UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Set joins in database query processing

Filip Van Assche

Scriptie ingediend tot het behalen van de graad van master in de informatica, afstudeerrichting databases

PROMOTOR :

Prof. dr. Jan VAN DEN BUSSCHE

2017
2018

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Set joins in database query processing

Filip Van Assche

Scriptie ingediend tot het behalen van de graad van master in de informatica, afstudeerrichting databases

PROMOTOR :

Prof. dr. Jan VAN DEN BUSSCHE

Set joins in database query processing

Masterproef voorgedragen tot het behalen van de graad van master in de informatica

Van Assche Filip

Promotor: Prof. dr. Jan Van den Bussche

Academiejaar: 2017 - 2018

Universiteit Hasselt

27 augustus 2018

Abstract

Deze masterproef gaat over de verschillende soorten set joins en de focus ligt op de set containment join. Beschouw twee database-relaties waarbij één van de attributen van elke entry een verzameling is. Dan kan het interessant zijn om na te gaan of er verzamelingen van de ene database-relatie, deelverzameling zijn van verzamelingen van de andere database-relatie. Set containment join algoritmes geven hierop het antwoord; zij vormen de focus van deze masterproef.

De verschillende soorten set joins worden in de relationele algebra uitgedrukt, maar het is niet efficiënt om die uitdrukkingen te gebruiken voor grote relaties. In de literatuur zijn verschillende set containment join algoritmes beschreven die dit probleem proberen oplossen op een efficiënte manier. Deze algoritmes worden beschreven, geïmplementeerd in Java, geanalyseerd en getest door middel van experimenten. De run time van deze algoritmes kan gebruikt worden om hun performantie te meten. Het resultaat is dat de keuze voor een bepaald algoritme afhangt van de parameters van de relatie, maar dat er doorgaans behoorlijk veel tijdwinst kan geboekt worden door de keuze van het juiste algoritme.

De set joins worden daarnaast ook in de querytalen SQL en XQuery uitgedrukt en de resultaten hiervan worden besproken. Tenslotte vergelijken we deze resultaten met de experimenten van de Java-implementatie van de geïmplementeerde set containment join algoritmes.

Voorwoord

Ik heb dit onderwerp gekozen omdat ik altijd al geïnteresseerd ben geweest in databases. Deze interesse werd onder meer vergroot door de vakken “Geavanceerde databasetechnologie” en “Databasesysteemarchitectuur”. Ik zou graag mijn familie, vrienden en vriendin Sarah bedanken voor alle steun die ik van hen gekregen heb. Daarnaast zou ik heel graag mijn promotor Prof. dr. Jan Van den Bussche willen bedanken voor het vertrouwen en de uitstekende opvolging van deze masterproef.

Inhoudsopgave

1 Introductie **1**

2 Set joins **3**
2.1 Het relationele model . . . 3
2.2 Key- en set-valued attribuut . . . 4
2.2.1 Key . . . 4
2.2.2 Set-valued . . . 5
2.2.3 Voorbeelden . . . 5
2.3 Set join . . . 7
2.3.1 Set containment join . . . 8
2.3.2 Division . . . 9
2.3.3 Set equality join . . . 10
2.3.4 Standaard equijoin en set overlap join . . . 11
2.3.5 Set disjointness join . . . 14
2.4 Joins en set joins in de klassieke relationele algebra . . . 15
2.4.1 Division . . . 15
2.4.2 Set containment join . . . 16
2.4.3 Set disjointness join . . . 17
2.4.4 Set equality join . . . 18
2.4.5 Set overlap join . . . 18
2.4.6 Tijdscomplexiteit . . . 18

3 Algoritmes voor set containment join **19**
3.1 Nested Loops (NL) . . . 20
3.2 Signature Nested Loops (SNL) . . . 21
3.2.1 Signature . . . 21
3.2.2 SNL . . . 23
3.3 Partitioned Set Join (PSJ) . . . 27
3.4 Indexed Nested Loops (INL) . . . 34
3.5 Inverted File Join (IFJ) . . . 39

4 Set joins in SQL en XQuery **45**
4.1 Set joins in SQL . . . 45
4.1.1 Set containment . . . 46
4.1.2 Division . . . 46
4.1.3 Set equality . . . 46
4.1.4 Set overlap (standaard equijoin) . . . 47
4.1.5 Set disjointness . . . 47
4.2 Set joins in XQuery . . . 48
4.2.1 Set containment . . . 50
4.2.2 Division . . . 50
4.2.3 Set equality . . . 50
4.2.4 Set overlap (standaard equijoin) . . . 51
4.2.5 Set disjointness . . . 51
4.3 Performantie van de set containment query in SQL en XQuery . 52

5 Experimenten **55**
5.1 Maatstaf voor performantie 55
5.2 Systeemspecificaties 55
5.3 Individuele bespreking van de verschillende algoritmes 55
5.3.1 Het NL-algoritme 55
5.3.2 Het SNL-algoritme 56
5.3.3 Het PSJ-algoritme 57
5.3.4 Het INL-algoritme 57
5.3.5 Het IFJ-algoritme 58
5.4 Vergelijking van de verschillende algoritmes 59

6 Conclusie **61**

1 Introductie

Deze masterproef gaat over **set joins** toegepast op database-relaties. Een doorsnee database-relatie heeft attributen die een waarde voorstellen. Maar men kan een attribuut van een relatie ook beschouwen als een set van waarden. Zo'n attribuut wordt een **set-valued attribuut** genoemd. De set join is dan een join tussen twee relaties op hun set-valued attribuut.

