e_5,\ D) &= \{o_1,\ o_2,\ o_4,o_5\} \\
apply(e_6,\ D) &= \{o_1,\ o_2,\ o_3,\ o_4,\ o_5,\ o_6,\ o_7\} \\
apply(e_6,\ D) &= \{o_1,\ o_2,\ o_3,\ o_4,\ o_5,\ o_6,\ o_7\}
\end{aligned}
$$

4 NoSQL

Relationele database management systemen (RDBMS) nemen al jarenlang resoluut de koppositie in wat betreft de opslag van gestructureerde data, zowel op het web als bij commerciële applicaties. De in 1970 verschenen paper "A relational model of data for large shared data banks", geschreven door Codd [7] leidde het tijdperk in van gegevensopslag op basis van relationele calculus en benaderbaar door middel van SQL. Vaak worden relationele databasesystemen nog steeds aanzien als de enige manier om gegevens op te slaan en die op een consistente manier voor meerdere clients terzelfdertijd beschikbaar te maken. Verschillende alternatieven hebben zich doorheen de jaren aangeboden, zoals bijvoorbeeld object databases of XML, maar geen enkele kon RDBMS's naar de kroon steken qua populariteit. Sterker nog: RDBMS's slorpten zulke alternatieve technologieën soms op, bijvoorbeeld door ondersteuning te bieden voor het opslaan van XML binnen een database. Een ander lot beschoren waren bijvoorbeeld OLAP [12] en stream processing, want die evolueerden in een product dat enkel gebruikt wordt voor een heel specifiek doeleinde.

Met de opkomst van het internet door de jaren heen is ook de hoeveelheid data die wereldwijd gegenereerd wordt, sterk gestegen. Die groei is vooral de laatste jaren explosief: in mei 2013 werd berekend dat 90% van alle data die ooit bestaan had tot op die dag, ontstaan is in een periode van slechts twee jaar voordien (`http://www.sintef.no/home/Press-Room/Research-News/Big-Data--for-better-or-worse/`). Grote internetgerelateerde bedrijven, zoals Facebook, Google, Yahoo en Amazon, komen rechtstreeks in contact met die enorme hoeveelheden opgeslagen data. Het probleem dat zij indertijd ondervonden was dat de conventionele RDBMS's niet geschikt waren voor gebruik in (grote) computerclusters wegens een gebrek aan schaalbaarheid en effectieve parallellisatie met daarbij ook nog eens hoge onderhoudskosten [10]. Dat probleem heeft geleid tot het ontstaan van de 'Big Data Movement', een beweging die er naar streefde om data management-oplossingen te ontwikkelen die weliswaar minder functionaliteit zouden voorzien dan de traditionele RDBMS's, maar wel van nature uit goed overweg kunnen met gedistribueerde systemen en parallellisme. Een belangrijke opmerking daarbij is dat die gedachtenstroom eigenlijk helemaal niet nieuw is. Reeds in 1965 bestond het idee en een implementatie van een niet-relationeel databasemodel [9] (MultiValue), maar het is pas sinds enkele jaren dat de term 'NoSQL' (wat staat voor 'Not only SQL') echt als buzzword gebruikt wordt voor de verzameling van niet-relationele databasetechnologieën. De belangrijkste kenmerken van de denkpiste [6] achter NoSQL-databases zijn:

- "Vermijd onnodige complexiteit": RDBMS's bieden een omgeving aan die strikt

gecontroleerd en onderhouden wordt (ACID), maar die functionaliteit is misschien overbodig. Bijvoorbeeld Adobe ConnectNow bewaart drie onafhankelijke kopies van de dezelfde gebruikerssessie in het geheugen, zonder dat daar consistentiecontroles op uitgevoerd worden of die sessie persistent opgeslagen moet zijn.

- "Hoge verwerkingscapaciteit": sommige NoSQL-databases beschikken over een hogere verwerkingscapaciteit dan traditionele RDBMS's. Een duidelijk voorbeeld hiervan is Hypertable, een kolomgebaseerde opslagtechnologie afgeleid van Google Bigtable die toelaat om een miljard gegevenscellen per dag op te slaan [14].

- "Horizontale schaleerbaarheid" en "lage kost om een databasecluster te creëeren": NoSQL databases zijn ontworpen met horizontale schaleerbaarheid in het achterhoofd, in tegenstelling tot RDBMS's. Machines kunnen toegevoegd worden aan een NoSQL-database-instantie of eruit wegvallen (door een crash bijvoorbeeld) zonder dat er operationele ingrepen vereist zijn.

- "Vermijd dure Object-Relational Mapping": NoSQL databases slaan data op door gebruik te maken van een simpele structuur, bijvoorbeeld key-value of documenten. Software die geen gebruik maakt van ingewikkelde datastructuren hebben dus geen nood aan de relationele datastructuur die een RDBMS aanbiedt.

- "De "one size fits all" denkpiste is verkeerd": de immer toenemede hoeveelheid gegevens die een database moet kunnen opslaan en de wens dat die data sneller en sneller verwerkt moet kunnen worden heeft zonder dat data-integriteit een absolute eis is, leidde tot het inzicht dat RDBMS's niet langer de enige en meteen ook ideale oplossing bieden.

- "Programmeertalen en frameworks abstraheren de interactie met relationele databases": door gebruik te maken van Object-Relation Mapping (ORM) bieden programmeertalen een manier aan om het gebruiken van rauwe SQL onnodig te maken. NoSQL databases bieden API's aan waarin enkel gegevensstructuren aangewend worden die rechtstreeks te gebruiken zijn in programmeertalen zoals bijvoorbeeld key-value-structuren, documenten en grafen.

Met het gaandeweg uitbreiden van de NoSQL *feature set* met o.a. transactions en krachtige querytalen (bijvoorbeeld UnQL, zie section 4.7 verderop) en tegelijkertijd het ontstaan van *parallel RDBMS*'s [15], waarbij een database opgesplitst wordt en verspreid over een cluster (ook wel *database sharding* of *partitioning* genoemd), is het verschil tussen beide technologieën in de praktijk aan het vervagen.

Wat volgt is de opsomming van vier prominente niet-relationele databasetechnologieën met telkens hun belangrijkste eigenschappen, een vermelding van enkele populaire implementaties en een interpretatie van hoe de technologie gemodelleerd kan worden door gebruik te maken van een attribute-value dataspace. Dit stuk eindigen gebeurt met een korte introductie van MapReduce, een high-level programmeersysteem voor databaseprocessen dat dankzij zijn hoge parallelliseerbaarheid toepasbaar is op NoSQL-databases.

4.1 Key-value Store

Een eerste NoSQL-databasetechnologie heet 'key-values store' en is qua datastructuur erg eenvoudig en daarom gemakkelijk implementeerbaar. Eenvoudig, omdat een key-value store gebruik maakt van een hash table met daarin paren van unieke keys en values. Een value zijnde een pointer naar een hoeveelheid gegevens (zie figure 7 voor een voorbeeld). De key zelf en de gegevens waarnaar een value wijst, zijn in de praktijk vaak louter bytestrings. De eenvouidige constructie laat overigens toe dat een key-value store gemakkelijk uit te breiden valt.

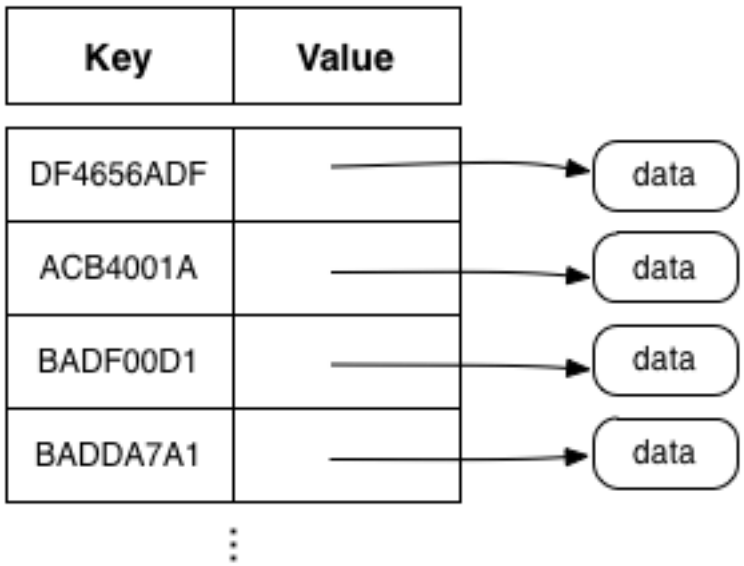

Figure 7 – Voorbeeld van een key-value store.

4.1.1 Toepassing

Een key-value store levert typisch een extreem performante *lookup table* op, die omwille daarvan meestal gebruikt wordt in een cachingsysteem. Om die performantie in de hand

te werken, is het wel zo dat de gegevens in een key-value store niet door een schema gedefinieerd kunnen zijn, zoals bijvoorbeeld wel het geval is bij traditionele relaties in de context van RDBMS's.

4.1.2 Implementaties

Bekende implementaties van key-value stores zijn BerkeleyDB (`http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`) en Amazon SimpleDB (`http://aws.amazon.com/simpledb/`).

4.2 Column Store

Een tweede categorie van NoSQL-databases heet 'column store' en is qua datastructuur ietwat complexer dan key-value stores. Een column store is een lijst van 'kolommen' waarbij elke kolom voorgesteld worden door de volgende drie elementen:

- een (unieke) naam;
- de eigenlijke inhoud (string(s), binair(e) bestand(en) ...) en
- een tijdsaanduiding waardoor bijvoorbeeld de meest recente inhoud opvraagbaar wordt. Merk op dat er door de toevoeging van een tijdsaanduiding aan elke 'inhoud' een lichte variant van versiebeheer ontstaat.

Een weergave hiervan is terug te vinden in figure 8. Deze figuur wekt door het gebruik van een tweedimensionale tabel ten onrechte de indruk op dat er een sterke gelijkenis bestaat tussen een relationele tabel en een column store, maar dat is slechts schijn. In een relationele database bevat elke rij van een relationele tabel een invulling voor elke 'kolom', zijnde een attribuut van het gebruikte relationele model. Bij column stores is het concept van een tabel eigenlijk zo goed als onbestaande. Zo is het perfect mogelijk dat een kolom onbestaande is voor een of andere rij, terwijl de rij er net onder over meerdere kolommen beschikt. Een kolom kan tevens ook lid zijn van een 'column family', waarover later meer.

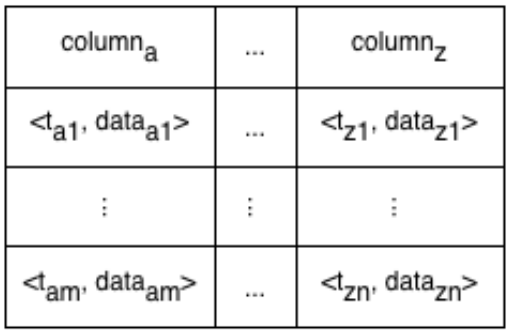

Figure 8 – Algemene vorm van een column store.

Column family Het is perfect mogelijk dat meerdere kolommen van een column store 'samenhoren' en gebundeld worden in wat een 'column family' heet. Die bundeling gebeurt aan de hand van een vooropgestelde syntax, namelijk family:qualifier. In het voorbeeld van hieronder (figure 9) is een fragment terug te vinden van een BigTable, Googles implementatie van een column store [16], waarin twee zulke families terug te vinden zijn: *contents* en *anchor*. In die voorbeeldstore wordt aangegeven vanop welke website naar een andere website gelinkt wordt door middel van een HTML-anchor. Concreet in het voorbeeld is op de eerste rij uitgedrukt dat vanop de websites *hln.be* en *4chan.org* naar *www.reddit.com* gelinkt wordt met als anchor tekst respectievelijk 'Reddit' en 'reddit.com' op de tijdstippen t_5 en t_4. De inhoud van de website naar waar gelinkt wordt (hier dus *www.reddit.com*), zit in de kolom 'contents' vervat, telkens gepaard met een tijdsaanduiding, waardoor het mogelijk wordt om 'terug te gaan in de tijd' en de inhoud van de website op het moment dat er naar gelinkt werd, te kunnen reconstrueren.

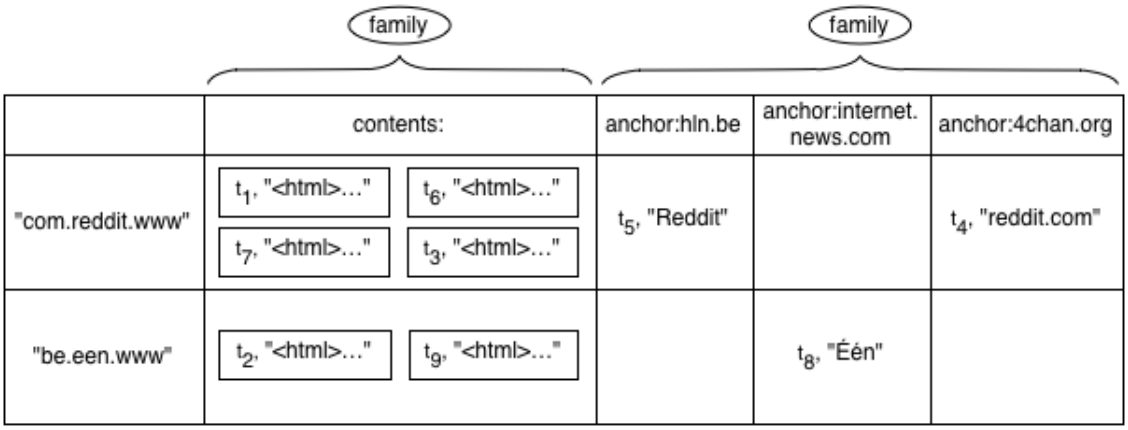

Figure 9 – Voorbeeld van een column store met daarin twee families: *contents* en *anchor*.

4.2.1 Toepassing

Column stores worden aangewend in situaties waarin enorme hoeveelheden gegevens gedistribueerd opgeslagen zijn en verwerkt moeten worden, zoals bij gedistribueerde bestandssystemen of bij *web crawlers*.

4.2.2 Implementaties

Bekende voorbeelden van column stores zijn Bigtable [16], HBase (`http://hbase.apache.org/`) en Cassandra (`http://cassandra.apache.org/`).

4.2.3 Performantievergelijking met row-stores

Row-stores zijn qua data-opslag fundamenteel verschillend van column-stores. Daar waar row stores elke rij of record apart opslaan, voorzien column-stores een record per kolom of attribuut. Voor een voorbeeld, zie figure 10. Welke invloed heeft dit verschil op de performantie waarmee een traditionele aggregatie (bvb. een som van een numeriek attribuut) of join uitgevoerd wordt? De performantie van die beide gevallen is uitdrukbaar in het aantal 'store accesses': het aantal keren dat er gegevens uit de store moet opgehaald worden vooraleer het resultaat bekomen kan worden.

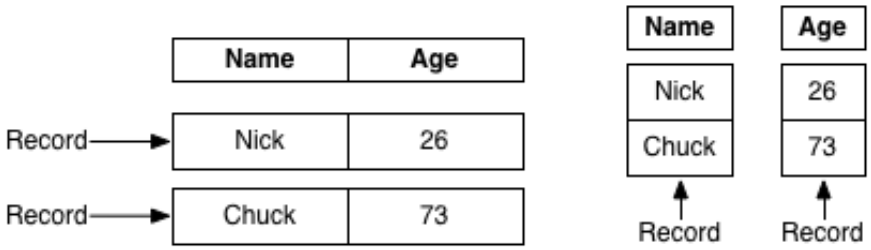

Figure 10 – Links: een voorbeeld van een row-store, waarvan elke record apart opgeslagen is. Rechts: dezelfde gegevens, maar nu weergegeven in een column-store. Elke kolom is apart opgeslagen.

4.2.3.1 Aggregatie Stel het volgende voorbeeld: "Geef de som van de leeftijd van alle personen." dat uitgevoerd moet worden op het voorbeeld van figure 10. In het geval van een row store moet elk record één keer ingelezen worden, omdat de leeftijd van elke persoon onderdeel is van een apart opgeslagen record. Bij een column store volstaat het inlezen van het record dat de kolom bevat met daarin de leeftijden van alle personen. Een aggregatie op een attribuut bij een column store kost dus één store access, en bij een row store één access per rij. Er kan dus gezegd worden dat een aggregatie bij een column store performanter is dan bij een row-store.

4.2.3.2 Joins In het geval van joins is de performantie afhankelijk van het aantal keren dat een tuple uitgelezen moet worden om zo bijvoorbeeld een vergelijking van een attribuut van die tuple te doen. Merk op dat bij een row store een tuple ophalen een goedkope operatie is, omdat een record uitlezen onmiddellijk een volledig tuple oplevert. Voor column stores betekent het ophalen van een tuple echter dat die tuple eerst nog moet samengesteld worden, wat per attribuut van die tuple het inlezen van een record inhoudt.

Beschouw opnieuw de voorbeeldgegevens uit figure 10, maar dit keer uitgebreid met een tabel die een persoon linkt aan een land. Die uitbreiding levert de situatie op zoals weergegeven in figure 11.

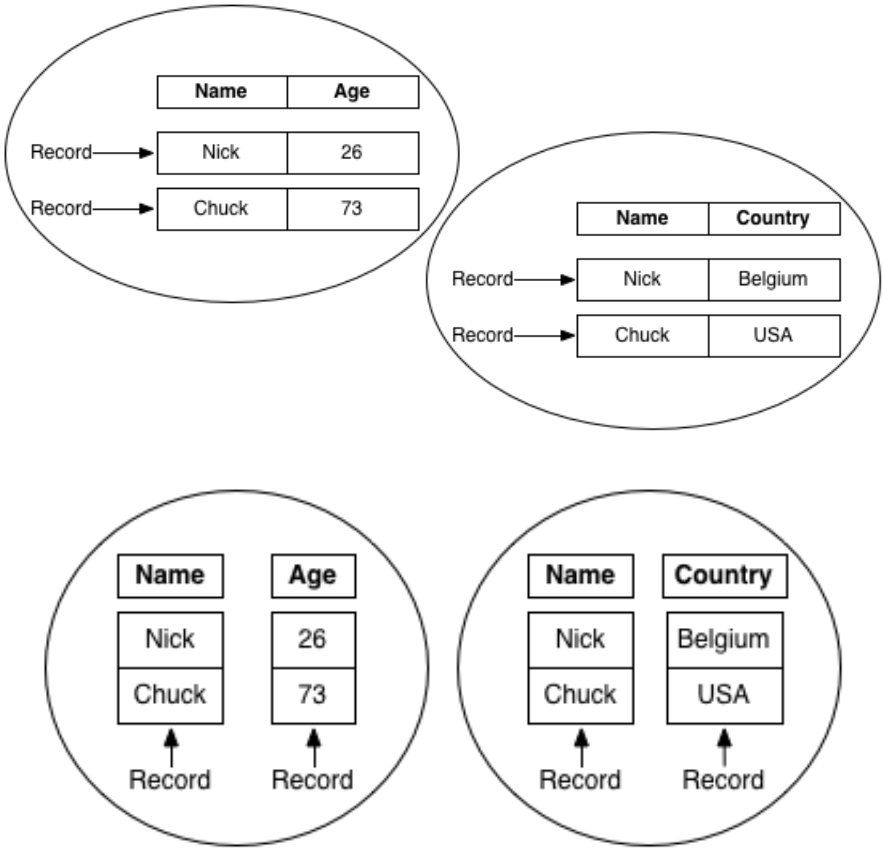

Figure 11 – Voorbeeldgegevens uit figure 10, maar dan uitgebreid met een tabel die een persoon linkt aan een land. Boven: de uitbreiding voor de row store. Beneden: de uitbreiding voor de column store.

Stel voor deze situatie de volgende query: "Geef de leeftijd van een persoon die in België woont", uit te voeren op de uitgebreide stores zoals weergegeven in figure 11. Deze query houdt niet toevallig in dat er tussen beide tabellen een join moet gebeuren. Voor een row store volstaat het om alle tuples uit de rechtertabel (die een naam linkt aan een woonplaats / land) te filteren op het attribuut 'country' om vervolgens het 'name'-attribuut van de resultaten van die selectie te vergelijken met de tuples uit de linkertabel. Een join uitvoeren op een column store houdt ook voor een column store in dat er tijdens dit proces tuples moeten geconstruuerd worden, maar omdat de attributen van een tuple verspreid zitten in meerdere records, is een tupleconstructie voor een column store een veel duurdere operatie dan bij row stores. Om het aantal tupleconstructies bij een join van een column store te beperken, bestaan er twee technieken: Early Materialization (EM) en Late Materialization (LM).