Er bestaan verschillende set joins. Een bepaalde set join wordt gedefinieerd aan de hand van zijn join-predikaat. Het meest voorkomende predikaat is set containment en dit wordt afgebeeld door het symbool $\subseteq$. Als voorbeeld beschouwen we de database-relaties "Drinker","Café" en "Biermerk" die geïllustreerd worden in Tabel 1. Elke drinker heeft een aantal bieren die hij/zij graag drinkt. Elk café biedt een aantal verschillende bieren aan. We gaan nu op zoek naar de cafés die alle bieren aanbieden die gedronken worden door een bepaalde drinker. Tabel 2 illustreert de werking van deze set containment join.

Drinker

Naam	**Bier**
Peter	Jupiler Leffe Blond
Frank	Cristal Maes
Tom	Orval Maes

Café

Naam	**Bier**
Koestal	Maes Duvel
Poeskaffee	Cristal Orval Maes
Bierpunt	Maes Cristal Jupiler

Biermerk

Bier
Cristal
Maes

Tabel 1: Bierdrinkers, cafés en bieren.

Drinker.Naam	**Café.Naam**
Frank	Poeskaffee
Frank	Bierpunt
Tom	Poeskaffee

Tabel 2: Output van $Drinker \underset{Drinker.Bier \subseteq Café.Bier}{\bowtie} Café$

Een speciale variant van de set-containment join is de division. Dit wordt afgebeeld door het symbool $\div$. Als we division toepassen op de relaties "Café" en "Biermerk", dan bestaat het resultaat uit alle cafés die minstens alle bieren uit de "Biermerk"-relatie aanbieden. Tabel 3 toont de werking van deze division.

Café.Naam
Poeskaffee
Bierpunt

Tabel 3: Output van $Café \div Biermerk$

We kunnen de division ook toepassen op de relaties "Drinker" en "Biermerk". Het resultaat van deze division bestaat uit alle drinkers die minstens alle bieren

uit de “Biermerk”-relatie graag drinken. Dit wordt geïllustreerd in tabel 4.

Drinker.Naam
Frank

Tabel 4: Output van *Drinker* ÷ *Biermerk*

Een set join kan berekend worden met de standaard relationele algebra. Deze berekening is echter niet efficiënt, want er worden resultaten geproduceerd die een kwadratische grootte hebben tegenover de grootte van de joinende relaties. Alhoewel dit kwadratische gedrag niet uit te sluiten valt, proberen enkele gespecialiseerde set containment join algoritmes dit probleem op te lossen. Het doel van deze masterproef is het bespreken en het implementeren van deze algoritmes en het onderling vergelijken van deze algoritmes op vlak van performantie. Bovendien wordt er ook achterhaald in welke mate set containment joins worden ondersteund in querytalen zoals SQL en XQuery en hun performantie wordt vergeleken met die van de set containment join algoritmes.

2 Set joins

Dit hoofdstuk behandelt set joins en de uitdrukking van set joins in de relationele algebra. We bespreken eerst de belangrijkste concepten van het relationele model. Dan volgt er een definitie van een set-valued attribuut, gevolgd door een uitgebreide bespreking van de verschillende soorten set joins. Tenslotte wordt besproken hoe joins en set joins uitgedrukt worden in de relationele algebra.

Een groot deel van dit hoofdstuk is gebaseerd op enkele papers. Paragraaf 2.2 en paragraaf 2.3 zijn gebaseerd op de introductie van [1].

2.1 Het relationele model

Een datamodel is een notatie die gebruikt wordt om data of informatie te beschrijven [7]. Het relationele model is wereldwijd het meest gebruikte datamodel. We bespreken de belangrijkste concepten van dit datamodel [9].

Een database bestaat uit verschillende relaties en elke relatie bevat data.

Een **relatie** is een subset van het cartesisch product van een lijst van domeinen die gekarakteriseerd worden door een naam. Een relatie wordt ook wel een **tabel** genoemd en zij wordt geïdentificeerd door haar **naam**.

We kunnen een logica beschrijven tussen een relatie en haar domeinen. Gegeven n domeinen die we D_1, D_2, $\ldots D_n$ noemen. Stel dat r een relatie is die gedefinieerd wordt op deze domeinen. Dan geldt: $r \subseteq D_1 \times D_2 \times \ldots \times D_n$.

Elke relatie bestaat uit een aantal rijen en een aantal kolommen. Een **kolom** in een relatie wordt ook wel een **attribuut** genoemd. Attributen zijn de belangrijkste opslageenheden van een database. Zij bevatten de inhoud van een relatie. De naam van het attribuut wordt gebruikt om de betekenis te beschrijven van de waarden van het attribuut.

Een **domein** bestaat uit de originele sets van atomaire waarden die gebruikt worden om data te modelleren. Met **atomaire waarde** bedoelen we dat elke waarde in het domein niet verder kan onderverdeeld worden in kleinere stukken [10]. Een domein is eigenlijk een verzameling van aanvaardbare waarden die een attribuut mag bevatten. Deze waarden zijn begrensd door bepaalde eigenschappen en door het datatype van het attribuut.

Een **rij** wordt vaak een **tuple** genoemd en bestaat uit een groep van datawaarden die met elkaar gerelateerd zijn. Een relatie bestaat dus uit een verzameling tuples. Een tuple heeft exact één component voor elk attribuut van de relatie.

Tuples en attributen vormen de basis van alle databases. Tabel 5 illustreert een relatie. We vermelden de verschillende onderdelen van deze relatie:

Persoon

Naam	Voornaam	Geboortedatum	Vermogen
Dekkers	Jan	28/01/1955	200 000
Jansens	Eline	21/8/1980	300 000
Peters	Ward	10/2/1990	100 000
Jakobs	Kathleen	2/1/1960	500 000

Tabel 5: De relatie "Persoon".

- De **naam** van de relatie is: "Persoon".
- De **attributen** van de relatie zijn: Naam, Voornaam, Geboortedatum en Vermogen.
- Het **domein** van Naam en Voornaam is respectievelijk de verzameling van strings die achternamen en voornamen van mensen voorstellen. Het domein van Geboortedatum is de verzameling van alle strings van de vorm dd/mm/jjjj van 1/1/1900 tot 31/12/1999. Het attribuut Geboortedatum heeft dus een datum als datatype en de waarden van dit attribuut zijn begrensd door een begin- en einddatum. Het domein van Vermogen is de verzameling van alle integers tussen 0 en 1 000 000. Dit attribuut heeft dus een integer als datatype en is ook begrensd door een minimum- en een maximumwaarde.
- De relatie bestaat uit 4 **tuples**. Dit zijn de 4 rijen die we kunnen aflezen van de tabel.

2.2 Key- en set-valued attribuut

Zoals reeds vermeld in paragraaf 2.1, bestaat een relatie uit attributen en tuples. Voor elke tuple in deze relatie, bevat een attribuut exact één waarde. Later zullen we zien dat we een attribuut ook kunnen beschouwen als een verzameling van waarden.

2.2.1 Key

De **key** is een attribuut of een groep van attributen, wiens waarden gebruikt worden om een tuple uniek te identificeren [11]. We hernemen de "Persoon"-relatie van tabel 5. Een mogelijke key voor deze relatie bestaat uit de attributen Naam en Voornaam, aangezien een persoon kan geïdentificeerd worden aan de hand van zijn volledige naam. Dit is echter niet volledig juist, want er bestaan personen met exact dezelfde naam. Er is dus nog geen sprake van unieke identificatie. Een betere key zou een extra PID-attribuut zijn. Het datatype van dit attribuut is integer en het doel van dit attribuut is het uniek identificeren van elke tuple. We beschouwen tabel 6 en we zien de "Persoon"-relatie met **PID** als key.

Persoon

PID	Naam	Voornaam	Geboortedatum	Vermogen
0	Dekkers	Jan	28/01/1955	200 000
1	Jansens	Eline	21/8/1980	300 000
2	Peters	Ward	10/2/1990	100 000
3	Jakobs	Kathleen	2/1/1960	500 000

Tabel 6: Relatie "Persoon" met key PID.

In een standaard database-relatie uit het relationele model, zoals relatie "Persoon" uit tabel 6, bevat elke tuple exact één waarde voor elk attribuut. Elke tuple moet dus uniek identificeerbaar zijn en in de praktijk gebeurt dit door een integer key-attribuut (zoals attribuut PID van tabel 6). Het is echter overzichtelijker en intuïtiever om tuples van elkaar te onderscheiden door middel van een naam in plaats van een ID. Dit zal duidelijk worden in de volgende paragraaf. Daarom zullen relaties vanaf nu zonder een ID key-attribuut voorgesteld worden (zoals in tabel 5), maar we houden in ons achterhoofd dat dat in de praktijk wel gebeurt (zoals in tabel 6). Vanaf nu zullen relaties dus voorgesteld worden met een naam als key-attribuut en we doen alsof dit de echte key is.

2.2.2 Set-valued

In dit werk beschouwen we relaties met twee attributen, van de algemene vorm $R(q, r)$. $R(q, r)$ is een verzameling van tuples. Elke tuple t heeft een waarde $t(q)$ en een waarde $t(r)$. We kunnen R dus beschouwen als een verzameling van koppeltjes van atomaire waarden. De relaties "Patiënt" en "Ziekte" uit tabel 7 zijn van deze vorm: elke tuple bestaat uit een (Naam, Symptoom)-koppeltje.

Attribuut q wordt het key-attribuut genoemd, maar dit moet niet strikt genomen worden: we laten wel degelijk verschillende tuples toe met dezelfde q-waarde. We gaan echter attribuut r beschouwen als een set (verzameling) van waarden, namelijk een **set-valued** attribuut. In dat perspectief kan attribuut q wel als key gebruikt worden. Op deze manier kunnen we R als een functie beschouwen die key-waarden q mapt op sets van r-waarden. Relatie R kunnen we dus beschouwen als een set-valued map. Aangezien attribuut q nu een geschikt key-attribuut is, kunnen we een handige notatie invoeren om deze beschouwing correct te meten:

$$R(q) = \{r \mid (q, r) \in R\}$$

Deze nieuwe beschouwing wordt geïllustreerd in tabel 8 en in tabel 9. De twee relaties "Patiënt" en "Ziekte" worden hier als een set-valued map voorgesteld: elke naam (van de patiënt of de ziekte) wordt gemapt op een verzameling van symptomen. We zien dat het attribuut Naam nu wel een geschikt key-attribuut is, want elke tuple heeft een verschillende Naam.