Early Materialization Early Materialization gaat als volgt te werk:

1. Creëer een materialized view (`http://uhesse.com/2009/07/08/brief-introduction-into-materialized-views/`) waaraan per record één tuple toegevoegd kan worden (dus zoals bij een row-store);
2. Voeg een tuple toe aan de materialized view zodra een aan eerste joinconditie voldoet;
3. Verwijder een tuple uit de materialized view als een volgende conditie van de joinoperatie voor die tuple faalt.

Het voordeel van de EM-techniek is dat het aantal tupleconstructies gereduceerd wordt: een tuple wordt slechts één keer samengesteld en nadien in de materialized view opgeslagen als de tuple voldoet aan een eerste joinconditie. Het nadien uitlezen van die tuple uit die view om er overige joincondities op te testen is significant sneller dan die tuple eerst opnieuw te moeten samenstellen. Nadelig aan deze aanpak is dat de kans bestaat dat er overbodige tupleconstructies hebben plaatsgevonden, want een tuple wordt reeds bij de eerste succesvolle joinconditie toegevoegd aan de materialized view. Als een volgende joinconditie faalt voor de tuple in kwestie, wordt die tuple alweer verwijderd uit de view en was de constructie ervan eigenlijk overbodig in de eerste plaats.

Late Materialization Late Materialization biedt een oplossing voor het performantieprobleem dat Early Materialization met zich meedraagt, nl. de kans op overbodige tupleconstructies, door zo lang mogelijk te wachten met de tupleconstructie. Een overzichtje van de werking ervan:

1. Creëer een materialized view waaraan per record één attribuut / kolom toegevoegd kan worden (dus zoals bij een column-store);
2. Voeg enkel die attributen / kolommen toe aan de materialized view die nodig zijn om de joinoperatie uit te kunnen voeren;
3. Verwijder die elementen uit een kolom waarvoor een joinconditie gefaald heeft.
4. Construeer tuples uit de originele database, gegeven de overgebleven attributen in de materialized view.

Deze werking houdt in dat er een minimum aan tupleconstructies plaatsvindt en is daarom performanter dan die van EM.

4.3 Document Store

De derde categorie van NoSQL-databases heet 'document store' en maakt net als key-value stores intensief gebruik van een key-value-structuur. Het grote verschil zit in de manier waarop die key-value-structuur belichaamd wordt. Herinner in het geval van de key-value store het gebruik van een hashtable. De structuur bij document stores zit vervat in 'virtuele documenten' die een hiërarchische structuur toelaten. Voor een voorbeeld hiervan, zie figure 12. Merk aan deze figuur op dat de informatie binnen zo'n document niet alleen genest kan zijn, maar dat er ook geen verplichtingen bestaan in verband met de aanwezigheid of waarde van de keys.

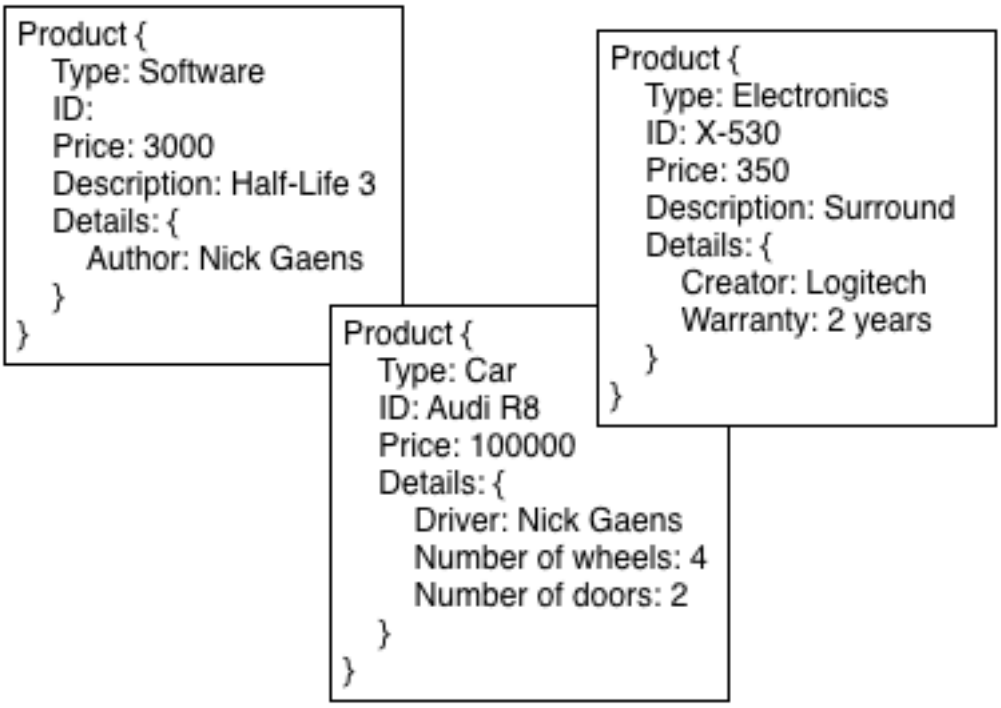

Figure 12 – Voorbeeld van een document store.

4.3.1 Toepassing

De afwezigheid van inhoudelijke verplichtingen (oftewel: document stores kunnen overweg met onvolledige data) leidt tot een robuust karakter dat document stores typeert. De geleverde robuustheid komt echter met een prijs, namelijk dat de query-mogelijkheden van dit type NoSQL-database er negatief door beïnvloed worden. Er bestaat namelijk geen gestandardiseerde manier in de vorm van een syntax om een document store te

queryen. Gebruik maken van een document store in een software-implementatie betekent dus dat de software zelf bijvoorbeeld joins zal moeten verwezenlijken.

4.3.2 Implementaties

Bekende implementaties van document stores zijn CouchDB (`http://couchdb.apache.org/`) en MongoDb (`http://www.mongodb.org/`).

4.4 Graph Database

Deze vierde vorm van NoSQL-databases heet 'graph database' en is qua concept verschillend van de drie voorgaande. Graph databases maken, zoals de naam al doet vermoeden, gebruik van een graafrepresentatie voor hun gegevens. Aan alle knopen en bogen van deze graaf kan een lijstje van eigenschappen gehangen worden, bijvoorbeeld in de vorm van key-value-paartjes. In het geval van een boog, worden dan de semantiek of het gewicht van die boog simpelweg onderdeel van dat lijstje. Een voorbeeld van een graph database is terug te vinden in figure 13.

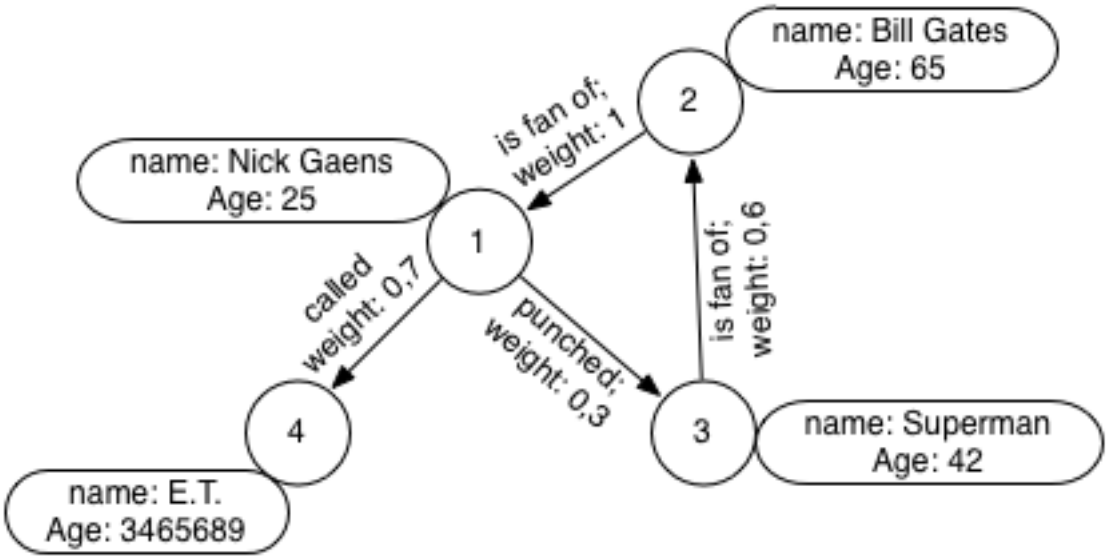

Figure 13 – Voorbeeld van een graph-database.

4.4.1 Toepassing

De eigenschappen die een graaf met zich meebrengt, zorgen ervoor dat gegevens waarvan de onderlinge relaties van belang zijn (bijvoorbeeld bij *social networking*) vaak hun weg vinden naar deze NoSQL-databasevorm. Grafen zijn als datastructuur allesbehalve nieuw en bijgevolg in het verleden intensief onderzocht. Er bestaat dan ook een ruime verzameling van uitgeëvolueerde, performante graafalgoritmes, bijvoorbeeld voor een kortste-padberekening of het zoeken van n^{de}-graads relaties.

Een negatieve eigenschap is echter dat de onderlinge verbondenheid grafen van nature ongeschikt maakt om ze in een geclusterde omgeving te herbergen.

4.4.2 Implementaties

De bekendste implementatie van een graph database is Neo4j (`http://www.neo4j.org`).

4.5 Attribute-value modellering

4.5.1 Losse datastructuur

De afwezigheid van onderlinge relaties binnen de opgeslagen gegevens is één van de beginselen van NoSQL. De gegevensstructuren van bijvoorbeeld key-value stores (zie section 4.1) en document stores (zie section 4.3) komen respectievelijk sterk en zeer sterk overeen met die van een attribute-value dataspace. Zo maken key-value stores intensief gebruik van key-value paartjes die vrijwel identiek zijn aan items uit de context van attribute-value dataspaces. De 'documents' die als encapsulerend element van key-value paartjes terug te vinden zijn in document stores, vertonen dan weer sterke gelijkenissen met objects uit de context van attribute-value dataspaces. Deze twee technologieën zijn dus sterk gelijkend op attribute-value dataspaces en beschikken dusdanig over een omgeving waarin een losse datastructuur zeer zeker aanwezig is. Dat geldt echter niet voor alle NoSQL-databasetechnologieën, want bijvoorbeeld graph databases (zie section 4.4) leunen, net als SeCo en iPDMS hierboven, hevig op het gebruik van een graaf voor de structurering van gegevens. Voor graph databases geldt dus dezelfde conclusie als bij SeCo en iPDMS, namelijk dat de onderlinge verbondenheid van de gegevens noodzakelijk is en dat de datastructuur niet als los geldt.

4.5.2 Schemaloosheid

Een belangrijk basisbeginsel van NoSQL is het ontbreken van een dataschema. Dit betekent dat er zo goed als een volledige vrijheid bestaat wat betreft het type en de

inhoud van toegelaten gegevens. De enige beperking die er geldt is betreffende de vorm van de gegevens: key-value stores bijvoorbeeld laten enkel key-value paartjes toe en document stores vereisen dat key-value paartjes geëncapsuleerd zijn in een 'document'. In het geval van graph databases kunnen er extra beperkingen in de vorm van semantische validaties bestaan. Neem bijvoorbeeld een graaf die een ouderschapsrelatie voorstelt met daarin een node A die 'ouder is van' node B, maar dat B terzelfdertijd 'ouder is van' A. Zulke illegale constructies kunnen reeds op voorhand gedetecteerd en tegengehouden worden.

4.5.3 Gegevensdiversiteit

Gelijkaardig als bij SeCo geldt ook hier dat alle gegevens die met bytestrings voorgesteld kunnen worden, kandidaat zijn om in een NoSQL-database terecht te komen.

4.5.4 Key-value stores

Er zijn twee mogelijke manieren waarop een key-value store gemodelleerd kan worden door gebruik te maken van een attribute-value dataspace. De eerste is rechtoe rechtaan en houdt een structuur in die gelijkaardig is aan het voorbeeld uit figure 7. Door de pointers waarmee door values naar gegevens gewezen worden weg te halen en die gegevens rechtstreeks onder te brengen in één allesomvattend object, ontstaat de opbouw zoals weergegeven in figure 14.

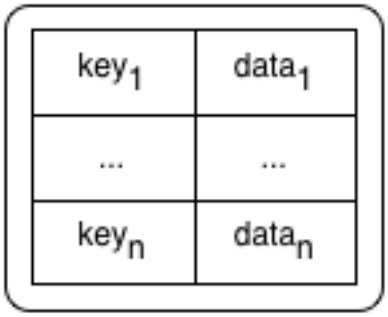

Figure 14 – Key-value store gemodelleerd aan de hand van een AV-dataspace die één groot object herbergt.

Een tweede aanpak bestaat erin om voor elk key-value paar van de store een apart dataspace object te maken. Elk object maakt dan gebruik van twee vaste attributen, namelijk 'key' en 'value'. Zie figure 15 voor een weergave van deze opbouw.

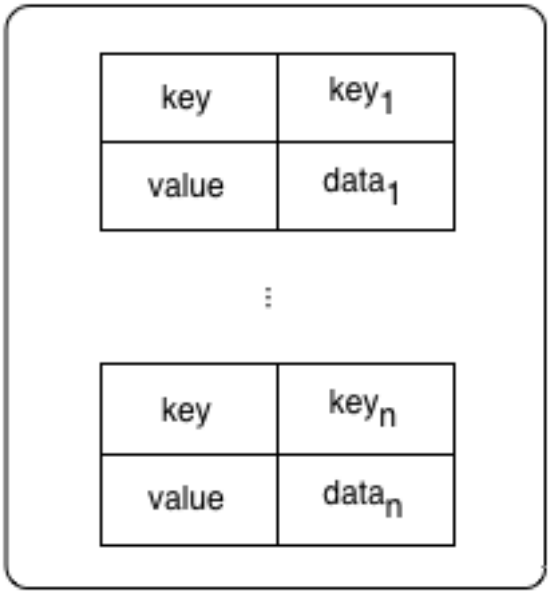

Figure 15 – Key-value store gemodelleerd aan de hand van een AV-dataspace.

4.5.5 Column stores

Er bestaan twee mogelijkheden waarop column stores door middel van een attribute value dataspace gevormd kunnen worden. De eerste mogelijkheid is om voor elke combinatie van row_id, family en column een apart object met telkens vier vaste attributen aan te maken. Een sterk vergelijkbare aanpak werd reeds eerder bij key-value stores geïntroduceerd en net als daar geldt ook hier dat deze aanpak kan leiden tot een situatie waarin een groot aantal objecten kunnen ontstaan. Voor een voorbeeldweergave van deze aanpak, zie figure 16.

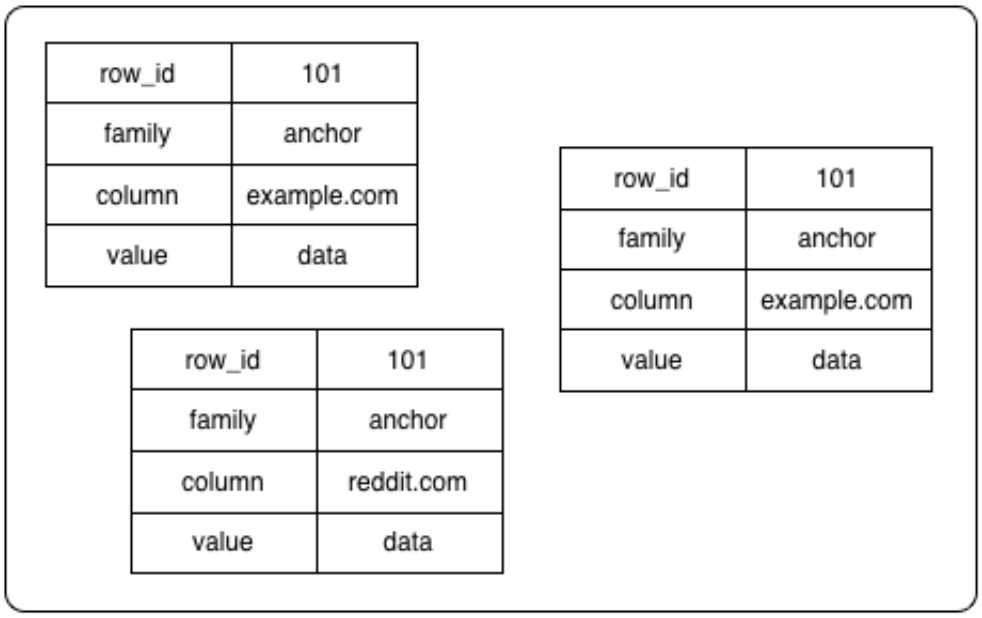

Figure 16 – Column store gemodelleerd met behulp van een attribute-value dataspace waarbij elke combinatie van een row_id, family en column in een apart object gegoten is.

Een tweede aanpak bestaat erin om slechts voor elke family-waarde een apart object aan te maken. Op die manier kan het aantal benodigde objecten sterk gereduceerd worden. Voor een voorbeeld hiervan, zie figure 17.

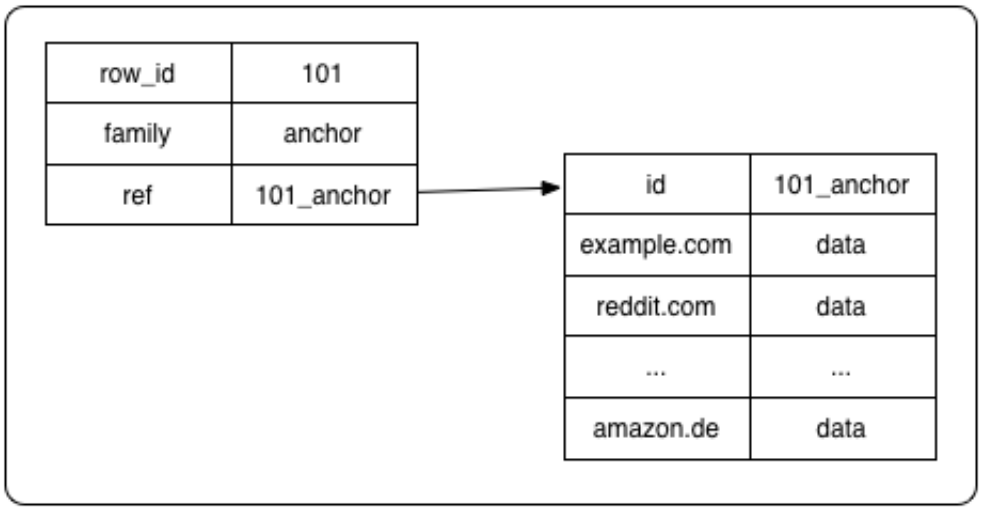

Figure 17 – Column store gemodelleerd met behulp van een attribute-value dataspace waarbij elke family een apart object beslaat.

4.5.6 Document stores

Net als bij de twee voorgaande NoSQL-databases bestaan er ook voor document stores twee mogelijkheden om gemodelleerd te worden met behulp van een attribute-value dataspace. De eerste mogelijkheid houdt in dat er voor elk document een object met een vaste verzameling attributen aangemaakt wordt. Uiteraard geldt ook hier, net als bij de twee voorgaande soortgelijke gevallen, dat er een groot aantal objecten kunnen ontstaan. Voor een voorbeeld, zie figure 18.

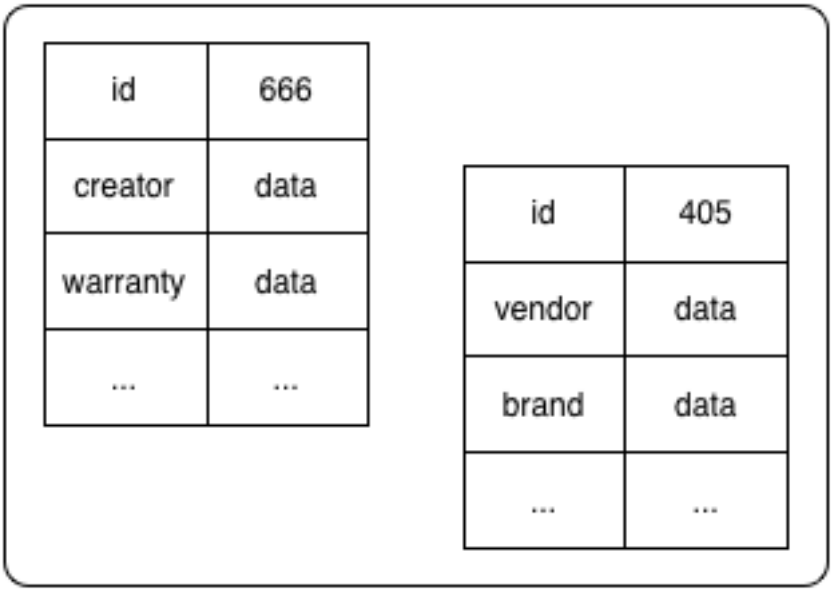

Figure 18 – Document store gemodelleerd met behulp van een attribute-value dataspace waarbij elk document door een apart object voorgesteld wordt.