2.2.3 Voorbeelden

We illustreren het gebruik van een key-attribuut en een set-valued attribuut aan de hand van enkele voorbeelden: we beschouwen de relaties "Patiënt", "Ziekte" en "Symptoom" uit tabel 7.

Patiënt

Naam	Symptoom
An	hoofdpijn
An	keelpijn
An	nekpijn
Bob	hoofdpijn
Bob	keelpijn
Bob	geheugenverlies
Bob	nekpijn
Caroline	hoofdpijn
Jakob	hoofdpijn
Jakob	misselijkheid
Jakob	koorts

Ziekte

Naam	Symptoom
griep	hoofdpijn
griep	keelpijn
Lyme	hoofdpijn
Lyme	keelpijn
Lyme	geheugenverlies
Lyme	nekpijn
Malaria	hoofdpijn
Malaria	misselijkheid
Malaria	koorts
Hepatitis C	misselijkheid
Hepatitis C	koorts

Symptoom

Symptoom
hoofdpijn
nekpijn

Tabel 7: De relaties "Patiënt", "Ziekte" en "Symptoom".

We zouden het attribuut Naam in de relaties "Patiënt" en "Ziekte" als key kunnen nemen. Er zijn echter meerdere tuples met eenzelfde naam en een verschillend symptoom, dus dit is niet correct. We kunnen dit probleem oplossen door Symptoom als een set-valued attribuut te beschouwen.

We overlopen eerst de relatie "Patiënt". Voor elke waarde van Naam bepalen we de waarden van Symptoom die bij Naam horen, dus

$$Patiënt(Naam) = \{Symptoom \mid (Naam, Symptoom) \in Patiënt\}.$$

We passen deze schrijfwijze toe voor de vier verschillende namen en bekomen:

1. Patiënt(An) = {hoofdpijn, keelpijn, nekpijn}
2. Patiënt(Bob) = {hoofdpijn, keelpijn, geheugenverlies, nekpijn}
3. Patiënt(Caroline) = {hoofdpijn}
4. Patiënt(Jakob) = {hoofdpijn, misselijkheid, koorts}

Als we deze nieuwe schrijwijze toepassen, kunnen we de relatie "Patiënt" herformuleren. Dit wordt geïllustreerd in tabel 8.

Patiënt

Naam	Symptoom
An	hoofdpijn keelpijn nekpijn
Bob	hoofdpijn keelpijn geheugenverlies nekpijn
Caroline	hoofdpijn
Jakob	hoofdpijn misselijkheid koorts

Tabel 8: De relatie "Patiënt".

Nu beschouwen we de relatie "Ziekte":

$$Ziekte(Naam) = \{Symptoom \mid (Naam, Symptoom) \in Ziekte\}$$

en we bekomen:

1. Ziekte(griep) = {hoofdpijn, keelpijn}
2. Ziekte(Lyme) = {hoofdpijn, keelpijn, geheugenverlies, nekpijn}
3. Ziekte(Malaria) = {hoofdpijn, misselijkheid, koorts}
4. Ziekte(Hepatitis C) = {misselijkheid, koorts}

We herformuleren de relatie "Ziekte" door de nieuwe schrijfwijze toe te passen. Tabel 9 illustreert dit.

Ziekte

Naam	**Symptoom**
griep	hoofdpijn keelpijn
Lyme	hoofdpijn keelpijn geheugenverlies nekpijn
Malaria	hoofdpijn misselijkheid koorts
Hepatitis C	misselijkheid koorts

Tabel 9: De relatie "Ziekte".

Uit deze herformulering kunnen we afleiden dat Naam nu wel een geschikt key-attribuut is. Elke tuple heeft een verschillende naam en is dus uniek. We zien nu ook dat het attribuut Symptoom nu een verzameling van symptomen voorstelt in elke tuple, in plaats van één enkele waarde. We kunnen Symptoom dus nu een set-valued attribuut noemen.

2.3 Set join

Om de werking van de set join te verduidelijken, gebruiken we de twee relaties $R(q, r)$ en $S(s, t)$. De key-attributen zijn $R.q$ en $S.s$ en de set-valued attributen zijn $R.r$ en $S.t$.

Een **set join** is een join tussen twee relaties op hun set-valued attributen. We geven eerst de algemene definitie van een set join op relaties R en S:

$$R \underset{r\Theta t}{\bowtie} S = \{(q, s) \mid \exists r, \exists t : R(q, r) \wedge S(s, t) \wedge R(q)\ \Theta\ S(s)\}$$

In de definitie merken we Θ op. Dit is een join-predicaat en meer bepaald een **binair set-predicaat**. Het heeft dus twee sets als argumenten, namelijk het set-valued attribuut van elk relatie-argument, dus $R.r$ en $S.t$ in dit geval. Θ

legt dus een voorwaarde op waaraan beide set-valued attributen r en t moeten voldoen opdat hun bijhorende key-waarde q en s zou voorkomen in de output van de set join.

Het resultaat van elke set join bestaat telkens uit koppeltjes van keys van de twee joinende relaties, dus in dit geval koppeltjes van de vorm (q, s). Merk op dat we in deze definitie Θ kunnen vervangen door eender welk join-predicaat.

Set join operatoren

Er bestaan verschillende set join operatoren. De bekendste set join operator is de **set containment join** met join-predicaat $\subseteq$. Andere bekende set join operatoren zijn: division (zie 2.3.2), set equality en set disjointness. We bespreken ook de standaard equijoin die door middel van een set join operatie ($\cap \neq \emptyset$) kan uitgedrukt worden. Deze operatie wordt ook wel de set overlap join genoemd.

Voorbeelden

De verschillende soorten set joins zullen aan de hand van voorbeelden verduidelijkt worden. Hierbij baseren we ons op relatie "Patiënt" uit tabel 8, relatie "Ziekte" uit tabel 9 en relatie *Symptoom* uit tabel 7. We hernoemen deze relaties respectievelijk naar P, Z en S zodat de compactere schrijfwijze kan gebruikt worden bij het definiëren van de verschillende set joins. Zo wordt de definitie overzichtelijker. Tabel 10 geeft de relaties P, Z en S weer. We krijgen dus: $\rho_{P(n,s)}(Patiënt)$ en $\rho_{Z(n,s)}(Ziekte)$ en $\rho_{S(s)}(Symptoom)$.

P

n	**s**
An	hoofdpijn keelpijn nekpijn
Bob	hoofdpijn keelpijn geheugenverlies nekpijn
Caroline	hoofdpijn
Jakob	hoofdpijn misselijkheid koorts

Z

n	**s**
griep	hoofdpijn keelpijn
Lyme	hoofdpijn keelpijn geheugenverlies nekpijn
Malaria	hoofdpijn misselijkheid koorts
Hepatitis C	misselijkheid koorts

S

s
hoofdpijn
nekpijn

Tabel 10: De relaties P, Z en S.

2.3.1 Set containment join

Gegeven relaties $R(q, r)$ en $S(s, t)$ en met behulp van de notatie van paragraaf 2.2, kunnen we de **set containment join** als volgt definiëren:

$$R \underset{r \subseteq t}{\bowtie} S = \{(q, s) \mid \exists r, \exists t : R(q, r) \wedge S(s, t) \wedge R(q) \subseteq S(s)\}$$

We passen deze set containment join toe op de relaties Z en P uit paragraaf 2.3, dus we voeren $Z \underset{Z.s \subseteq P.s}{\bowtie} P$ uit. Deze set containment join geeft alle pa-

ren terug van de naam van de ziekte en de naam van de patiënt, waarbij de verzameling van symptomen van die ziekte een deelverzameling is van de verzameling van symptomen van die patiënt. Om dit te verduidelijken, gebruiken we de verzameling-notatie op het einde van paragraaf 2.2. We zien dat Z(griep) $\subseteq$ P(An), Z(griep) $\subseteq$ P(Bob), Z(Lyme) $\subseteq$ P(Bob), Z(Malaria) $\subseteq$ P(Jakob) en Z(Hepatitis C) $\subseteq$ P(Jakob). De output van deze set containment join is te zien in tabel 11.

Z.n	**P.n**
griep	An
griep	Bob
Lyme	Bob
Malaria	Jakob
Hepatitis C	Jakob

Tabel 11: Output van de set containment join $Z \underset{Z.s \subseteq P.s}{\bowtie} P$

Een ander voorbeeld is de set containment join $P \underset{P.s \subseteq Z.s}{\bowtie} Z$. Deze set containment join geeft dus alle paren terug van de naam van de patiënt en de naam van de ziekte, waarbij de verzameling van symptomen van die patiënt een deelverzameling is van de verzameling van symptomen van die ziekte. Volgens de verzameling-notatie van paragraaf 2.2 krijgen we: P(An) $\subseteq$ Z(Lyme), P(Bob) $\subseteq$ Z(Lyme), P(Caroline) $\subseteq$ Z(griep), P(Caroline) $\subseteq$ Z(Lyme), P(Caroline) $\subseteq$ Z(Malaria) en P(Jakob) $\subseteq$ Z(Malaria). De output van deze set containment join vinden we terug in tabel 12.

P.n	**Z.n**
An	Lyme
Bob	Lyme
Caroline	griep
Caroline	Lyme
Caroline	Malaria
Jakob	Malaria

Tabel 12: Output van de set containment join $P \underset{P.s \subseteq Z.s}{\bowtie} Z$

2.3.2 Division

De division operator $\div$ wordt geassocieerd met de relationele algebra. Ondanks het feit dat deze operator uitdrukbaar is aan de hand van de kernoperatoren van de relationele algebra, wordt de division toch beschouwd als een aparte operator. Dit komt omdat de division wordt gebruikt om een veel voorkomende soort van nuttige queries voor te stellen. Merk op dat dit ook geldt voor set joins in het algemeen. Zo gebruiken we bijvoorbeeld de operator $\underset{r \subseteq t}{\bowtie}$ voor de set containment join op relaties $R(q, r)$ en $S(s, t)$ (zie paragraaf 2.3.1).