Een tweede aanpak bestaat erin om, net als bij column families hierboven, gebruik te maken van geneste objecten. Het aantal objecten blijft zo beperkt. Voor een voorbeeldweergave, zie figure 19.

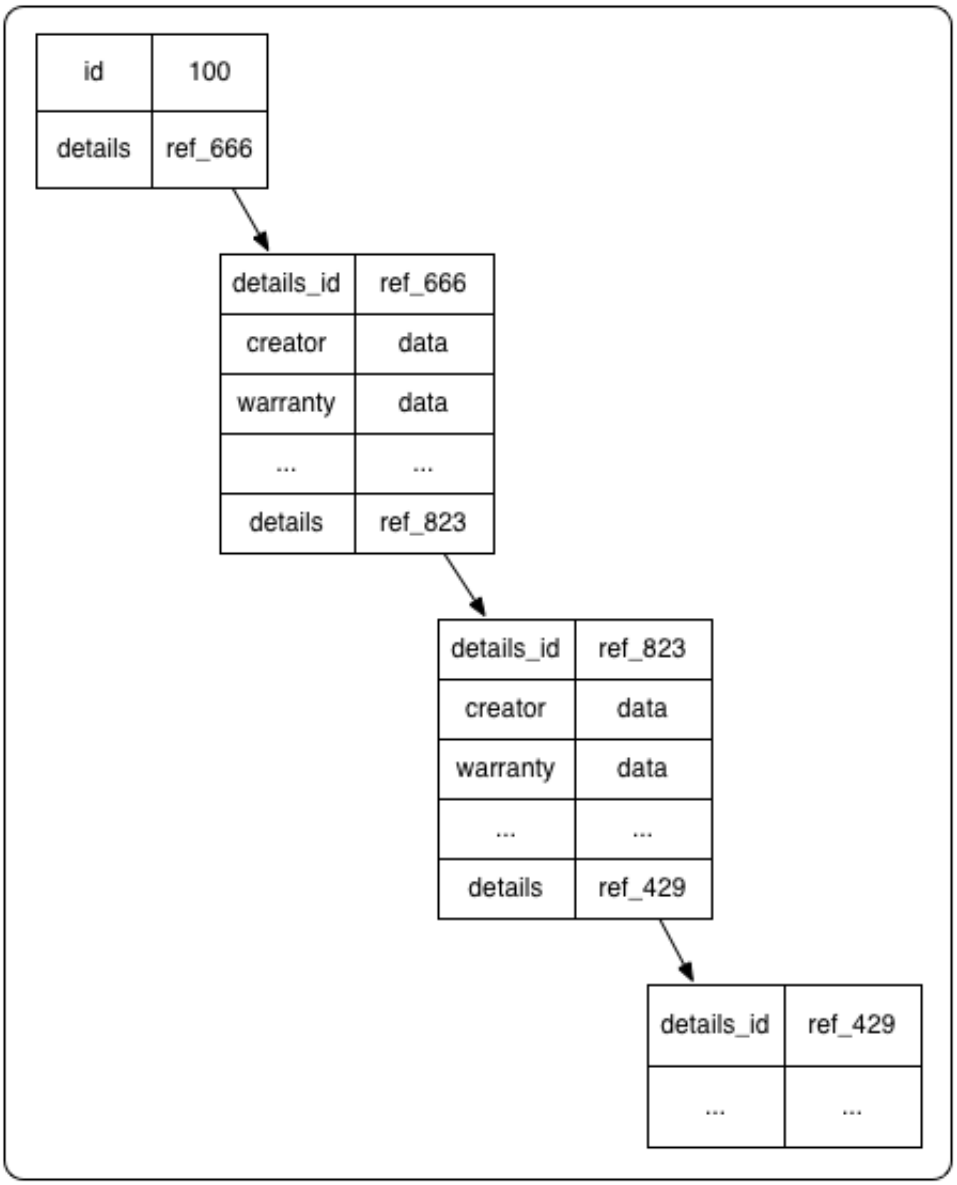

Figure 19 – Document store gemodelleerd met behulp van een attribute-value dataspace waarbij elk document door geneste objecten voorgesteld wordt.

4.5.7 Graph databases

In tegenstelling tot bij de drie voorgaande NoSQL-databases bestaat er voor graph databases slechts één aanpak om ze met behulp van een attribute-value dataspace te modelleren. Die houdt in dat voor elke knoop en voor elke boog van de graaf een

apart object aangemaakt dient te worden. Elk object beschikt dan vervolgens over een key-value-paartje dat weergeeft of een knoop of boog gerepresenteerd wordt. Voor een voorbeeld hiervan, zie figure 20.

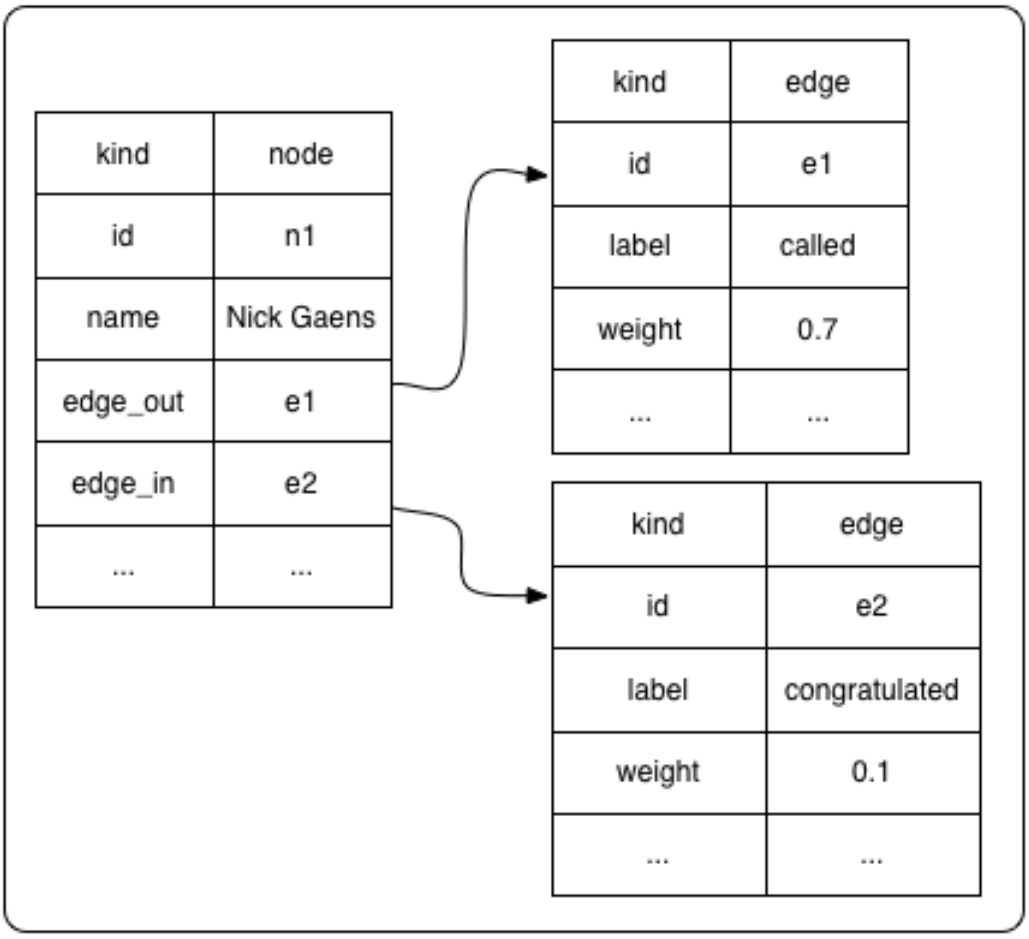

Figure 20 – Voorbeeld van een graph-database, gemodelleerd met behulp van een attribute-value dataspace.

4.6 Map-Reduce

Map-Reduce [8] is een programmeersysteem waarbij (geparallelliseerde) databaseprocessen op een simpele manier geschreven kunnen worden. Het systeem houdt in dat twee functies geïmplementeerd moeten worden, map en reduce, die dan achtereenvolgens uitgevoerd worden waarbij de tweede de output van de eerste als input gebruikt.

4.6.1 Map

De map-functie wordt als eerste van beide uitgevoerd en neemt één key-value paartje[2] als input en output vervolgens een lijst van key-value paartjes die een tussentijds resultaat vormen. Er zijn echter twee kanttekeningen die hierbij geplaatst moeten worden:

- De keys en values die de output van de map-functie vormen kunnen perfect van een andere aard zijn als de key en value van het inputpaartje.
- De keys van de output van de map-functie zijn geen *keys* zoals die in traditionele databasesystemen bekend zijn. Er kunnen namelijk meerdere paartjes in de output voorkomen met dezelfde waarde voor de key.

4.6.1.1 Voorbeeld Neem een verzameling van tekstdocumenten in beschouwing en stel als opgave om per document een lijst samen te stellen van woorden die minstens één keer voorkomen in dat document.

- De verzameling van documenten vertaalt zich in eerste instantie naar een inputverzameling van key-value paartjes waarvan de key een document ID i is en de value het overeenkomstige document d.
- De map-functie neemt vervolgens zo'n paartje (i, d) als input en gaat iteratief door d op zoek naar woorden in dat document. Merk op dat dit inhoudt dat voor elk document een aparte instantie van de map-functie kan bestaan.
- Voor elk woord w in d output de map-functie een nieuw paartje (w, i). Merk op dat hetzelfde woord w meerdere keren als key kan voorkomen in de output van alle geïnstantieerde map-functies.

4.6.2 Reduce

De reduce-functie wordt na de map-functie uitgevoerd en neemt een geaggregeerd key-value paartje uit diens output als eigen input. Zo'n geaggregeerd paartje is van de vorm (key, [value$_1$, value$_2$, ..., value$_n$]) en de value-lijst hiervan bestaat uit alle values uit het tussentijds resultaat die er in combinatie met de betreffende key in voorkomen. De output van de reduce-functie geldt tenslotte als eindresultaat van het databaseproces.

[2] De input van de map-functie kan eigenlijk van eender welke vorm zijn, maar omdat de uitvoer van het map-reduce systeem altijd een verzameling van key-value paartjes is, is het veilig om aan te nemen dat een verzameling van dezelfde vorm als invoer dient.

4.6.2.1 Voorbeeld

Voortbordurend op het voorbeeld uit 4.6.1, bestaat het tussentijds resultaat uit paartjes van de vorm $(w, [i_1, i_2, \ldots, i_n])$ met key w zijnde een woord en value een lijst van document ID's voor elke keer dat w voorkomt in het gerefereerde document. De reduce-functie zal in dit voorbeeld deze lijst van document ID's moeten filteren op dubbels en vervolgens sorteren. Net als bij de map-functie geldt hier dat er meerdere instanties van de reduce-functie in het leven geroepen kunnen worden, met in extremis één instantie per geaggregeerd key-value paartje. De output van alle reduce-functie-instanties wordt uiteindelijk opnieuw geaggregeerd en levert het gevraagde resultaat.

4.6.3 Parallellisatie

Het voorbeeld dat hierboven terug te vinden is, toont aan dat het map-reducesysteem toelaat om in sterke mate geparallelliseerd te worden. Een map-functie kan erbij werken op het niveau van een document, dus er kunnen zoveel instanties van de map-functies terzelfdertijd uitgevoerd worden als dat er documenten zijn. De reduce-functie op haar beurt werkt op het niveau van een woord, dus per woord in het tussentijdse, geaggregeerde resultaat kan er een instantie van deze functie uitgevoerd worden. In de praktijk zal zo'n ver doorgedreven parallellisatie echter niet snel toegepast worden, omdat bijvoorbeeld de beschikbare hardware dat niet toelaat of de overhead tijdens het opzetten van de parallellisatie de eventuele tijdswinst teniet doet.

4.7 UnQL

In een poging om een gestandardiseerde querytaal voor NoSQL-databases te ontwikkelen, is UnQL (Unstructured Data Query Language, uitgesproken als 'uncle') (`http://www.couchbase.com/press-releases/unql-query-language`) gespecifieerd. UnQL is ontworpen als superset van SQL (dus ondere andere JOIN, GROUP BY en HAVING zijn als keyword van de partij) en heeft diens syntax zo goed als volledig geadopteerd. Het opzet van dit laatste was om de overgang voor SQL-ontwikkelaars van SQL naar UnQL zo laagdrempelig mogelijk te zijn. Twee kleine voorbeelden van UnQL-queries:

1. het toevoegen van een document aan een document store:

```
INSERT INTO cool_nosql_collection VALUE {
  type:"nested",
  content: {
    content: "document object",
    docNumber:1,
    articleContent: {str:"Dataversity", str2:"Article"},
```

```
    newArticle:true
  }
};
```

2. het verkrijgen van een document uit een document store:

   ```
   SELECT FROM cool_nosql_collection
   ```

   geeft de volgende output:

   ```
   {"type":"nested",
    "content":{"content":"document object",
               "docNumber":1,
               "articleContent":{"str":"Dataversity",
                                 "str2":"Article"},
               "newArticle":true}
   }
   ```

Wat opvalt aan deze output is dat het gebruik maakt van het JSON-formaat. De reden hiervoor is dat JSON als datacontainer geen schema hanteert om data te kunnen representeren. Deze schemaloosheid is tevens één van de belangrijkste eigenschappen van NoSQL en JSON vormt dus de uitgelezen kandidaat als representatieformaat.

Ondanks dat UnQL reeds volledig gedefinieerd is, ondervindt het last van het sterk gefragmenteerde en nog steeds fluctuerende NoSQL-landschap. Niet alleen moet er voor elke populaire programmeertaal een implementatie van UnQL geleverd worden, maar die moet ook nog eens foutloos samenwerken met zoveel mogelijk onderling sterk verschillende NoSQL-databases. Claudius Weinberger, de ontwikkelaar achter ArangoDB (een multi-model NoSQL-database die als combinatie van key-value store, document store en graph database ontwikkeld is), stelt het volgende over UnQL (`http://www.arangodb.org/2012/04/07/is_unql_dead`):

> UNQL started with quite some hype last year. However, after some burst of activity the project came to a hold. So it seems, that – at least as a project – UNQL has been a failure. IMHO one of the major issues with the current UNQL is, that it tries to cover everything in NoSQL, from key-value stores to document-stores to graph-database. Basically you end up with greatest common divisor – namely key-value access. But with graph structures and also document-structures you really want to supports joins, paths or some sort of sub-structures.

Apart from all the technical and theoretical benefits of SQL and what advantages the underlying theory has to offer, the major plus from an users point of view is that it is readable. You simple can see an SQL statement – be it in C, Java, Ruby – and understand what is going on. It is declarative, not imperative. With other imperative solution [...] you need to understand the underlying syntax or language. With SQL you only need to understand English – at least most of the time.

And here I think is where UNQL is totally right. We need something similar for the NoSQL world. But it should not try to be a "fits all situation". It should be a fit for 80% of the problems. For simple key-values stores a fluent-interface is indeed enough. For very complex graph traversals a traversal program must be written.

5 Search Computing

Een antwoord vinden voor complexe zoekacties is niet eenvoudig. Ondanks dat heel erg veel informatie beschikbaar is op het internet, ontbreekt het aan softwaretools die die informatie op zo'n manier bevraagbaar en doorzoekbaar maken dat ook het antwoord vinden voor complexe zoekacties geen onoverkomelijke opdracht is. Search Computing [1] is net ontwikkeld om een oplossing te bieden voor dit probleem [13]. Specifieker: het SeCo-project richt zijn pijlen op de laatste drie fasen zoals weergegeven in figure 1: vanaf het moment dat er de intentie bestaat om naar informatie op zoek te gaan tot en met het formuleren van een vraag in de vorm van een query.

5.1 Semantic Resource Framework

De informatieruimte of gegevensverzameling die binnen het Search Computing project aangewend wordt, bestaat uit geconceptualiseerde real-world objecten van eender welke aard: 'auto', 'concert', 'brood', 'computer' enzovoorts. Deze concepten worden vervolgens onderling verbonden door een bepaalde semantiek, bijvoorbeeld 'artiest <treedt op> concert' of 'computer <voert uit> softwareprogramma'. Deze constructie heet 'Semantic Resource Framework' [1] en kan door gebruik te maken van een graafrepresentatie gemakkelijk gevisualiseerd worden. Deze graaf heet op zijn beurt 'Semantic Resource Graph' en een voorbeeld ervan kan gevonden worden in figure 21. Merk op dat deze graaf virtueel oneindig groot kan worden, omdat er talloze real-world concepten bestaan die op talloze manier onderling verbonden zijn.

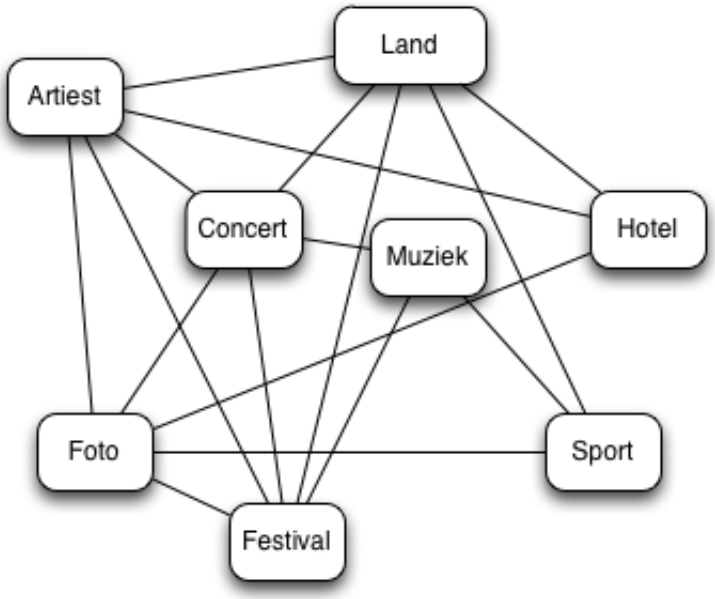

Figure 21 – Semantic Resource Graph met real-world concepten als nodes en semantische verhoudingen als edges. Bijvoorbeeld de edge tussen 'Artiest' en 'Festival' kan gelabeld worden met 'treedt op'.

5.2 Exploratory search

Gebruik makend van een of meerdere keywords om queries mee te vormen, kunnen gebruikers met behulp van een zoekrobot (bvb. Google (`http://www.google.com`) of Bing (`http://www.bing.com`) voor online en OS-geïntegreerde zoekfuncties voor offline toepassingen) vaak enorme hoeveelheden data tegen een hoge snelheid doorzoeken. Het probleem dat er echter optreedt wanneer een gebruiker vanuit het niets een booleanse expressie moet samenstellen door keywords aan elkaar te koppelen met behulp van operatoren, is dat die dat als hoogdrempelig ervaart [1]. Het Search Computing project probeert voor dit probleem een oplossing te bieden door incrementele querysamenstelling mogelijk te maken. Die querysamenstelling kan door de Semantic Resource Graph (zie figure 21) te manipuleren: aan een initieel lege query kan een keyword toegevoegd worden door nodes uit die graaf te selecteren. Dit proces heet 'exploratory search' en biedt de gebruiker dus de mogelijkheid om een zoekengine als het ware bijvragen te stellen telkens een nieuwe node geselecteerd wordt. Zie figure 22 voor een voorbeeld waarbij de afbeelding aan de linkerkant aangeeft dat de gebruiker de 'Concert'-node heeft geselecteerd. Een implementatie die deze interactie mogelijk maakt moet nu de

mogelijkheid bieden aan de gebruiker om een concert te specifiëren, maar dat valt buiten het bestek van deze thesis. In de rechterafbeelding selecteert de gebruiker in een volgende stap de 'Land'-node, omdat hij of zij informatie wenst te vergaren over een concert in een bepaald land.

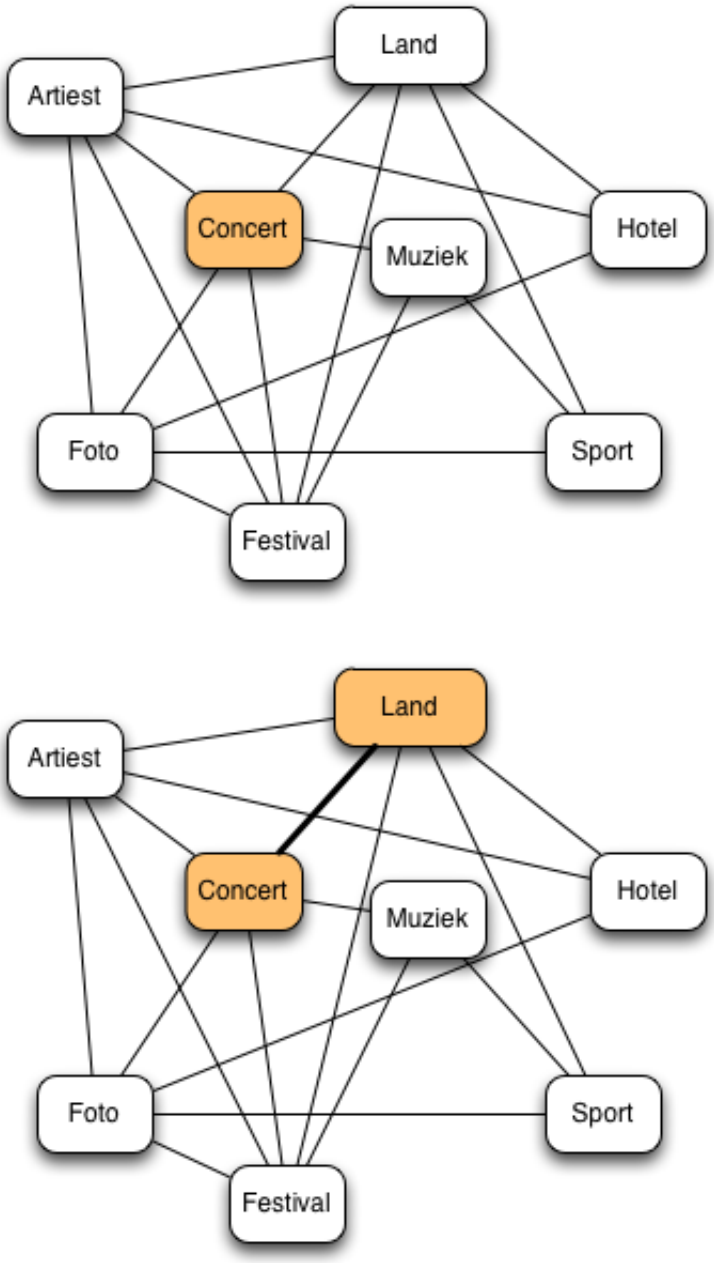

Figure 22 – Exploratory Search door manipulatie van een Semantic Resource Graph.

5.3 Attribute-Value modellering

5.3.1 Losse datastructuur

SeCo maakt intensief gebruik van een *semantic resource graph* (zie figure 21): een graaf waarvan de nodes real-world concepten voorstellen die door semantische edges met elkaar verbonden zijn. Net omdat SeCo een graafrepresentatie aanwendt om de beschikbare gegevens mee voor te stellen, kan gesteld worden dat de objecten zelf reeds relationele informatie bevatten om de graaf in kwestie mee op te bouwen. Zie bijvoorbeeld een voorbeelddataspace zoals weergegeven in figure 23. In dit geval is er daarom weinig sprake van een losse datastructuur.

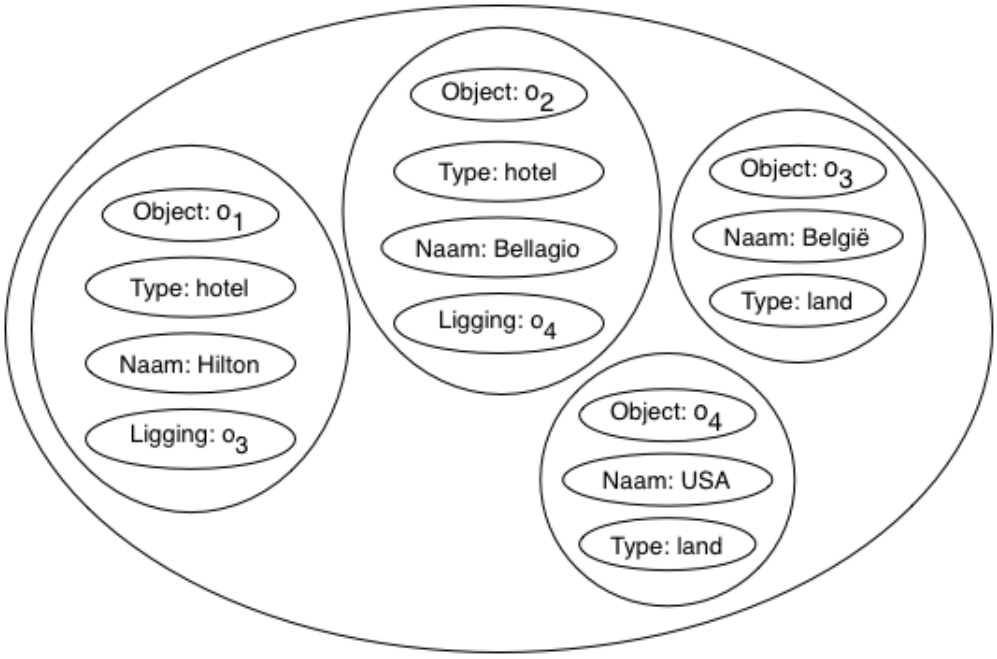

Figure 23 – Voorbeelddataspace met vier objecten op basis van de structuur uit figure 21 waarbij de objecten reeds zelf relationele informatie bevatten. Merk op o_1 en o_2 een referentie bevatten naar respectievelijk o_3 en o_4.

Anderzijds kan er ook gesteld worden dat die relationele informatie zich niet in de objecten zelf bevindt, en dat dus de betekenis van de edges op een niveau 'boven' of buiten de daadwerkelijke objecten gedefinieerd wordt. Zie ter verduidelijking hiervan de voorbeelddataspace zoals weergegeven in figure 24. De drie objecten in dit voorbeeld bevatten louter een typeaanduiding en een naam en weten dus niet of en hoe ze onderling verbonden zijn. De relationele informatie van de SeCo-graaf kan dan extern opgeslagen

worden, bijvoorbeeld in een tweede dataspace of een systeem van een andere aard.

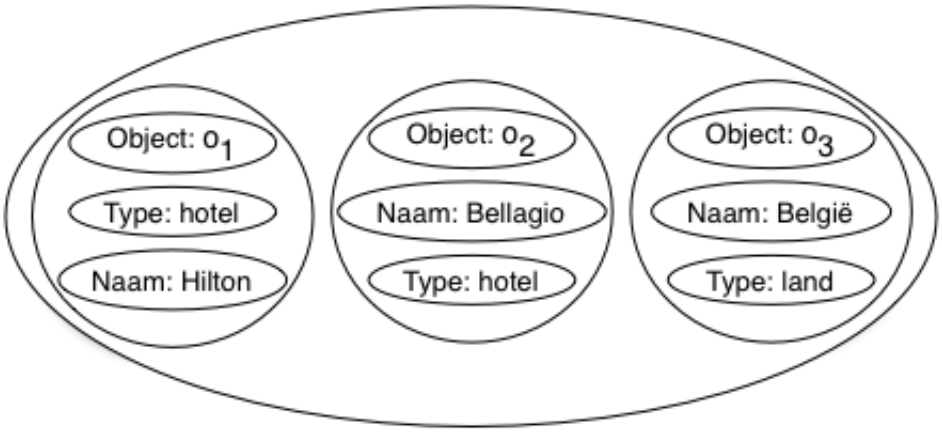

Figure 24 – Voorbeelddataspace met drie objecten op basis van de structuur uit figure 21 waarbij de objecten geen relationele informatie bevatten. Die relationele informatie moet dan via een extern systeem aangeboden worden.

Het is dus duidelijk dat SeCo, dankzij de graafrepresentatie, altijd relationele informatie met zich meedraagt en dat zulke informatie het onmogelijk maakt om een structuur op te zetten waarbij de objecten onderling geen weet hebben van hun verhoudingen.

5.3.2 Schemaloosheid

Over het gebruik van een dataschema waaraan de gegevens in een SeCo-graaf moeten voldoen vooraleer ze toegelaten worden, is in de literatuur niets bekend. Vermoedbaar is wel dat er op z'n minst een vorm van controle bestaat naar de leegheid van ingevoerde data. Het is immers zinloos om bijvoorbeeld items toe te laten waarvan de key en/of value leeg is. Ook zou in de opstelling zoals weergegeven in figure 23 de aanwezigheid van een item met als key 'Type' voor elk object verplicht kunnen zijn.

5.3.3 Gegevensdiversiteit

Vermits SeCo zich toelegt op de tekstuele representatie en doorzoekbaarheid van real-world concepten, is het voldoende om ondersteuning te bieden aan bytestrings die die concepten kunnen aangeven. Hoe groter het aantal beschreven concepten, des te ruimer en diverser de gegevensverzameling.

6 iMeMex Personal Dataspace Management

Gebruikers verzamelen, organiseren en doorzoeken dagelijks digitale gegevens, zoals e-mails, documenten, contacten, afbeeldingen, video's en muziek en doen zo aan Personal Information Management (PIM). Zoals de opsomming uit de vorige zin duidelijk maakt, is die persoonlijke informatie een heterogene mengeling. Daarbij komt ook nog dat die gegevens uit uiteenlopende bronnen onttrokken kan worden: bestandssystemen op PC's, internet, GSM's, draagbare muziekspelers enzovoorts. Het probleem dat deze situatie met zich meebrengt is dat bijvoorbeeld de query "Toon mij alle broncodebestanden waarin sprake is van een 'GPL-licentie' en die behoren tot het project 'Half-Life 3' in één actie beantwoorden niet kan met traditionele, bestaande PIM-tools. De reden hiervoor is dat de voorgaande query zowel naar *outside information* als naar *inside information* [2]vraagt:

- outside information: het al dan niet behoren van een broncodebestand tot het Half-Life 3-project oftewel de positie van een bestand binnen het bestandssysteem;
- inside information: het al dan niet aanwezig zijn van bepaalde content (hier: GPL-licentie) binnen een bestand.

Deze query slaat met andere woorden een *outside-inside-kloof* die enkel te overbruggen is door meerdere manuele acties na elkaar uit te voeren. In het bovenstaand voorbeeld zouden die acties kunnen zijn:

1. isoleer alle bestanden die behoren tot het Half-Life 3-project;
2. test alle bestanden op de aanwezigheid van "GPL" als inhoud.

Het iMeMex project laat in de eerste plaats toe om heterogene gegevens op een uniforme manier te representeren door gebruik te maken van het iMeMex Data Model (iDM, zie section 6.1). Door gebruik te maken van iMeMex Query Language (iQL) kunnen queries op dit datamodel uitgevoerd worden. Er kan dus van het iMeMex-project gezegd worden dat het een 'personal information space' upgradet naar een doorzoekbare database.

6.1 iMeMex Data Model

Het iMeMex Data Model [4] is een datamodel dat toelaat heterogene gegevens op een uniforme manier te representeren en vervolgens te queryen met behulp van iQL. De uniforme representatie is mogelijk door intensief gebruik te maken van *resource views*: zowel elementen in een XML-bestand als alle knopen in een hiërarchisch bestandssysteem

worden door telkens één resource view voorgesteld. Resource views kunnen daarom echter wel van type verschillen, afhankelijk van de structuur van hun inhoud. Om die types te definiëren, wordt gebruik gemaakt van *resource view classes* die elk precies definiëren hoe een resource view intern voorgesteld zal worden. Ook nu reeds het vermelden waard is dat de resource views onderling verbonden kunnen zijn met elkaar, voorgesteld door een *resource view graph*. Wat volgt is een bespreking (formele definitie met enkele voorbeelden) van zowel een resource view en een resource view class. Vervolgens wordt er een voorbeeld van een resource view graph gegeven en besproken.

6.1.1 Resource View

Een resource view V_i is samenstelling van vier componenten:

- naam (η_i): bevat zonder meer de naam van V_i;
- tuple (τ_i): τ_i is van de vorm (W, T) met
  - W een schema van de vorm $W = (a_j)$ met j = 1, 2, ..., k zijnde een verzameling attributen waarvan a_j de naam is van de rol die door het domein D_j in W gespeeld wordt;
  - T een tuple van de vorm $T = (v_j)$ met j = 1, 2, ..., k zijnde een verzameling van atomaire waarden waarvan v_j de waarde voorstelt die hoort bij a_j. T moet overigens conform zijn aan W.
- inhoud (χ_i): een string van symbolen die uit een alphabet Σ_c komen. χ_i is van de vorm $\chi_i = (c_1, ..., c_l)$ met $c_j \in \Sigma_c$ en j = 1, ..., l.
- groep (γ_i): γ_i is van de vorm (S, Q) met
  - S een (ongeordende) verzameling van resource views;
  - Q een geordende lijst van resource views.

  Stel dat er een resource view V_i een niet-lege groepscomponent γ_i bevat. Als dan $\exists$ V_k: $V_k \in S \cup Q$, dan kan gesteld worden dat V_i direct gerelateerd is aan V_k.Elke resource view kan gerelateerd zijn aan meerdere andere resource views. In het geval dat $V_i \rightarrow V_j \rightarrow ... \rightarrow V_k$ kan gesteld worden dat V_k indirect gerelateerd is aan V_i.

Deze vier componenten bevatten de volgende eigenschappen:

- de naamcomponent η_i is belangrijk voor iQL, omdat queries in iQL veelvuldig gebruik maken van de waarde ervan;
- het schema W van τ_i wordt ingevuld door een resource view class (zie section 6.1.2);
- de gegevens van de inhoudscomponent χ_i is ongestructureerd en stelt bijvoorbeeld de karakters van een tekstbestand of van een XML-node voor;
- de resource views uit S en Q van de groepscomponent γ_i creëeren een gerichte graaf die onderling gerelateerde views met elkaar verbindt. Afhankelijk van het belang van de volgorde van de resource views slaan we die ofwel op in S (onbelangrijk; S is niet zonder reden een ongeordende verzameling zoals hierboven reeds vermeld staat), ofwel in Q (wel belangrijk*;* Q kan die volgorde namelijk bewaren).

6.1.2 Resource View Class

Een resource view class bepaalt welke restricties er gelden voor een resource view. Een formele definitie ervan is als volgt.

Stel D zijnde een verzameling van resource views, dus $D = \{ V_i \}$ met i = 1, ..., n en stel C zijnde een resource view class. C bevat restricties op de vier componenten van V_i (zijnde η_i, τ_i, χ_i en γ_i; zie section 6.1.1) die als volgt kunnen zijn:

- C kan aangeven of er componenten van V_i leeg moeten zijn of niet;
- C bepaalt het schema W van τ_i;
- C bepaalt de verzameling van resource view classes die kunnen gelden voor V_j zodat $V_i \rightarrow V_j$.

Als een resource view V_i conform een resource view class C is, dan wordt dat als V_i^C genoteerd. De vier componenten van V_i worden met dezelfde voorwaarde dan voorgesteld door η_i^C, τ_i^C, χ_i^C en γ_i^C.

6.1.3 Voorbeelden

Dit voorbeeld spitst zich specifiek toe op resource views en resource view classes die gelden bij het representeren van een bestandssysteemhiërarchie (bestanden en folders) en de gestructureerde inhoud van een XML-bestand. Vooreerst creëert table 1 een formeel overzicht van de restricties die een resource view class aan een resource view kan opleggen.

Beschrijving	Naam	η_i^C	τ_i^C	χ_i^C	$S \in \gamma_i^C$	$Q \in \gamma_i^C$
Bestand	file	N_f	$(W_{FS},\ T_f)$	C_f	$\emptyset$	()
Folder	folder	N_F	$(W_{FS},\ T_F)$	()	$\{V_1^{kind}, ..., V_m^{kind}\}$ met kind $\in$ {file, folder}	()
XML text node	xmltext	()	()	C_t	$\emptyset$	()
XML element	xmlelem	N_E	$(W_E,\ T_E)$	()	$\emptyset$	$(V_1^{xmlnode},..., V_n^{xmlnode})$ met xmlnode $\in$ {xmltext, xmlelem}
XML document	xmldoc	()	()	()	$\emptyset$	$(V_{root}^{xmlelem})$
XML bestand	xmlfile	N_f	$(W_{FS},\ T_f)$	C_f	$\emptyset$	(V_{doc}^{xmldoc})

Table 1 – Overzicht in tabelvorm van de restricties die resource view classes uitspreken over resource views die bestanden, folders en de inhoud van XML-bestand representeren.

6.1.3.1 Bestandssysteem De hiërarchie van een bestandssysteem (weergegeven door de eerste twee rijen uit de bovenstaande tabel) geldt als eerste voorbeeld. Een traditioneel bestandssysteem kan beschouwd worden als zijnde een boom waarvan de knopen folders voorstellen en de bladeren bestanden of lege folders. Stel N zijnde een interne knoop (folder) en n een blad (bestand), dan zijn de volgende eigenschappen van toepassing:

- N en n beschikken over een verzameling attributen zoals grootte en aanmaakdatum die met behulp van een schema W_{FS} gedefinieerd worden. Bijvoorbeeld W_{FS} = {grootte: int, aanmaakdatum: datetime, ...}.
- De waarden van N en n die bij deze attributen horen, conform met W_{FS}, zitten vervat in T_n.
- De naam van N en n is voorgesteld door respectievelijk N_N en N_n.
- De inhoud van n zit vervat in C_n.
- Alle subfolders van N, alsook de bestanden in N, zitten vervat in S. Merk op dat de betreffende constructie $\{V_i^{kind}, ..., V_m^{kind}\}$ met kind $\in$ {file, folder} recursief van aard is

Zoals vermeld in section 6.1.2 kan een resource view class afgeleid zijn van een andere. Gebruik makend van deze eigenschap is het mogelijk om *gespecialiseerde* file resource view classes te bepalen, bijvoorbeeld van een XML-bestand. In dit geval spreken we van een 'XML file resource view class', die alle componenten van de file resource view class overerft. Het verschil is terug te vinden bij de groepscomponent γ_i^C, want die is namelijk ingevuld met $(\emptyset,\ (V_{doc}^{xmldoc}))$. De betekenis hiervan is dat een XML file resource view class een referentie bevat naar de resource view V_{doc}^{xmldoc} dewelke de ingang vormt tot de daadwerkelijke XML-structuur binnen het XML-bestand (zie het voorbeeld hier net onder).

6.1.3.2 XML Als tweede voorbeeld beschouwen we XML. Meer specifiek komen de kernbouwstenen van een XML-structuur aan bod, zijnde 'document', 'element', 'attribute' en 'tekst'. Een bottom-up overzicht van de resource view classes die bij elk van die bouwstenen horen:

- De tekstwaarde van elke XML-node zit zonder meer vervat in C_t van $\chi_i^{xmltext}$.
- Stel E zijnde een XML-element, dan:
  - stelt N_E de naam voor van E;
  - stelt W_E het schema voor waaraan E moet voldoen en T_E de overeenkomstige waarden van E en
  - bevat Q alle kinderen van E, opnieuw gebruik makend van een recursieve opstelling ($V_1^{xmlnode}$,..., $V_n^{xmlnode}$) met xmlnode $\in$ {xmltext, xmlelem}.
- XML-attributen worden gemodelleerd door τ_i^C van $V_i^{xmlelem}$.
- Het globale 'XML-document' is in feite een encapsulatie van het root-element van de XML-structuur, zijnde $V_{root}^{xmlelem}$.

Een concreter voorbeeld van dit laatste wordt weergegeven in figure 25. De eerste afbeelding toont een fragment uit een XML-bestand, waarvan de tweede afbeelding een voorstelling creëert gebruik makend van resource view classes. Rechts van elke resource view class staat een overzicht van alle ingevulde componenten van die class.

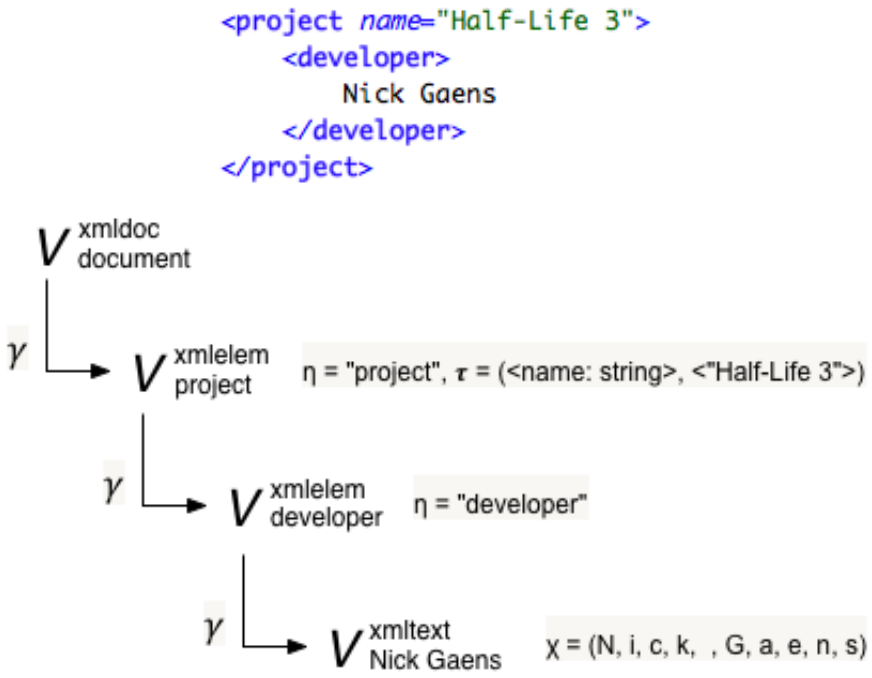

Figure 25 – Voorbeeld van een XML-structuur die vertaald is naar resource view classes met telkens die componenten erbij die ingevuld zijn.