We bespreken de werking van de division aan de hand van een voorbeeld. Beschouw twee relaties $R(q, r)$ en $S(t)$, waarbij attributen r en t hetzelfde domein hebben. Dan definiëren we de **division operatie** $R \div S$ als de verzameling van

alle q-waarden, zodat voor elke t-waarde in een tuple van S, er een tuple (q, r) in R bestaat. Een concrete uitwerking van de division op relaties "Patiënt" en "Symptoom" uit tabel 7 wordt geïllustreerd in tabel 13. We zien inderdaad dat de waarden *hoofdpijn* en *nekpijn* uit relatie "Symptoom" zowel bij *An* als *Bob* uit relatie "Patiënt" voorkomen.

Patiënt.Naam
An
Bob

Tabel 13: Output van *Patiënt* $\div$ *Symptoom*

We kunnen division ook op een andere manier beschrijven: voor elke q-waarde in R, beschouwen we de set van r-waarden die voorkomen in de tuples van R met die q-waarde. Als deze set (alle t-waarden in) S bevat, dan zit de q-waarde in het resultaat van de division $R \div S$.

We gebruiken deze informatie en de set-valued notatie van paragraaf 2.2 om de **division** aan de hand van relaties $R(q, r)$ en $S(t)$ te definiëren:

$$R \div S = \{q \mid \exists r : R(q, r) \wedge S \subseteq R(q)\}$$

We werken opnieuw een concreet voorbeeld uit, maar deze keer met behulp van de set-valued notatie. Het resultaat van de division op P en S uit tabel 10 wordt geïllustreerd in tabel 14. Aangezien $S \subseteq P(An)$ en $S \subseteq P(Bob)$, is het resultaat $\{An, Bob\}$. We merken op dat tabel 13 en tabel 14 dezelfde output bevatten.

P.n
An
Bob

Tabel 14: Output van $P \div S$

De division is eigenlijk een speciale variant van de set containment join. Gegeven relaties $R(q, r)$ en $S(t)$. We voegen een extra key-attribuut s toe aan relatie S als "vulling" en we noemen de nieuwe relatie $S_1(s, t)$. Dit nieuwe attribuut s bestaat uit allemaal dezelfde key-waarden. Nu kunnen we de division uitvoeren door een set containment join toe te passen op S_1 en R en door tenslotte het attribuut s weg te projecteren:

$$R \div S = \pi_q(S_1 \underset{t \subseteq r}{\bowtie} R)$$

2.3.3 Set equality join

De set equality join kan uitgedrukt worden door middel van de set-valued notatie van paragraaf 2.2. Gegeven relaties $R(q, r)$ en $S(s, t)$, dan kunnen we de **set equality join** als volgt definiëren:

$$R \underset{r=t}{\bowtie} S = \{(q, s) \mid \exists r, \exists t : R(q, r) \wedge S(s, t) \wedge R(q) = S(s)\}$$

We merken op dat we de set equality join aan de hand van de set containment join kunnen definiëren:

$$R \underset{r=t}{\bowtie} S = R \underset{r \subseteq t}{\bowtie} S \cap R \underset{r \supseteq t}{\bowtie} S$$

We passen deze set equality join toe op de relaties P en Z uit paragraaf 2.3, dus we voeren $P \underset{P.s=Z.s}{\bowtie} Z$ uit. Deze set equality join geeft alle paren terug van de naam van de patiënt en de naam van de ziekte, waarbij de verzameling van symptomen van de patiënt exact gelijk is aan de verzameling van symptomen van de ziekte. Twee verzamelingen zijn gelijk als ze deelverzamelingen zijn van elkaar. Uit relaties P en Z leiden we af: P(Bob) $\subseteq$ Z(Lyme), Z(Lyme) $\subseteq$ P(Bob), P(Jakob) $\subseteq$ Z(Malaria) en Z(Malaria) $\subseteq$ P(Jakob). De output van deze set equality join wordt geïllustreerd in tabel 15.

P.n	**Z.n**
Bob	Lyme
Jakob	Malaria

Tabel 15: Output van de set equality join $P \underset{P.s=Z.s}{\bowtie} Z$

Opmerking
Onze notatie voor de set equality join $R \underset{r=t}{\bowtie} S$ wordt ook gebruikt om de standaard equijoin (op niet-set-valued attributen) $R \underset{r=t}{\bowtie} S$ uit te drukken (zie paragraaf 2.3.4).

Deze notatie is verwarrend, maar dit is het enige mogelijke misverstand tussen de notatie van set joins en standaard joins. Er zal steeds duidelijk vermeld worden of het om een standaard equijoin of een set equality join gaat, zodat er geen verwarring mogelijk is.

2.3.4 Standaard equijoin en set overlap join

De **standaard** equijoin op twee relaties $R(q, r)$ en $S(s, t)$ is een speciaal geval van de join operatie, waarbij de join-conditie van de vorm $R.r = S.t$ is en dus uitsluitend uit gelijkheden (=) bestaat tussen twee attributen van R en S [8]. We spreken van de "standaard" equijoin omdat het hier gaat om de gewone gelijkheid (=) tussen niet-set-valued attribuutwaarden (r en t). De standaard equijoin kunnen we als volgt definiëren:

$$R \underset{r=t}{\bowtie} S = \{(q, r, s, t) \mid R(q, r) \wedge S(s, t) \wedge r = t\}$$

We tonen de werking van deze equijoin aan met behulp van een voorbeeld. We gebruiken de relaties "Patiënt" en "Ziekte" uit tabel 7 en we passen de equijoin $Patiënt \underset{Patiënt.Symptoom=Ziekte.Symptoom}{\bowtie} Ziekte$ toe. Tabel 16 toont de output van deze equijoin.