6.1.4 Resource View Graph

Een resource view graph is een graafrepresentatie van een verzameling resource views die onderling naar elkaar verwijzen (zie de uitleg van de groepscomponent γ van een resource view in section 6.1.1). figure 26 toont een voorbeeld van een bestandssysteem en de resource view graph die dat systeem voortbrengt. In de eerste afbeelding van deze figuur is een fragment van een traditioneel bestandssysteem terug te vinden met bovenaan een folder 'Projecten' met daarin drie subfolders: 'Half-Life 3', 'Duke Nukem Forever 2' en een alias folder naar de 'Projecten'-folder. In de 'Half-Life 3'-folder staan tenslotte twee bestanden: 'main.cpp' en 'presentation.ppt' met telkens een klein uitreksel van hun inhoud. In de tweede afbeelding van deze figuur is een graafrepresentatie van deze boomstructuur terug te vinden. Merk de twee volgende eigenschappen op:

- de graaf is gericht en
- aliases binnen een bestandssysteem worden vertaald naar een bidirectionele edge tussen twee resource view nodes.

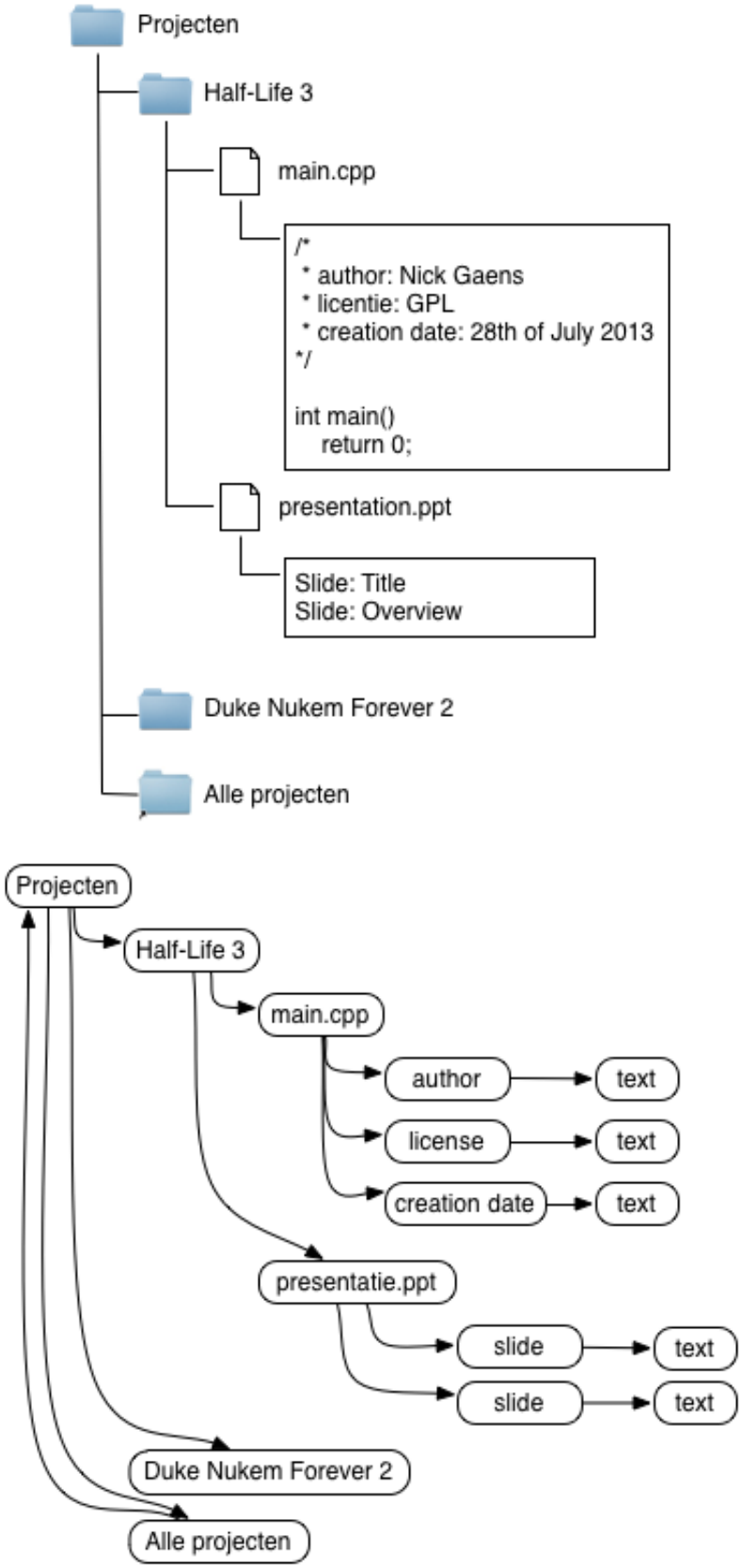

Figure 26 – De eerste afbeelding geeft een hiërarchie weer zoals die kan bestaan op een bestandssysteem, alsook wat voorbeeldinhoud van twee bestanden. De tweede afbeelding toont de Resource View Graph die deze gegevens vervat.

6.2 Attribute-Value modellering

6.2.1 Losse datastructuur

iPDMS maakt - net als Search Computing - gebruik van een graaf, maar hier stelt die eerder een hiërarchische structuur voor die erg hard lijkt op een traditioneel computerbestandssysteem. Zie bijvoorbeeld de onderste afbeelding van figure 26 waarin resource views naar elkaar 'wijzen' om zo een hiërarchie op te bouwen. Dit betekent natuurlijk dat de resource views over informatie moeten beschikken waarin de hiërarchie ("Welke resource views beschouw ik als een van mijn kinderen?") beschreven staat. En inderdaad, herinner dat elke resource view beschikt over een groepscomponent γ (ziesection 6.1.1) waarin de kinderen van een resource view ondergebracht kunnen worden. Van het iPDMS-project kan dus niet gezegd worden dat het een losse datastructuur hanteert, omdat het net onderlinge relaties tussen resource views als een kerneigenschap bevat. Dit laatste valt ook op als we de opbouw van een resource view met behulp van een attribute-value dataspace modelleren (zie figure 27). De objecten in die dataspace zijn namelijk onderling verbonden door middel van id's. De volgorde van de geordende lijst van resource views (Q van γ) wordt uitgedrukt met behulp van een extra item ('volgorde') in de betrokken objectjes.

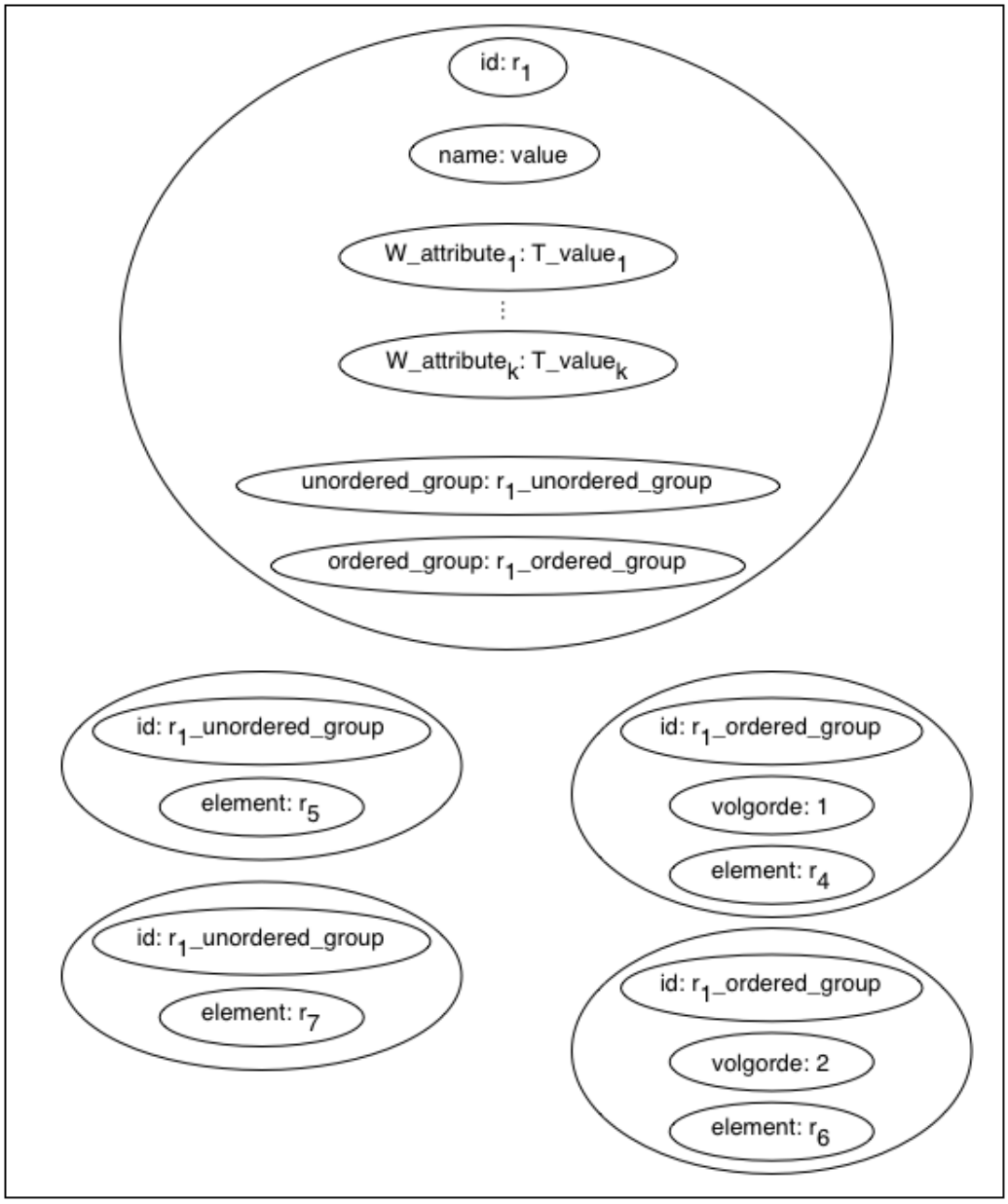

Figure 27 – iMeMex dataspace voorgesteld als een attribute-value dataspace.

6.2.2 Schemaloosheid

Zoals voorgesteld in section 6.1.2 bestaan er een aantal resource view classes die elk vastleggen welke componenten van resource views ingevuld dienen te worden (en welke niet). Een eerste vaststelling daarbij is dat het gedrag van resource view classes sterk

aanleunt bij dat van een relationeel databaseschema, omdat beide de 'vorm' bepalen die toe te voegen gegevens moeten aannemen vooraleer ze aanvaard worden. Van schemaloosheid is dus helemaal geen sprake.

6.2.3 Gegevensdiversiteit

Wat betreft de diversiteit van de gegevens is de omvang van de verzameling van resource view classes de bepalende factor. Een resource view class definieert immers een soort template of schema voor een bepaald bestandstype waaraan een resource view moet voldoen opdat die een bestand van het betreffende type mag representeren. Dus het aantal ondersteunde bestandstypes is gelijk aan het aantal beschikbare resource view classes.

6.3 iMeMex Query Language

section 6.1 geeft weer hoe een iMeMex Data Model opgebouwd wordt door gebruik te maken van resource views die op hun beurt gerestricteerd worden door resource view classes. Door de resource view-hiërarchie in rekening te brengen, gebruik makend van de onderlinge relaties tussen resource views, ontstaat een gerichte graafrepresentatie, zijnde de resource view graph. In het begin van deze sectie werd de context reeds gecreëerd om te kunnen zoeken binnen een iDM. In de praktijk betekent dit dat er een querytaal moet bestaan of ontwikkeld worden die uitgevoerd kan worden op een resource view graph. De iMeMex Query Language (iQL) biedt deze mogelijkheid.

6.3.1 Syntax en semantiek

Een formele syntaxdefinitie en gedetailleerde beschrijving van de semenatiek van iQL zijn in de bestaande literatuur nagenoeg onbestaande. In [2] en [4] zijn in totaal slechts een handvol voorbeelden terug te vinden. Wel wordt er in [4] een tekstueel overzicht gegeven van de eigenschappen van de huidige implemenatie van iQL en dat zijn de volgende:

- de syntax van iQL is een mengeling van enerzijds typische keyword expressies zoals bekend uit de interactie met zoekrobots en anderzijds een navigationele opbouw zoals bijvoorbeeld XPath;
- ondanks dat de syntax van iQL sterke gelijkenissen vertoont met XPath (en XQuery), verschilt de semantiek ervan. XPath laat enkel het queryen toe van een sterk schemagebonden structuur, terwijl iQL dankzij de lossere aard van resource view graphs een meer tolerarant omgaat met bijvoorbeeld imprecisies in een query.

Een voorbeeld hiervan is dat iQL bij verwerken van de query $size > 5$ behalve het attribuut $size$ ook sterk daarop lijkende attributen beschowt, zoals $fileSize$ of $docSize$.

6.3.2 Voorbeelden

Query: "Nick Gaens"
Betekenis: Lever alle resource views op waarvan de inhoudscomponent χ "Nick Gaens" bevat.

Query: "Nick" and "Gaens"
Betekenis: Levert alle resource views op waarvan de inhoudscomponent χ zowel "Nick" als "Gaens" bevat.

Query: [size > 42 and creationdate < today()]
Betekenis: Lever alle resource views op waarvan de waarde (uit $T \in \tau$) voor het attribuut 'size' (uit $W \in \tau$) groter is dan 42 en die op dezelfde manier een waarde hebben voor het attribuut 'creationdate' vroeger dan vandaag.

Query: //main.cpp[class="source_file"]
Betekenis: Lever alle resource views op met de naam η gelijk aan "main.cpp" en die tot de resource view class "source_file" behoren.

Query: //Half-Life 3//main.cpp[class="source_file"]
Betekenis: Lever alle resource views op met de naam η gelijk aan "main.cpp" en die tot de resource view class "source_file" behoren. Ook moeten de op te leveren resource views onrechtstreeks verbonden zijn met een resource view met de naam η gelijk aan "Half-Life 3".

Query: //Half-Life 3//main.cpp[class = "source_file"] and "GPL"]
Betekenis: Lever alle resource views op met de naam η gelijk aan "main.cpp" en die tot de resource view class "source_file" behoren én waarvan de inhoudscomponent χ "GPL" bevat. Ook moeten de op te leveren resource views onrechtstreeks verbonden zijn met een resource view met de naam η gelijk aan "Half-Life 3".

6.3.3 ASL

In figure 27 wordt een voorbeeld gegeven van hoe een iMeMex dataspace gemodelleerd kan worden als een attribute-value dataspace. Het is dus geen verrassing dat een iQL-query kan geschreven worden als een ASL-expressie. Wat volgt is een simpel voorbeeld dat dit aantoont.

Query: "Geef alle objecten met 'de naam' 'Introductie' die 'geschreven zijn door' 'Nick' en waarnaar gelinkt wordt vanuit een object met de naam 'PIM'.".

- iQL:

```
//PIM/Introductie/[auteur="Nick"]
```

- ASL:

```
(LINK[id eq-val S](name = PIM)) AND name = Introductie AND
    auteur = Nick
```

Merk op dat er in de ASL-expressie een kleine *shortcut* genomen is om te bepalen of er vanuit een 'PIM'-genaamd object gelinkt wordt naar het huidige document. We lenen hiervoor de letter S, zijnde de notatie van de verzameling objecten waarnaar het huidige object link (zie section 6.1.1).

Verderop in de tekst (zie section §7) zal duidelijk worden dat ASL op zijn beurt voorgesteld kan worden door SQL. Ter volledigheid wordt de bovenstaande query daarom ook even in SQL voorgesteld.

- SQL:

```
SELECT * FROM Relation R1 WHERE R1.name = "Introductie" AND
     R1.auteur = "Nick" AND EXISTS (SELECT * FROM Relation
    R2 WHERE R2.name = "PIM" and R1.id IN R2.S)
```

7 Performantie ASL

Voorgaand zijn o.a. attribute-value dataspaces en ASL aan bod gekomen. Interessant is nu om te proberen achterhalen op welke manier ASL-expressies querytechnisch opgebouwd zijn en uitgevoerd worden en hoe snel zulke expressies in de praktijk zijn. Om dat te kunnen doen, zijn er twee mogelijkheden. Ofwel moet er een volledig nieuwe implementatie voorzien worden van een omgeving die niet enkel een abstracte attribute-value dataspace kan voorstellen maar ook nog eens ondersteuning biedt voor ASL in de vorm van een query processor. Ofwel gebruik maken van bestaande en populaire technologieën, namelijk relationele databases en SQL voor respectievelijk de representatie van een attribute-value dataspace en het verwerken van een ASL-expressie. Het is niet zinvol om het wiel proberen heruit te vinden: relationele databases en SQL bestaan al een lange tijd en zijn inmiddels uitgebreid onderzocht en geoptimaliseerd door de jaren heen. Logischerwijs wordt er daarom voor de tweede optie gekozen. Ook zal die keuze met het oog op de benchmarks verderop in section 7.3 niet onverstandig zijn.

7.1 Attribute-value dataspace in een relationele database

Zoals in section §3 voorgesteld, is een attribute-value dataspace structureel erg eenvoudig: een verzameling van objecten die elk uit items bestaan. Elk item is op zijn beurt een combinatie van een attribute met een value. Die structuur kan zonder problemen vertaald worden naar een relationeel model, door het volgend schema te hanteren:

vertaald naar een relationeel model levert het volgende databaseschema op:

```
(oid TEXT, attribute_name TEXT, attribute_value TEXT)
```

Figure 28 – Relationeel model dat een attribute-value dataspace simuleert.

.

7.2 ASL-to-SQL converter

Omdat er gekozen is voor SQL om ASL-expressies naar te vertalen, is er uiteraard nood aan een tool die ASL-expressies omzet naar SQL-queries. Die tool is binnen het kader van deze thesis ontwikkeld in de programmeertaal Python. In deze subsectie staat een overzicht van de belangrijkste elementen van de tool, soms gepaard met voorbeelden en daadwerkelijke programmacode ter illustratie.

Om te beginnen kan gesteld worden dat initieel de tool achtereenvolgens twee stappen uitvoert:

1. parse de gegeven ASL-expressie en bouw een expressieboom op, waarvan de bladeren keywords zijn en de knopen de operatoren;

2. loop in preorde recursief door die boom en vertaal elke knoop of blad en bouw zo een SQL-query op.

In pseudo-code levert dat het volgende op:

```
def translate_ASL_to_SQL(expression):
    # First, build an expression tree by parsing the expression.
    expression_tree = parse(expression)

    # Second, translate each node of the tree to SQL
    # and start with the root while using preorder traversal.
    translate_node_to_SQL(expression_tree.root)
```

Figure 29 – Pseudo-code die het concept van de ASL-to-SQL vertaler weergeeft.

7.2.1 Parser

Onder het motto 'verdeel en heers' is het opportuun om de oorspronkelijke ASL-expressie te parsen en op te delen in kleinere stukken. Dat gebeurt op de resolutie van keywords en operatoren omdat die dan vervolgens vrij eenvoudig te converteren zijn naar SQL. Het parsen van de ASL-expressie gebeurt met behulp van een externe Python-library, namelijk PyParsing `http://pyparsing.wikispaces.com/`. Die library laat toe om geneste strings om te zetten in een multidimensionale array die dan een expressieboom vormt die de tool in de tweede fase zal kunnen doorlopen. PyParsing is een erg krachtige library, maar dat gaat helaas gepaard met een ietwat complexe en *on-Pythonic* syntax zoals te zien is in figure 30.

```
token = Word(alphanums + ':' + '(' + ')' + '{' + '}' + ' ' + ',' + '*' + '-')
grammar = Forward()
nestedBrackets = nestedExpr('[', ']', content=grammar)
grammar << (token | nestedBrackets)
self._parts = grammar.parseString(input)[0]
```

Figure 30 – Syntax van PyParsing, een externe Python-library die geneste strings kan parsen.

Ook vereist deze library dat de invoer geëncapsuleerd is, wat wil zeggen dat de onderdelen van een ASL-expressie die zich tussen de keywords bevinden tussen (vierkante) haakjes genest moeten worden worden. De onderstaande voorbeelden tonen aan dat dit echter slechts een kleine syntactische impact heeft.

e_8 := [(name: Michael Schumacher)
AND [(setting: {presentation, race}) OR (year: NOT 2003)]]
e_9 := [Link((*) eq (*)((team: Ferrari))]
e_{10} := [[Link((*: Vettel) eq-attr (*: Schumacher))([Link((name: Schumacher) eq-val
(setting: *))((setting: presentation))])] AND [(setting: race, presentation)]]

Dit levert de volgende expressiebomen op in de vorm van geneste arrays:

e_8 → ['name: Michael Schumacher', 'AND',
['setting: {presentation, race}', 'OR', 'year: NOT 2003']]
e_9 → ['Link((*) eq (*)((team: Ferrari))']
e_{10} → ['Link((*: Vettel) eq-attr (*: Schumacher))(',
['Link((name: Schumacher) eq-val (setting: *))((setting: presentation))'],
'AND', '(setting: race, presentation)]

7.2.2 Converter

De parser genereert dus een expressieboom die vervolgens als input dient voor de tweede fase van de tool, zijnde de 'converter' oftewel 'omzetter'. De converter is in weze een recursief algoritme dat de expressieboom in preorde bewandelt en elke knoop en elk blad omzet naar een SQL-subquery. Die SQL-subqueries worden dan toegevoegd aan totaalquery die uiteindelijk als resutltaat zal gelden. De converter krijgt zijn uitvoering te maken met twee soorten elementen, namelijk de keywords (bladeren van de expressieboom) en operatoren (knopen van die boom) van ASL. Er volgt nu een overzicht van de gebruikte conversieregels met, waar nodig, een relevant stuk programmacode en enkele voorbeelden.

7.2.2.1 Keywords Een keyword wordt door de converter omgezet naar een eenvoudige subquery waarvan de WHERE-clausule gevuld wordt met de elementen van het keyword. De *keyword converter* houdt daarbij met de volgende zaken rekening:

- Is het attribute van het keyword een verzameling van waardes (notatie d.m.v. '{' en '}') of slechts één enkele waarde?
- Is de value van het keyword een verzameling van waardes (notatie d.m.v. '{' en '}') of slechts één enkele waarde?
- Gaat er een negatie ('NOT') vooraf aan het attribute en/of de value?
- Bevat het keyword wildcards ('$*$')?

Een overzicht van de relevante programmacode bevindt zich hieronder.

```
def keyword_to_sql_query(keyword, relation_number, use_like):
    attribute, value = keyword.split(':')
    attribute = attribute.strip()
    value = value.strip()

    negating_attribute = attribute.upper().startswith('NOT')
    negating_value = value.upper().startswith('NOT')
    if negating_attribute:
        attribute = attribute[3:].strip()
    if negating_value:
        value = value[3:].strip()

    ### ATTRIBUTE ###
    if attribute.startswith('{'):
        attribute_elements = [attribute_element.strip() for attribute_element in re.sub('}', '', re.sub('{', '', attribute)).split(',')]
        query += ' ('
        for attribute_element in attribute_elements:
            query += 'R' + str(relation_number) + '.attribute_name '
            if negating_attribute:
                query += '!'
            query += '= \'' + attribute_element + '\''
            if not attribute_element is attribute_elements[-1]:
                if negating_attribute:
                    query += ' AND '
                else:
                    query += ' OR '
        query += ')'
    elif attribute != '*':
        query += ' R' + str(relation_number) + '.attribute_name '
        if negating_attribute:
            query += '!'
        query += '= \'' + attribute + '\''
    ################

    if attribute != '*' and value != '*':
        query += ' AND'

    ### VALUE ###
    if value.startswith('{'):
        value_elements = [value_element.strip() for value_element in re.sub('}', '', re.sub('{', '', value)).split(',')]
        query += ' ('
        for value_element in value_elements:
            query += 'P' + str(relation_number) + '.attribute_value '
            if use_like:
                if negating_value:
                    query += 'NOT '
                query += 'LIKE \'%' + value_element + '%\''
            else:
                if negating_value:
                    query += '!'
                query += '= \'' + value_element + '\''
            if not value_element is value_elements[-1]:
                if negating_value:
                    query += ' AND '
                else:
                    query += ' OR '
        query += ')'
    elif value != '*':
        query += ' P' + str(relation_number) + '.attribute_value '
        if use_like:
            if negating_value:
                query += 'NOT '
            query += 'LIKE \'%' + value + '%\''
        else:
            if negating_value:
                query += '!'
            query += '= \'' + value + '\''
    #############

    query += ')'
```

Figure 31 – Programmacode die een keyword omzet in SQL.

Voorbeelden

e_{11} := (year: NOT 2003)

e_{12} := ({name, team, setting}: *)

Dit levert de volgende omzettingen op (we laten de SELECT- en FROM-clausules even achterwege, omdat die in dit voorbeeld niet relevant zijn):

e_{11} → ... WHERE R1.attribute_name = ′year′AND R1.attribute_value !=’2003’
of
e_{11} → ... WHERE R1.attribute_name = ’year’
AND R1.attribute_value NOT LIKE ’%2003%’
e_{12} → ... WHERE R1.attribute_name = ′name′OR
R1.attribute_name = ′team′OR R1.attribute_name = ′setting′

Het eerste voorbeeld laat zien dat voor het string matchen van `attribute_value` de keuze bestaat om al dan niet gebruik te maken van het SQL-keyword `LIKE` in combinatie met SQL-wildcards (’%’-karakter voor en na ’2003’ in de vertaling van e_{11}). Die optie (’use_like’-argument in figure 31) is ingebouwd om flexibiliteit te simuleren tijdens het queryen van een AV-dataspace.

Het tweede voorbeeld toont op zijn beurt dat verzamelingen van attributes (of values) met behulp van het SQL-keyword ’OR’ vertaald worden. Ook valt op dat de ASL-wildcard (’∗’) eigenlijk genegeerd wordt, wat logisch is, want een wildcard betekent uiteraard dat eender welke waarde mag, dus is er geen nood aan de toevoeging van een beperking in de vorm van een extra vergelijking.

7.2.2.2 Link De link-operator, zoals toegelicht in section 3.1.2.1, drukt een impliciete join-operatie uit, dus er zullen in de uiteindelijke SQL-vertaling instanties van de betrokken relatie gejoind moeten worden.

Een vertaling naar SQL van de algemene vorm van de link-operator, zijnde link (A eqrel B)(C), levert het volgende op.

```
SELECT R1.oid
FROM Relation R1, Relation R2
WHERE A(R1)
  AND B(R2)
  AND eq-rel(R1, R2)
  AND R2.oid IN (C);
```

Figure 32 – Vertaling van de algemene vorm van de link-operator naar SQL.

Opvallend aan deze vertaling zijn de volgende elementen:

- A(R1)en B(R2) worden gelezen als "keyword A over relatie R1" en "keyword B over relatie R2". Beide onderdelen worden dan ook vertaald met behulp van de keyword converter uit section 7.2.2.1.

- eq-rel(R1, R2) wordt gelezen als "eq-rel tussen relaties R1 en R2" en drukt dus eigenlijk de joinconditie uit voor een join uit tussen R1 en R2. De code die de vertaling van een eq-rel naar SQL inhoudt, is weergegeven in figure 33.

```
def eq_rel_to_sql_condition(eq_rel, relation_number=1):
    if eq_rel == 'eq-attr':
        return ' R' + str(relation_number) + '.attribute_name = R' + str(relation_number + 1) + '.attribute_name'
    elif eq_rel == 'eq-val':
        return ' R' + str(relation_number) + '.attribute_value = R' + str(relation_number + 1) + '.attribute_value'
    elif eq_rel == 'eq':
        return ' R' + str(relation_number) + '.attribute_name = R' + str(relation_number + 1) + '.attribute_name ' \
            'AND R' + str(relation_number) + '.attribute_value = R' + str(relation_number + 1) + '.attribute_value'
```

Figure 33 – Programmacode die een eq-rel omzet in SQL.

- R2.oid IN (C) wordt gelezen als "met de object id van R2 voorkomend in de output van C". Herinner uit section 3.1.2.1 dat de C-component van een link-expressie in feite een volledige ASL-expressie is. Omwille van die reden wordt er in de code simpelweg een recursieve aanroep gedaan, voorafgegaan door een constructie van een subquery die gebruik maakt van een IN-clausule van SQL. Een opmerking die daar onmiddellijk bij geplaatst moet worden is dat sommige implementaties van een relationele database, waaronder het in deze thesis gebruikte SQLite3, geen ondersteuning bieden voor het SQL-keyword IN. Daarom dat er geopteerd wordt voor een gelijkwaardig alternatief op basis van een EXISTS-clausule. De constructie

  ```
  ...  AND R1.oid IN (SELECT R2.oid FROM Relation R2 WHERE C(R2))
  ```

  wordt dus vertaald naar het gelijkwaardige

  ```
  ...  AND EXISTS (SELECT NULL FROM Relation R2 WHERE R2.oid =
  R1.oid AND C(R2))
  ```

  .

Een belangrijke opmerking na dit bovenstaande is dat de joins die vanuit ASL gegenereerd worden, in feite semi-joins zijn. En dat is niet toevallig: een semi-join houdt namelijk in dat er geen nood is om te weten welke object of objecten er aan de joinconditie voldoet of voldoen, maar wel dat er een minstens één object bestaat dat dat doet. Men kan dus stellen dat de 'natuur van ASL ligt bij semi-joins'.

7.3 Benchmarks

Het is zinvol om de resultaten van de bovenstaande conversietool te benchmarken om zo een idee te krijgen over hoe performant ASL is.

7.3.1 Queries

Om de benchmarks uit te kunnen voeren, zijn de volgende twee queries opgesteld:

- „Geef alle crashes met een importance van 10." (query 1) en
- „Geef elke piloot die minstens tweemaal vermeld wordt en het wereldkampioenschap gewonnen heeft." (query 2).

De eerste query zal gebruik maken van de ASL-AND-operator en de tweede van de LINK-operator.

Voor elk van beide queries bestaan er vijf vertalingen:

```
a.  ASL-expressie.
b.  Een vertaling naar SQL met behulp van de tool die in section 7.2,
waarbij er geen SQL-operator LIKE met SQL-wildcards gebruikt worden.
c.  Dezelfde vertaling naar SQL als in b., maar nu met de SQL-operator
LIKE en SQL-wildcards waar mogelijk.

d.  Een manuele vertaling naar SQL waarbij er geen SQL-operator LIKE met
SQL-wildcards gebruikt worden.
e.  Dezelfde manuele vertaling als in d., maar nu met de SQL-operator
LIKE en SQL-wildcards waar mogelijk.
```

De manuele vertalingen (d. en e. in het kader hierboven) zijn zinvol om in te kunnen schatten wat de impact is van het *semi-structured* representeren van een attribute-value dataspace in een relatie in combinatie met de vertalingen door de ASL-to-SQL-tool ten opzichte van een *native* relationeel model met een meer rechttoe-rechtaan SQL-query. Zowel het *semi-structured* als *native* model wordt in section 7.3.2 geïntroduceerd.
De aan- of afwezigheid van het SQL-keyword `LIKE` (zie section 7.2.2.1) wordt verwacht de uitvoersnelheid van de queries te zullen beïnvloeden. Om een eventueel prestatieverlies tegen te gaan, zullen er ter experiment verschillende indices aangemaakt worden op zowel de gesimuleerde AV-dataspace-relatie als de *native* relatie. De indices worden voorgesteld in section 7.3.2.

De vertalingen van de twee queries zijn als volgt:

- 'query 1':

```
a.  ("setting" = "crash") AND ("importance" = "10")
b.  SELECT R1.oid FROM Dataspace R1 WHERE R1.attribute_name
= 'setting' AND R1.attribute_value = 'crash' AND EXISTS
(SELECT NULL FROM Dataspace R2 WHERE R1.oid = R2.oid AND
R2.attribute_name = 'importance' AND R2.attribute_value = '10');
c.  SELECT R1.oid FROM Dataspace R1 WHERE R1.attribute_name
= 'setting' AND R1.attribute_value LIKE '%crash%' AND EXISTS
(SELECT NULL FROM Dataspace R2 WHERE R1.oid = R2.oid AND
R2.attribute_name = 'importance' AND R2.attribute_value LIKE
'%10%');

d.  SELECT team FROM F1_Events WHERE setting = 'crash' AND
importance = 10;
e.  SELECT team FROM F1_Events WHERE setting LIKE '%crash%' AND
importance = 10;
```

- 'query 2':

```
a.  LINK[("driver":  *) eq-val ("driver":  *)][("setting":
"world champion")]
b.  SELECT R1.oid FROM Dataspace R1 WHERE attribute_name
= 'driver' AND EXISTS (SELECT NULL FROM Dataspace R2 WHERE
R2.attribute_name = 'driver' AND R1.oid != R2.oid AND
R1.attribute_value = R2.attribute_value AND EXISTS (SELECT
NULL FROM Dataspace R3 WHERE R3.attribute_name = 'setting' AND
R3.attribute_value = 'world champion' AND R2.oid = R3.oid));
c.  SELECT R1.oid FROM Dataspace R1 WHERE attribute_name
= 'driver' AND EXISTS (SELECT NULL FROM Dataspace R2 WHERE
R2.attribute_name = 'driver' AND R1.oid != R2.oid AND
R1.attribute_value = R2.attribute_value AND EXISTS (SELECT
NULL FROM Dataspace R3 WHERE R3.attribute_name = 'setting' AND
R3.attribute_value LIKE '%world champion%' AND R2.oid = R3.oid));

d.  SELECT R1.driver FROM F1_Events R1, F1_Events R2 WHERE
R1.driver = R2.driver AND R2.setting = 'world champion';
e.  SELECT R1.driver FROM F1_Events R1, F1_Events R2 WHERE
R1.driver = R2.driver AND R2.setting LIKE '%world champion%';
```

7.3.2 Testdatabases en -dataspaces

Om de vertaalde queries van hierboven daadwerkelijk uit te voeren, zijn er natuurlijk relationele instanties nodig. Enerzijds is er nood aan gesimuleerde AV-dataspaces voor de 'b.-' en 'c.-vertaling' en anderzijds aan *native* relaties voor de 'd.-' en 'e.-vertaling'. Om niet afhankelijk te zijn van één RDBMS, zullen alle tests in tweevoud uitgevoerd worden: zowel op SQLite3 (`http://www.sqlite.org/`) als DB2 van IBM (`http://www-01.ibm.com/software/data/db2/`). Voor deze thesis zijn voor beide RDBMS's telkens twee scripts ontwikkeld (in Python). De SQLite3-scripts heten respectievelijk *DatabaseGenerator_SQLite.py* en *DataspaceGenerator_SQLite.py* voor de *native* relatie en *DatabaseGenerator_DB2.py* en *DataspaceGenerator_DB2.py* voor de gesimuleerde AV-dataspace. De eerste twee maken gebruik van de in Python ingebouwde `sqlite3`-module om rechtstreeks SQLite3-instanties on-disk aan te maken. De twee DB2-scripts kunnen op hun beurt helaas geen gebruik maken van de '3rd party'-module `PyDB2`. Die module gaf tijdens de ontwikkeling allesbehalve een stabiele indruk en werd daarom achterwege gelaten. Als oplossing werd gekozen om de DB2-scripts SQL-commando's laten weg te schrijven naar een tekstbestand om die dan in een volgend stadium via de

Command Line Interpreter van DB2 te importeren en automatisch uit te voeren.

Wat volgt is een overzicht van de belangrijkste onderdelen van de vier scripts.

- Creëeren van de eigenlijke instantie:
  - DatabaseGenerator_SQLite:

```
CREATE TABLE F1_Events(team TEXT, driver TEXT,
setting TEXT, importance INT);
```

  - DataspaceGenerator_SQLite:

```
CREATE TABLE F1_Events(oid INT, attribute_name
TEXT, attribute_value TEXT);
```

  - DatabaseGenerator_DB2:

```
CREATE TABLE <table_name> (team CHAR(23), driver
CHAR(17), setting CHAR(14), importance INT);
```

  - DataspaceGenerator_DB2:

```
CREATE TABLE <table_name> (oid INT, attribute_name
CHAR(10), attribute_value CHAR(23));
```

- Creëeren van (optionele) indices:
  - DatabaseGenerator_SQLite en DatabaseGenerator_DB2:

```
CREATE INDEX team_index ON F1_Events(team);
CREATE INDEX driver_index ON F1_Events(driver);
CREATE INDEX setting_index ON F1_Events(setting);
CREATE INDEX importance_index ON
F1_Events(importance);
```

  - DataspaceGenerator_SQLite en DataspaceGenerator_DB2:

```
CREATE INDEX oid_index ON F1_Events(oid);
CREATE INDEX attribute_name_index ON
F1_Events(attribute_name);
CREATE INDEX attribute_value_index ON
F1_Events(attribute_value);
```

- Vullen van de instantie (SQLite) of genereren van INSERT-statements (DB2):
  - DatabaseGenerator_SQLite:

```
i = 0
while i < sample_size:
    team = rand_team()
    cur.execute("INSERT INTO F1_Events VALUES(?, ?, ?, ?)", (team, rand_driver_of_team(team), rand_setting(), rand_importance()))
    i += 1
```

– DataspaceGenerator_SQLite:

```
oid = 0
while oid < sample_size:
    team = rand_team()
    driver = rand_driver_of_team(team)
    setting = rand_setting()
    importance = rand_importance()
    cur.executemany("INSERT INTO F1_Events VALUES(?, ?, ?)", [(oid, 'team', team),
                                                              (oid, 'driver', driver),
                                                              (oid, 'setting', setting),
                                                              (oid, 'importance', str(importance))])
    oid += 1
```

– DatabaseGenerator_DB2:

```
i = 0
while i < sample_size:
    team = rand_team()
    insert_into_statement = 'INSERT INTO ' + table_name + ' VALUES(\'' + team + '\', ' +\
                                                          '\'' + rand_driver_of_team(team) + '\', ' +\
                                                          '\'' + rand_setting() + '\', '\
                                                              + str(rand_importance()) + ');\n'
    file.write(insert_into_statement)
    i += 1
```

– DataspaceGenerator_DB2:

```
oid = 0
while oid < sample_size:
    team = rand_team()
    driver = rand_driver_of_team(team)
    setting = rand_setting()
    importance = str(rand_importance())
    insert_into_statement = 'INSERT INTO ' + table_name \
                          + ' VALUES (' + str(oid) + ', \'team\', \'' + team + '\'), '\
                                        + '(' + str(oid) + ', \'driver\', \'' + driver + '\'), '\
                                        + '(' + str(oid) + ', \'setting\', \'' + setting + '\'), '\
                                        + '(' + str(oid) + ', \'importance\', \'' + importance + '\');\n'
    file.write(insert_into_statement)
    oid += 1
```

- De vier helperfuncties die in allevier de codefragmenten van het voorgaande puntje gebruikt worden:

```
def rand_team():
    return random.choice(teams_drivers.keys())
def rand_driver_of_team(team):
    return random.choice(teams_drivers[team])
def rand_setting():
    return random.choice(settings)
def rand_importance():
    return random.randint(1, 10)
```

- De data waarmee de instanties gevuld worden:

```
teams_drivers = {'McLaren-Mercedes': ('Jenson Button', 'Kevin Magnussen'),
                 'Mercedes': ('Nico Rosberg', 'Lewis Hamilton'),
                 'Ferrari': ('Fernando Alonso', 'Kimi Raikkonen'),
                 'Red Bull Racing-Renault': ('Daniel Ricciardo', 'Sebastian Vettel'),
                 'Williams-Mercedes': ('Felipse Massa', 'Valtteri Bottas'),
                 'Force India-Mercedes': ('Nico Hulkenberg', 'Sergio Perez'),
                 'STR-Renault': ('Jean-Eric Vergne', 'Daniil Kyvat'),
                 'Lotus-Renault': ('Roman Grosjean', 'Pastor Maldonado'),
                 'Marussia-Ferrari': ('Jules Bianchi', 'Max Chilton'),
                 'Sauber-Ferrari': ('Esteban Gurierrez', 'Adrian Sutil'),
                 'Caterham-Renault': ('Marcus Ericsson', 'Kamui Kobayashi')}

settings = ('victory',
            'pole position',
            'overtaking',
            'pit stop',
            'crash',
            'bad start',
            'good start',
            'world champion',
            'flat tire',
            'car failure')
```

Merk aan de bovenstaande codefragmenten het volgende op:

- Het simuleren van een attribute-value dataspace gebeurt met behulp van een relatieschema dat bestaat uit drie attributen: 'oid' (object id), 'attribute_name' en 'attribute_value'.
- De voorbeeldgegevens horen thuis in de context van de autorijsport 'Formule 1' wat ook meteen de naamgeving 'F1_Events' verklaart.
- De indices zijn optioneel. Voor de benchmarks zullen er vier varianten bestaan van elke instantie:
  - Database:
    - Geen index ('no_index')
    - Index op 'importance'-kolom ('importance_index')
    - Indices op 'team'-, 'driver'- en 'setting'-kolom ('team_index', 'driver_index' en 'setting_index')
    - Indices op allevier de kolommen
  - Dataspace:
    - Geen index ('no_index')

    - Index op oid ('oid_index')
    - Index op attribute_name en attribute_value ('attribute_name_index' en 'attribute_value_index')
    - Index op alledrie de kolommen

- De variabele `sample_size` bepaalt de grootte van de instantie, die als 200, 2000, 20000 of 200000 kan ingesteld worden. Dus er zullen vier verschillende instanties bestaan per combinatie van indices.