Patiënt.Naam	Ziekte.Naam	Symptoom
An	griep	hoofdpijn
An	Lyme	hoofdpijn
An	Malaria	hoofdpijn
An	griep	keelpijn
An	Lyme	keelpijn
An	Lyme	nekpijn
Bob	griep	hoofdpijn
Bob	Lyme	hoofdpijn
Bob	Malaria	hoofdpijn
Bob	griep	keelpijn
Bob	Lyme	keelpijn
Bob	Lyme	geheugenverlies
Bob	Lyme	nekpijn
Caroline	griep	hoofdpijn
Caroline	Lyme	hoofdpijn
Caroline	Malaria	hoofdpijn
Jakob	griep	hoofdpijn
Jakob	Lyme	hoofdpijn
Jakob	Malaria	hoofdpijn
Jakob	Malaria	misselijkheid
Jakob	Hepatitis C	misselijkheid
Jakob	Malaria	koorts
Jakob	Hepatitis C	koorts

Tabel 16: Output van de standaard equijoin $Patiënt \underset{Patiënt.Symptoom=Ziekte.Symptoom}{\bowtie} Ziekte$

Uit tabel 16 leiden we af dat het attribuut Symptoom slechts één keer voorkomt in de output, terwijl het in principe twee keer zou moeten voorkomen volgens de definitie, namelijk Patiënt.Symptoom en Ziekte.Symptoom. Dit komt doordat de equijoin overbodige attributen weglaat uit het resultaat. Een overbodig attribuut is een attribuut dat dezelfde naam en dezelfde waarde heeft als een ander attribuut in het resultaat, dus Patiënt.Symptoom en Ziekte.Symptoom in dit geval.

Aangezien we alleen geïnteresseerd zijn in de koppels (Patiënt.Naam, Ziekte.Naam), projecteren we alleen deze attributen en laten we het Symptoom-attribuut weg. Het resultaat wordt geïllustreerd in tabel 17.

Patiënt.Naam	Ziekte.Naam
An	griep
An	Lyme
An	Malaria
Bob	griep
Bob	Lyme
Bob	Malaria
Caroline	griep
Caroline	Lyme
Caroline	Malaria
Jakob	griep
Jakob	Lyme
Jakob	Malaria
Jakob	Hepatitis C

Tabel 17: Output van $\pi_{(Patiënt.naam, Ziekte.Naam)}(Patiënt \underset{Patiënt.Symptoom=Ziekte.Symptoom}{\bowtie} Ziekte)$

We gaan nu de **set overlap join** definiëren en we tonen aan dat deze set join gelijkaardig is aan de standaard equijoin.

Met behulp van de set-valued notatie van paragraaf 2.2 en gegeven relaties $R(q, r)$ en $S(s, t)$, dan kunnen we de **set overlap join** als volgt definiëren:

$$R \underset{r \cap t \neq \emptyset}{\bowtie} S = \{(q, s) \mid \exists r, \exists t : R(q, r) \wedge S(s, t) \wedge R(q) \cap S(s) \neq \emptyset\}.$$

De set overlap join kan dus beschouwd worden als het joinen van twee relaties op hun set-valued attribuut waarbij er minstens één element van beide set-valued attributen moet overlappen: hun doorsnede mag niet leeg zijn. Deze beschrijving is zeer gelijkaardig aan die van de standaard equijoin: minstens één attribuutwaarde moet gelijk zijn. Er is slechts een klein verschil tussen de standaard equijoin en de set overlap join: in de output van de standaard equijoin komen alle attributen voor van de joinende relaties, terwijl de set overlap join alleen de koppeltjes van keys (q, s) teruggeeft in de output. Als we dus de key attributen projecteren uit de output van de standaard equijoin, bekomen we de output van de set overlap join. We kunnen dit zo beschrijven:

$$R \underset{r \cap t \neq \emptyset}{\bowtie} S = \pi_{q,s}(R \underset{r=t}{\bowtie} S)$$

De set overlap join is dus een projectie van de standaard equijoin.

We illustreren het verband tussen de equijoin en de set overlap join aan de hand van een voorbeeld. We passen de set overlap join toe op relaties P en Z uit tabel 10: $P \underset{P.s \cap Z.s \neq \emptyset}{\bowtie} Z$.

Deze set overlap join geeft alle paren terug van de naam van de patiënt en de naam van de ziekte, waarbij de set van symptomen van die patiënt minstens één gemeenschappelijk element moet hebben met de set van symptomen van de ziekte. De doorsnede van beide sets van symptomen mag dus niet leeg zijn: er moet dus minstens één symptoom overlappen.

We werken $P \underset{P.s \cap Z.s \neq \emptyset}{\bowtie} Z$ concreet uit. De output van deze standaard equijoin wordt geïllustreerd in tabel 18.