Ter volledigheid staan hieronder nog twee extracten uit de resultaten van de scripts

- DatabaseGenerator_SQLite en DatabaseGenerator_DB2.py:

	team	driver	setting	importance
1	Ferrari	Kimi Raikkonen	world champion	5
2	Mercedes	Lewis Hamilton	overtaking	6
3	STR-Renault	Jean-Eric Vergne	victory	2
4	Caterham-Renault	Marcus Ericsson	victory	8
5	Force India-Mercedes	Nico Hulkenberg	victory	7
6	Williams-Mercedes	Felipse Massa	overtaking	4
7	McLaren-Mercedes	Kevin Magnussen	world champion	8
8	Marussia-Ferrari	Max Chilton	pit stop	3
9	McLaren-Mercedes	Kevin Magnussen	flat tire	7
10	Force India-Mercedes	Nico Hulkenberg	good start	4
11	Ferrari	Kimi Raikkonen	pole position	6
12	Lotus-Renault	Pastor Maldonado	crash	7
13	Red Bull Racing-Renault	Sebastian Vettel	good start	5
14	STR-Renault	Jean-Eric Vergne	bad start	7
15	Sauber-Ferrari	Adrian Sutil	car failure	6
16	Lotus-Renault	Pastor Maldonado	overtaking	1

Figure 34 – Voorbeeld van een relatie gegenereerd door een DatabaseGenerator-script.

- DataspaceGenerator_SQLite en DataspaceGenerator_DB2.py:

	oid	attribute_name	attribute_value
1	0	team	Mercedes
2	0	driver	Nico Rosberg
3	0	setting	crash
4	0	importance	10
5	1	team	Mercedes
6	1	driver	Nico Rosberg
7	1	setting	overtaking
8	1	importance	10
9	2	team	Marussia-Ferrari
10	2	driver	Max Chilton
11	2	setting	pit stop
12	2	importance	5
13	3	team	Caterham-Renault
14	3	driver	Marcus Ericsson
15	3	setting	pit stop
16	3	importance	10

Figure 35 – Voorbeeld van een relatie gegenereerd door een DataspaceGenerator-script.

7.3.3 Resultaten

De tests werden uitgevoerd op een desktop met daarin een Intel Core i5-3570 (quad-core, 3.4GHz per core), 8GB DDR3 RAM-geheugen en een up-to-date Windows 7 64-bit-installatie. De SQLite3-versie is '3.8.5 2014-06-04' en die van DB2 is 10.5.1. Elke query werd drie keer uitgevoerd, waarna het gemiddelde van alledrie de resultaten als als eindresultaat is opgenomen.

Dit zijn de resultaten in tabelvorm:

- 'query 1':

SQLite, Database, =, milliseconden	no_index	importance_index	team_index + driver_index + setting_index	team_index + driver_index + setting_index + importance_index
200	1	1	1	1
2000	2	2	2	2
20000	9	7	9	7
200000	77	53	59	61

DB2, Database, =, milliseconden	no_index	importance_index	team_index + driver_index + setting_index	team_index + driver_index + setting_index + importance_index
200	1	1	1	1
2000	3	2	15	16
20000	15	9	62	63
200000	17	16	203	250

SQLite, Database, LIKE, milliseconden	no_index	importance_index	team_index + driver_index + setting_index	team_index + driver_index + setting_index + importance_index
200	1	1	1	1
2000	2	2	2	2
20000	12	7	12	7
200000	105	57	112	61

DB2, Database, LIKE, milliseconden	no_index	importance_index	team_index + driver_index + setting_index	team_index + driver_index + setting_index + importance_index
200	1	1	1	1
2000	4	2	16	17
20000	17	16	74	76
200000	32	31	224	271

SQLite, Dataspace, =, milliseconden	no_index	oid_index	attribute_name_index	oid_index + attribute_name_index + attribute_value_index
200	4	2	4	2
2000	282	5	306	40
20000	40949	44	46333	3430
200000	4031779	430	5251712	1469109

DB2, Dataspace, =, milliseconden	no_index	oid_index	attribute_name_index	oid_index + attribute_name_index + attribute_value_index
200	6	5	4	8
2000	7	8	8	9
20000	17	20	22	25
200000	128	281	294	317

SQLite, Dataspace, LIKE, milliseconden	no_index	oid_index	attribute_name_index	oid_index + attribute_name_index + attribute_value_index
200	4	2	4	5
2000	277	5	300	307
20000	41875	47	44992	48616
200000	4001225	461	5256384	6091248

DB2, Dataspace, LIKE, milliseconden	no_index	oid_index	attribute_name_index	oid_index + attribute_name_index + attribute_value_index
200	3	4	5	6
2000	5	6	7	7
20000	18	20	22	23
200000	174	233	241	244

Table 2 – Uitvoeringstijden (in milliseconden) van 'query 1'.

- 'query 2':

SQLite, Database, =, milliseconden	no_index	importance_index	team_index + driver_index + setting_index	team_index + driver_index + setting_index + importance_index
200	6	6	6	4
2000	25	24	13	15
20000	278	284	169	158
200000	15515	15517	20554	21277

SQLite, Database, LIKE, milliseconden	no_index	importance_index	team_index + driver_index + setting_index	team_index + driver_index + setting_index + importance_index
200	6	6	5	4
2000	25	25	13	15
20000	278	283	163	157
200000	15503	15458	20672	21120

DB2, Database, =, milliseconden	no_index	importance_index	team_index + driver_index + setting_index	team_index + driver_index + setting_index + importance_index
200	1	1	1	1
2000	4	4	2	2
20000	16	15	6	5
200000	31	34	30	31

DB2, Database, LIKE, milliseconden	no_index	importance_index	team_index + driver_index + setting_index	team_index + driver_index + setting_index + importance_index
200	1	1	1	1
2000	5	4	3	3
20000	17	17	15	15
200000	31	32	28	29

SQLite, Dataspace, =, milliseconden	no_index	oid_index	attribute_name_index	oid_index + attribute_name_index + attribute_value_index
200	220	30	177	27
2000	35361	135	31764	3270
20000	5005041	3764	6565816	258937

SQLite, Dataspace, LIKE, milliseconden	no_index	oid_index	attribute_name_index	oid_index + attribute_name_index + attribute_value_index
200	251	46	204	172
2000	41106	537	32421	28529
20000	5828173	6311	7239839	3551639

DB2, Dataspace, =, milliseconden	no_index	oid_index	attribute_name_index	oid_index + attribute_name_index + attribute_value_index
200	1	1	7	8
2000	8	3	11	11
20000	23	13	33	34
200000	176	277718	413	430

DB2, Dataspace, LIKE, milliseconden	no_index	oid_index	attribute_name_index	oid_index + attribute_name_index + attribute_value_index
200	5	7	5	5
2000	7	9	11	10
20000	22	26	39	40
200000	253	287300	591	421

Table 3 – Uitvoeringstijden (in milliseconden) van 'query 2'.

7.3.4 Bespreking resultaten

Een eerste vaststelling die duidelijk opvalt aan de resultaten van de benchmarks is dat de uitvoeringstijden van beide queries uitgevoerd op de 'dataspace'-instantie hoger tot veel hoger zijn dan dan die van de 'database'-instantie, ongeacht de aanwezigheid van indices. Dat is geen onverwachte vaststelling, want zoals reeds eerder in section 7.3.1 aangehaald, is er een sterk verschil tussen de modellen van beide instantie-types. De *semi-structuredness* van de dataspace-instantie leidt immers tot een vervierdubbeling van het aantal rijen ten opzichte van de *native* database-instantie (vergelijk de INSERT-commando's van beide DataspaceGenerator-scripts met die van de DatabaseGenerator-scripts uit section 7.3.2) wat dus betekent dat elke eventuele tablescan vier keer langer duurt.
Een bijkomend 'probleem' voor de uitvoering op de 'dataspace'-instanties is dat de 'semi-joinnatuur van ASL', zoals reeds eerder aangehaald op het eind van 7.2.2.2, er voor zorgt dat de ASL-to-SQL-vertalingen in deze benchmark geneste subqueries bevatten. Neem bijvoorbeeld de vertaling 'b.' (zie section 7.3.1) van de tweede query, uitgevoerd op dataspace-instanties zonder indices. De query in kwestie:

```
SELECT R1.oid FROM F1_Events R1 WHERE attribute_name = 'driver'
AND EXISTS (SELECT NULL FROM F1_Events R2 WHERE R2.attribute_name
= 'driver' AND R1.oid != R2.oid AND R1.attribute_value =
R2.attribute_value AND EXISTS (SELECT NULL FROM F1_Events R3
WHERE R3.attribute_name = 'setting' AND R3.attribute_value =
'world champion' AND R2.oid = R3.oid));
```

wordt respectievelijk door SQLite en DB2 uitgevoerd met de volgende queryplannen:

```
a.  Doe een volledige scan van R1;
b.  Voor elke rij van R1, voer de eerste subquery
(SQ1) uit;
c.  SQ1:  Doe een volledige scan van R2;
d.  Voor elke rij van R2, voer de tweede subquery
(SQ2) uit;
e.  SQ2:  Doe een volledige scan van R3.
```

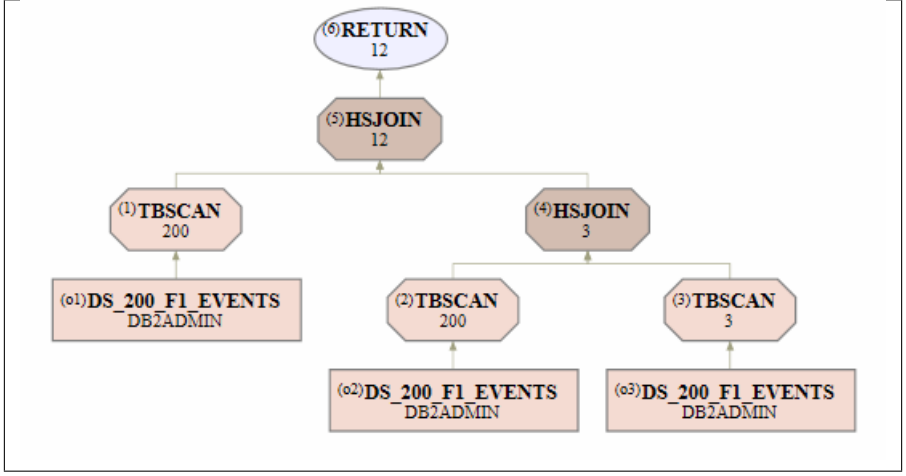

Table 4 – Queryplan van SQLite (boven) en DB2 (onder) van de ASL-to-SQL-tool-vertaling van 'query 2', uitgevoerd op de 'dataspace'-instantie zonder indices.

Wat opvalt is dus dat beide RDBMS's drie maal voor een volledige tablescan opteren, al is er een significant verschil aanwezig. Stel dat x telkens het aantal rijen is van R1, R2 en R3. Het queryplan van SQLite toont aan dat er drie geneste tablescans uitgevoerd worden, wat neerkomt op x^3 in te lezen rijen. DB2 is zuiniger wat dat betreft en houdt het op drie onafhankelijk tablescans in combinatie met hash joins en dus $3 * x$ in te lezen rijen. Dit verschil is trouwens een bepalende factor in de grote verschillen tussen de resultaten van SQLite en die van DB2: die laatste blijkt immers over een krachtigere *SQL processor* te beschikken die subqueries niet naïef uitvoert, maar wel degelijk omzet

in join-algoritmes.

Vergelijk het voorgaande met de manuele vertaling ('d.', zie section 7.3.1) van dezelfde query, uitgevoerd op de *native* database-instantie zonder indices. De query in kwestie:

```
SELECT R1.driver FROM F1_Events R1, F1_Events R2 WHERE
R1.driver = R2.driver AND R2.setting = 'world champion';
```

levert de volgende queryplannen op:

```
a.  Doe een volledige scan van R2;
b.  Doorzoek R1 en gebruik daarbij de 'AUTOMATIC
COVERING INDEX' op de driver-kolom.
```

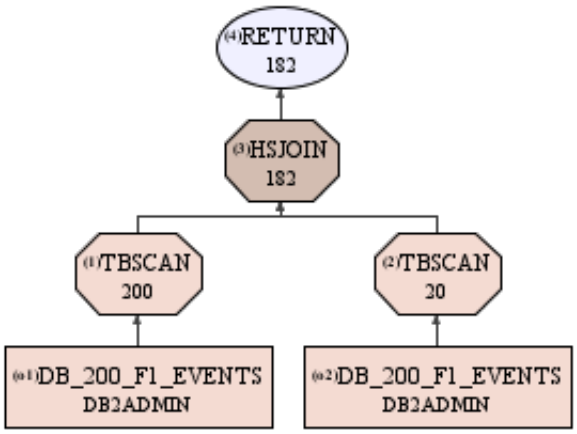

Table 5 – Queryplan van SQLite (boven) en DB2 (onder) van de manuele vertaling van 'query 2', uitgevoerd op de 'database'-instantie zonder indices.

Merk op dat geen van beide RDBMS's geneste tablescans uitvoert, maar kiest voor twee parallelle scans, wat in beide gevallen in een lager aantal in te lezen rijen resulteert in vergelijking met de queryplannen uit table 4. Opvallend is wel dat SQLite een '`AUTOMATIC COVERING INDEX`' op het 'driver'-attribuut definieert. Zo'n index wordt *at-runtime* door SQLite aangemaakt als berekend wordt dat de kost van het aanmaken van de index lager ligt dan de verwachte uitvoertijd van de query zonder index. De automatisch

aangemaakte index wordt ook onmiddellijk weer verwijderd zodra de query afgehandeld is.

Een eerste besluit is dus dat de oorzaak van het (veel) trager zijn van de vertalingen door de ASL-to-SQL-tool tweeledig is: ten eerste zijn er de semi-joins die ASL impliceert, dewelke er voor zorgen dat de ASL-to-SQL-tool de vertaling naar SQL uitvoert door geneste subqueries te genereren. Ten tweede is er het simuleren van een attribute-value dataspace met behulp van een semi-structured datamodel. De vervierdubbeling van het aantal aanwezige rijen heeft een versterkend effect op de tablescans ter uitvoering van die subqueries.

Een tweede invalshoek op de resultaten van de benchmarks is het bekijken van de impact van de aanwezigheid van indices, zowel bij de database- als de dataspace-instanties. Ook is het interessant om te zien hoe beide RDBMS's omgaan met indices in combinatie met het SQL-keyword `LIKE` en SQL-wildcards bij het matchen van strings. Wat volgt is een overzicht van alle toegepaste indices met daarbij welke invloed ze hebben op de uitvoertijden en op welke manier ze door SQLite en DB2 gebruikt worden in de achterliggende queryplannen. De SQL-definitie van de indices zijn terug te vinden in section 7.3.2.

1. 'importance_index': het importance-attribuut van de database-instantie wordt enkel in (de vertalingen van) 'query 1' gebruikt en het toepassen van een index op dat attribuut levert een geringe tijdswinst op. Die tijdswinst is te danken aan het feit dat zowel SQLite als DB2 een snelle *indexscan* uitvoeren i.p.v. een tablescan:

```
a.  Doe een indexscan op R1 en gebruik daarbij de
index 'importance_index'.
```

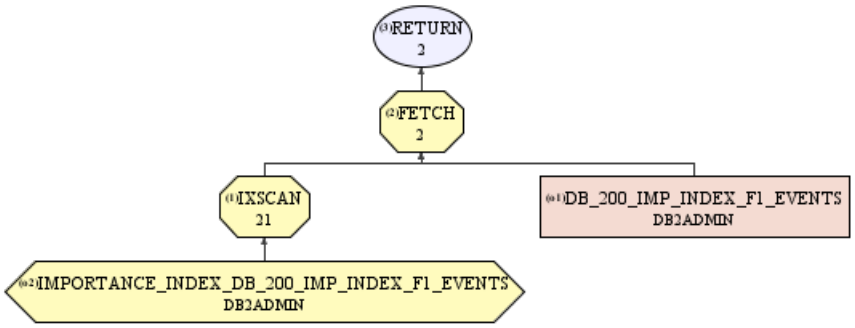

Table 6 – Queryplan van SQLite (boven) en DB2 (onder) van 'query 1', uitgevoerd op de 'database'-instantie met een index op het 'importance'-attribuut.

2. 'oid_index': het 'oid'-attribuut van de gesimuleerde AV-dataspace maakt deel uit van de joinconditie van elke semi-join in zowel query 1 als 2 (zie beide vertalingen b. en c. in section 7.3.1). Het definiëren van een index op een attribuut dat gebruikt wordt bij het joinen van relaties levert normaliter een flinke snelheidswinst op. Die verwachting wordt dan ook volledig ingelost, getuige daarvan de sterk positief beïnvloede meetresultaten. De aanwezigheid van 'oid_index' is uiteraard ook terug te vinden in de queryplannen; neem bijvoorbeeld dat van de uitvoering van query 1 op de dataspace-instantie door DB2. De query in kwestie:

```
SELECT R1.oid FROM DS_200_OID_INDEX_F1_EVENTS
R1 WHERE R1.attribute_name = 'setting' AND
R1.attribute_value = 'crash' AND EXISTS (SELECT NULL FROM
DS_200_OID_INDEX_F1_EVENTS R2 WHERE R1.oid = R2.oid AND
R2.attribute_name = 'importance' AND R2.attribute_value =
'10');
```

levert het volgende queryplan op:

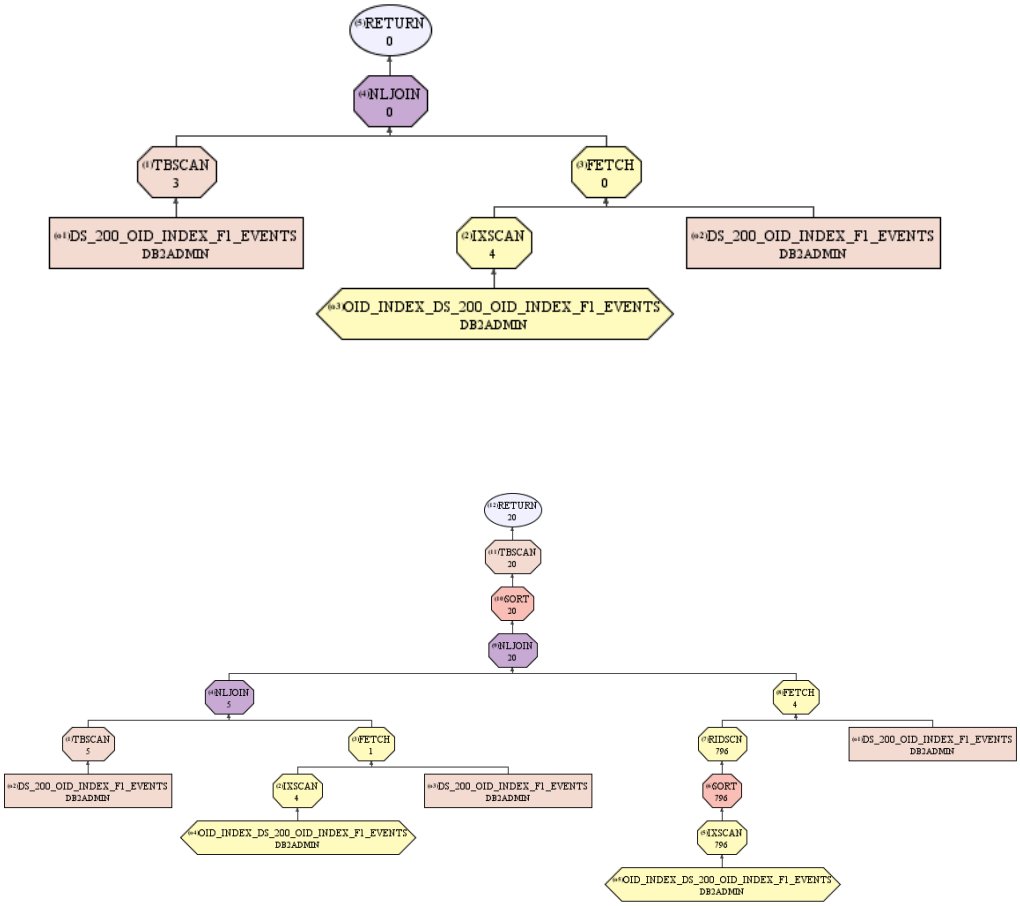

Figure 36 – Queryplannen van DB2 voor de uitvoering van query 1 (bovenste afbeelding) en query 2 (onderste afbeelding) op de 'dataspace'-instantie met een index op het 'oid'-attribuut.

Merk op[3] hoe de joinoperaties gebruik maken van de aanwezigheid van de index. Een resultaat dat echter sterk uit de toon valt, is dat van de uitvoering van 'query 2' op de DB2-dataspace-instantie van 200000 records. De oorzaak van dit onverwachte

[3]Een bijkomend verschil tussen de verschillende DB2-queryplannen zijn de verschillende joinalgoritmes die het RDBMS kan kiezen om twee relaties mee te joinen. In deze thesis komen er twee aan bod: 'HSJOIN' en 'NLJOIN', wat respectievelijk staat voor 'Hash-join' en 'Nested-loop-join'. Het daadwerkelijk bepalen welk algoritme voor een bepaalde join gebruikt zal worden, gebeurt door de query optimizer die deel uitmaakt van het RDBMS.

resultaat is terug te vinden in het gegenereerde queryplan:

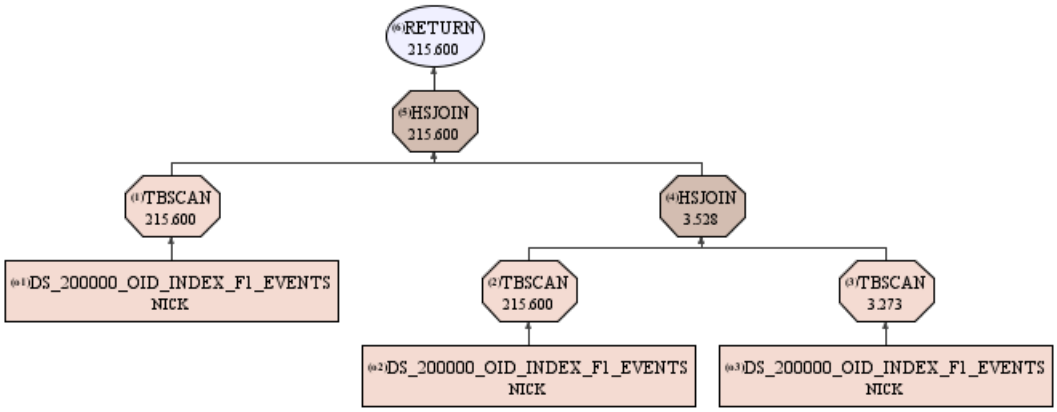

Figure 37 – Queryplan van de uitvoering van 'query 2' op de DB2-'dataspace'-instantie met een index op het 'oid'-attribuut.

Zoals te zien, beslist DB2 om in dit geval de snellere indexscans volledig achterwege te laten en in de plaats daarvan tot driemaal toe te opteren voor een full tablescan. Het motief achter deze beslissing is helaas ongekend, maar resulteert wel in een onverwacht slecht resultaat.

3. 'team_index', 'driver_index' en 'setting_index': deze drie indices zijn enkel definieerbaar op het *native* relationele model en zorgen er voor dat er slechts een klein verschil in de resultaten waargenomen kan worden indien ze daadwerkelijk aanwezig zijn. Zo gebruiken zowel SQLite als DB2 'setting_index' voor het uitvoeren van een indexscan bij de uitvoering van 'query 1'. Getuige daarvan zijn de volgende queryplannen:

```
a.  Doe een indexscan op R1 en gebruik daarbij de
index 'setting_index'.
```

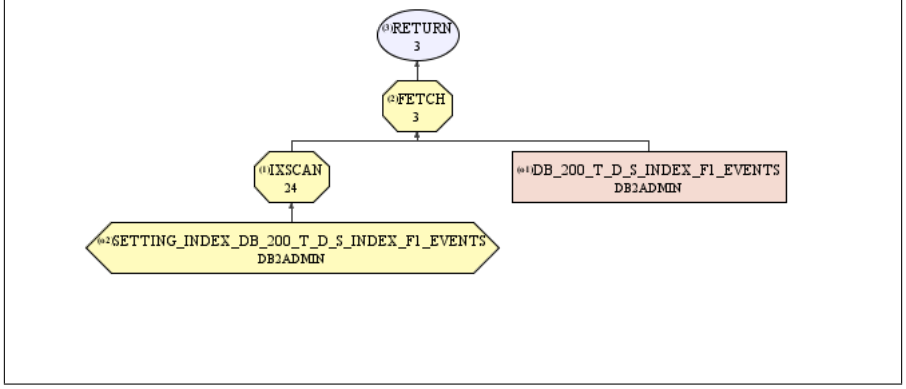

Table 7 – Queryplannen van SQLite (boven) en DB2 (onder) voor de uitvoering van 'query 1' op de 'database'-instantie met indices op de attributen 'team', 'driver' en 'setting'.

Ook de uitvoering van 'query 2' wordt op dezelfde manier beïnvloed, zij het nu door indexscans van zowel 'setting_index' als 'driver_index':

```
a.  Doe een indexscan op R2 en gebruik daarbij de
index 'setting_index'.
b.  Doe een indexscan op R1 en gebruik daarbij de
index 'driver_index'.
```

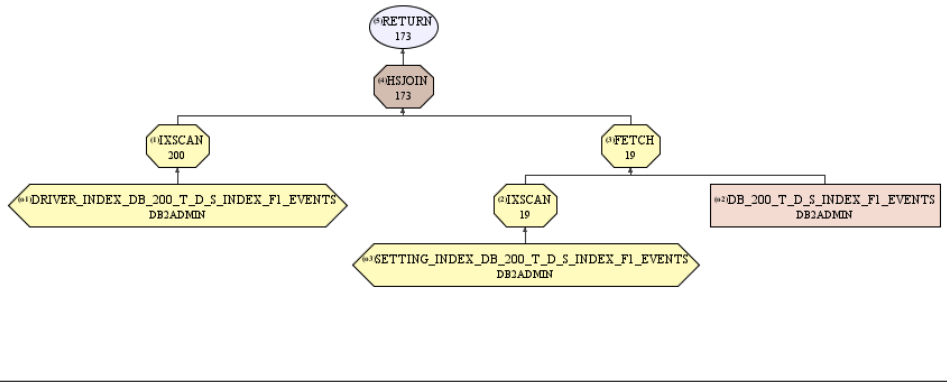

Table 8 – Queryplannen van SQLite (boven) en DB2 (onder) voor de uitvoering van 'query 2' op de 'database'-instantie met indices op de attributen 'team', 'driver' en 'setting'.

Een belangrijke vaststelling echter dat de aanwezigheid van de SQL-operator `LIKE` met daarbij een SQL-wildcard als eerste karakter van de vergelijkende string er voor zorgt dat zowel DB2 als SQLite een eventuele index op het te vergelijken attribuut negeren. Ter bewijs daarvan het DB2-queryplan van de uitvoering van 'query 2' met de SQL-operator `LIKE` op de 'database'-instantie:

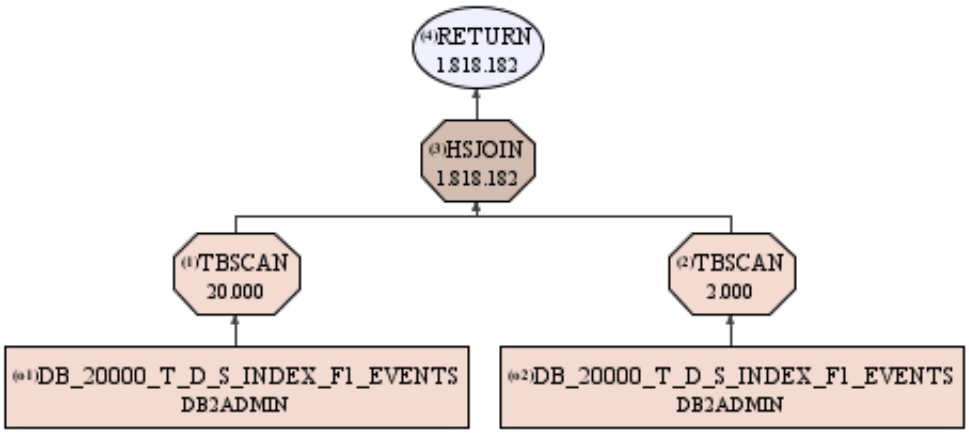

Figure 38 – DB2-queryplan van de uitvoering van 'query 2' met de SQL-operator `LIKE` op de 'database'-instantie.

Zoals opgemerkt worden de indices genegeerd en worden er full tablescans in de plaats uitgevoerd. Een 'oplossing' die er voor zorgt dat de indices bij de aanwezigheid van de SQL-operator `LIKE` wel degelijk in rekening worden gebracht tijdens het uitvoeren van de query bij DB2, bestaat erin om te voorkomen dat het eerste karakter van de vergelijkende string de SQL-wildcard is. Om aan die voorwaarde te voldoen, werd de query

```
SELECT R1.driver FROM F1_Events R1, F1_Events R2 WHERE R1.driver
= R2.driver AND R2.setting LIKE '%world champion%';
```

ter experiment vervangen door

```
SELECT R1.driver FROM F1_Events R1, F1_Events R2 WHERE R1.driver
= R2.driver AND R2.setting LIKE 'world champion%';
```

En inderdaad, zoals verwacht resulteert de licht aangepaste variant in een queryplan dat wel degelijk terug gebruik maakt van indexscans op basis van indices wiens attributen met `LIKE` vergeleken worden:

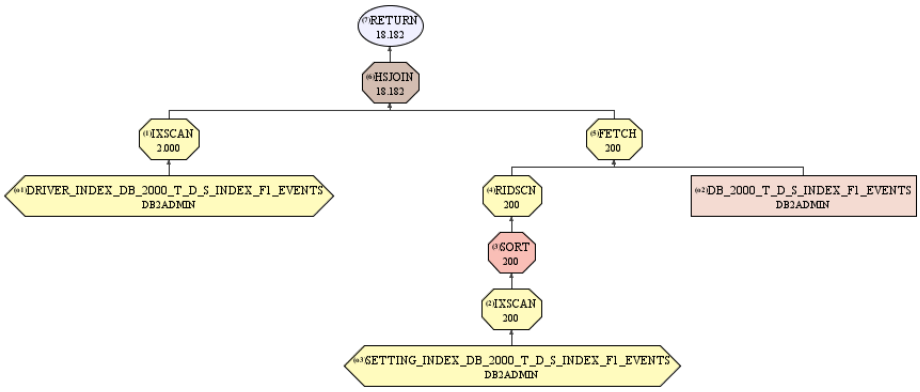

Figure 39 – DB2-queryplan van de uitvoering van de licht aangepaste variant van 'query 2' met de SQL-operator `LIKE` op de 'database'-instantie.

4. 'attribute_name_index' en 'attribute_value_index' zijn tenslotte twee indices die enkel kunnen voorkomen bij de 'dataspace'-instanties. Hun aanwezigheid levert het contra-intuïtieve en nagenoeg onverklaarbare gegeven op dat de uitvoertijden *hoger* liggen dan in het geval dat de indices niet bestaan. Ook de queryplannen geven wat dit betreft geen inzicht in de oorzaak van deze vaststelling:

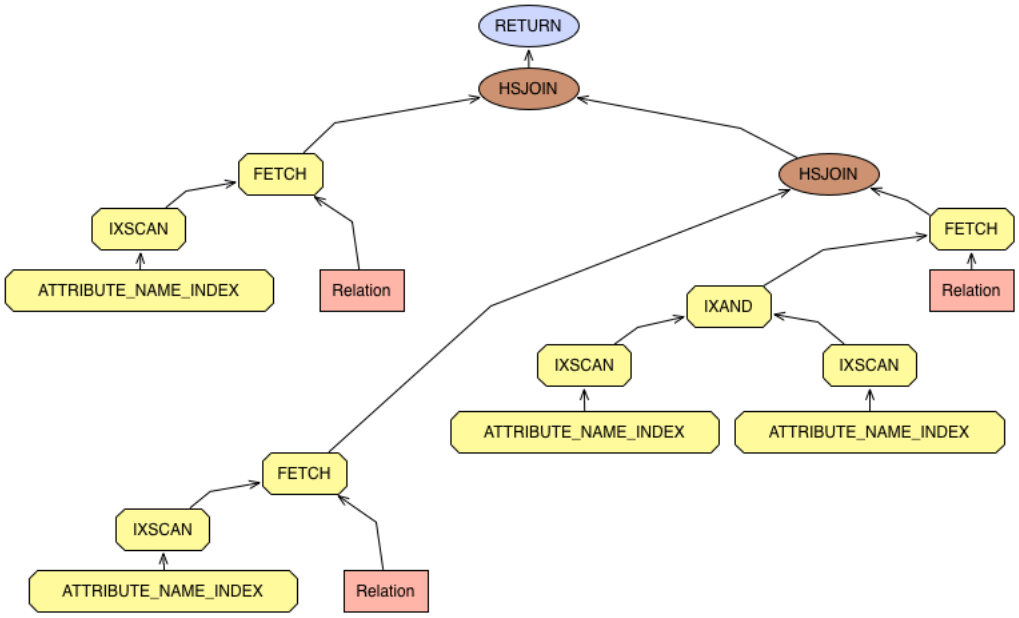

Figure 40 – DB2-queryplan van de uitvoering 'query 2' op de 'dataspace'-instantie met indices op de attributen 'oid', 'attribute_name' en 'attribute_value'.

Integendeel, de aanwezigheid van meerdere indexscans laat net vermoeden dat er wel degelijk een positief effect op de uitvoertijd waargenomen zal worden, maar het tegendeel is waar.

Tot slot is het nog meldenswaardig dat het attribuut 'attribute_value' net als de attributen horend bij de indices van het voorgaande puntje vergeleken kan worden door middel van het SQL-keyword `LIKE`. Omwille van die reden geldt ook hier de opmerking in verband met het weglaten van de eerste SQL-wildcard om de index in kwestie niet genegeerd te laten worden door DB2 tijdens de uitvoering van de query.

Voor een attribute-value dataspace gesimuleerd met behulp van een relationeel model zoals voorgesteld in section 7.3.2 kan dus best geopteerd worden voor de creatie van een index op het 'oid'-attribuut. Die index levert immers een forse prestatiewinst op (één enkele uitzondering daargelaten, zie puntje 2 uit de opsomming hier net boven). Van indices op de overige twee attributen wordt het best afgezien: de resultaten verslechteren, ook al nemen beide RDBMS's de indices mee in de uitvoering van de queries. Ook is een index op het 'attribute_value'-attribuut afhankelijk van de aanwezigheid van SQL-wildcards bij `LIKE`-vergelijkingen.

Het *native* relationeel model waarmee de gesimuleerde dataspace vergeleken wordt, baat

het meest bij de aanwezigheid van een index op die attributen die deel uitmaken van de eventuele joinconditie(s). Voor de gebruikte voorbeeldqueries betekent dat een index op het 'driver'- en 'setting'-attribuut.

8 Samenvatting

Voor de representatie en het doorzoekbaar maken van een heterogene gegevensverzameling bestaan er meerdere implementaties en technologieën. Zo voorzien op een concreet niveau het Search Computing-project en het iMeMex Personal Dataspace Management-project beide een omgeving die respectievelijk onderling verbonden real-world concepten en een computerbestandssysteem representeren. Elk van beide omgevingen is te doorzoeken: in het geval van SeCo met behulp van graafmanipulatie en in het geval van SeCo door het opstellen van een iQL-query. Op een iets hoger en abstracter niveau bevinden zich NoSQL-databasetechnologieën die elk een eigen interface aanbieden waarmee gegevens toegevoegd, opgeslagen of gewijzigd kunnen worden. In een poging om één uniforme interface voor NoSQL-databases te ontwikkelen, is UnQL ontstaan. UnQL is een querytaal die door het leven gaat als superset van SQL, maar kampt met serieuze adoptieproblemen. Het NoSQL-landschap is immers sterk gefragmenteerd en aan menig NoSQL-implementatie worden nog volop gesleuteld. Op het meest abstractie datarepresentatieniveau bevindt zich de attribute-value dataspace: een omgeving die bestaat uit een dataschemaloze verzameling van louter attribute-value paartjes. Opvallend is dat alle voorgaande manieren van gegevensrepresentatie (Search Computing, iMeMex PDMS en NoSQL-databasetechnologieën) te vormen zijn (of op zijn minst zeer sterk verwant) met deze attribute-value dataspaces. Het bevragen van een attribute-value dataspace kan met behulp van BSL en diens uitbreiding ASL. Behalve het modelleren van de zonet vermelde gegevensrepresentatietools aan de hand van een attribute-value dataspace, kunnen queries uit iQL en UNQL (resp. vanuit het iMeMex PDMS-project en NoSQL) als ASL-expressie geschreven worden.

Omdat het interessant is om een idee te krijgen van de performantie van ASL, kunnen er het best benchmarks uitgevoerd worden. Een weloverwogen beslissing werd gemaakt om attribute-value dataspaces en ASL respectievelijk te representeren door middel van een relationele database en te vertalen naar SQL. Om dat laatste te kunnen realiseren, is een tool geschreven die dus ASL-expressies omzet in SQL-queries. De benchmarks zelf bestaan uit het vergelijken van twee queries, beide zowel vertaald door de ASL-to-SQL-tool als manueel, respectievelijk uitgevoerd op een gesimuleerde attribute-value dataspace en een meer *native* relationeel model op twee verschillende RDBMS's (DB2 en SQLite). De impact van het simuleren van een attribute-value dataspace met behulp van een *semi-structured* model wordt besproken, net als de invloed van enkele indices op de resultaten aan de hand van queryplannen.

References

[1] Bozzon, A., Brambilla, M., Ceri, S. and Fraternali, P. Information Exploration in Search Computing. SeCO Workshop 2010: 10-25

[2] Dittrich, J., Vaz Salles, M. A. and Blunschi, L. iMeMex: From Search to Information Integration and Back. IEEE Data Eng. Bull. 32(2): 28-35 (2009)

[3] Dittrich, J., Blunschi, L., Girard, O. R., Karakashian, S. K. and Vaz Salles, M. A. A Dataspace Odyssey: The iMeMex Personal Dataspace Management System. CIDR 2007: 114-119

[4] Dittrich, J. and Vaz Salles, M. A. iDM: A Unified and Versatile Data Model for Personal Dataspace Management. VLDB 2006: 367-378

[5] Fletcher, G. H. L., Van den Bussche, J., Van Gucht, D. and Vansummeren, S. Towards a theory of search queries. ACM Trans. Database Syst. 35(4): 28 (2010)

[6] Strauch, C. NoSQL Databases. Selected Topics on Software-Technology Ultra-Large Scale Sites (Lecture). Stuttgart Media University, 2011, 149 p.

[7] Codd, E. F. A Relational Model of Data for Large Shared Data Banks. Commun. ACM 26(1): 64-69 (1983)

[8] Garcia-Molina, H., Ullman, J. D. and Widom, J. Database systems - the complete book (2. ed.). Pearson Education 2009, ISBN 978-0-13-187325-4, Chapter 20, pp. 993-996

[9] Nelson, D. B., Generalized Information Retrieval Language and System (GIRLS). User Requirements Specification. TRW (19 maart 1965) `http://www.tincat-group.com/mv/GIRLS.pdf`

[10] Tiwari, S., Professional NoSQL. John Wiley & Sons, Inc., ISBN: 978-0-470-94224-6, p. 4 (2011)

[11] Backus, J., Naur, P. Backus-Naur Form `http://www.garshol.priv.no/download/text/bnf.html`

[12] The OLAP Council. OLAP Whitepaper. 1997 `http://www.olapcouncil.org/research/whtpaply.htm`

[13] Ceri, S. and Brambilla, M. Search Computing - Trends and Developments [outcome of the second SeCO Workshop on Search Computing, Como/Milan, Italy, May 25-31, 2010]. Lecture Notes in Computer Science 6585, Springer 2011, ISBN 978-3-642-19667-6

[14] Doug, J. Hypertable. Presentation at NoSQL meet-up in San Francisco on 2009-06-11. `http://static.last.fm/johan/nosql-20090611/hypertable_nosql.pdf`

[15] Garcia-Molina, H., Ullman, J. D. and Widom, J. Database systems - the complete book (2. ed.). Pearson Education 2009, ISBN 978-0-13-187325-4, Chapter 20, pp. 985-1035

[16] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26(2) (2008)

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Organisatie en doorzoeken van een gegevensverzameling

Richting: **master in de informatica-databases**
Jaar: **2014**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Gaens, Nick

Datum: **5/09/2014**