P.n	Z.n
An	griep
An	Lyme
An	Malaria
Bob	griep
Bob	Lyme
Bob	Malaria
Caroline	griep
Caroline	Lyme
Caroline	Malaria
Jakob	griep
Jakob	Lyme
Jakob	Malaria
Jakob	Hepatitis C

Tabel 18: Output van de set overlap join $P \underset{P.s \cap Z.s \neq \emptyset}{\bowtie} Z$

We zien dat de inhoud van tabellen 17 en 18 volledig overeenkomen. Dit bevestigt dus dat de set overlap join een projectie is van de standaard equijoin.

2.3.5 Set disjointness join

Twee sets zijn disjunct als zij geen gemeenschappelijke elementen hebben, dus als hun doorsnede de lege verzameling $\emptyset$ is. Bij een set disjointness join gaan we dus twee relaties joinen op hun set-valued attribuut waarbij de doorsnede van beide sets leeg moet zijn.

We gebruiken opnieuw de set-valued notatie van paragraaf 2.2 om de set disjointness te definiëren. Gegeven relaties $R(q, r)$ en $S(s, t)$, dan kunnen we de **set disjointness join** als volgt definiëren:

$$R \underset{r \cap t = \emptyset}{\bowtie} S = \{(q, s) \mid \exists r, \exists t : R(q, r) \wedge S(s, t) \wedge R(q) \cap S(s) = \emptyset\}.$$

We voeren deze set disjointness join uit op relaties P en Z uit tabel 10, dus we voeren $P \underset{P.s \cap Z.s = \emptyset}{\bowtie} Z$ uit. Deze set disjointness join geeft alle paren terug van de naam van de patiënt en de naam van de ziekte, waarbij de set van symptomen van die patiënt disjunct is met de set van symptomen van de ziekte. Beide sets van symptomen mogen dus geen enkel element gemeenschappelijk hebben: hun doorsnede moet leeg zijn. We werken $P \underset{P.s \cap Z.s = \emptyset}{\bowtie} Z$ concreet uit en we zien dat: P(An) $\cap$ Z(Hepatitis C) $= \emptyset$, P(Bob) $\cap$ Z(Hepatitis C) $= \emptyset$ en P(Caroline) $\cap$ Z(Hepatitis C) $= \emptyset$. De output van deze set disjointness join wordt geïllustreerd in tabel 19.

P.n	Z.n
An	Hepatitis C
Bob	Hepatitis C
Caroline	Hepatitis C

Tabel 19: Output van de set disjointness join $P \underset{P.s \cap Z.s = \emptyset}{\bowtie} Z$

2.4 Joins en set joins in de klassieke relationele algebra

De relationele algebra is een formele querytaal die geassocieerd is met het relationele model. In de relationele algebra kunnen queries geformuleerd worden door verschillende operatoren te combineren. Elke operator in de relationele algebra heeft steeds één of twee relatie-instanties als argumenten en geeft als resultaat een relatie-instantie terug [8].

De *join*-operatie is een enorm nuttige en veelgebruikte operatie in de relationele algebra. Een join kan eigenlijk gedefinieerd worden als een cartesisch product $\times$ in combinatie met enkele selecties σ en projecties π. In de praktijk is het zo dat de join-operatie veel meer voorkomt dan het gewone cartesisch product. Daarnaast is het resultaat van een cartesisch product meestal veel groter dan het resultaat van een join en het is ook belangrijk om joins te herkennen en ze te implementeren zonder de onderliggende cartesische producten uit te voeren. Om deze redenen is de join een 'vaste' operatie in de relationele algebra, met een eigen notatie: $R \underset{c}{\bowtie} S$. In het algemeen heeft de join-operatie een *join-conditie* c die gelijkaardig is aan een selectie-conditie, twee relatie-instanties als argumenten en één relatie-instantie als output. De *join*-operatie wordt zo gedefinieerd in de klassieke relationele algebra:

$$R \underset{c}{\bowtie} S = \sigma_c(R \times S)$$

We zien dus dat $\bowtie$ gedefinieerd wordt als een cartesisch product gevolgd door een selectie met conditie c waarbij c betrekking heeft op attributen van relaties R en S.

In de vorige paragraaf hebben wij de *set join* $R \underset{r \Theta t}{\bowtie} S$ gedefinieerd op relaties $R(q, r)$ en $S(s, t)$ en meermaals gebruikt. Net zoals de *standaard join*, kunnen wij ook de *set join* uitdrukken in de relationele algebra. We maken hierbij gebruik van de relationele algebra-operatoren: selectie $\sigma_c(R)$, projectie $\pi_r(R)$, renaming $\rho_{R(x,y)}(R)$, en de set operatoren unie $R \cup S$, intersectie $R \cap S$, verschil $R - S$ en het cartesisch product $R \times S$.

2.4.1 Division

We beschouwen de relaties $R(q, r)$ en $S(t)$ en we leggen eerst intuïtief uit hoe we de division kunnen uitdrukken in de relationele algebra en daarna geven we de relationele algebra-expressie zelf.

De output van de division moet bestaan uit de q-waarden van R die toegelaten zijn. Een q-waarde wordt niet toegelaten als we, door het toevoegen van een t-waarde uit S en deze waarde te hernoemen naar r, een tuple (q, r) bekomen

dat niet voorkomt in R. De relationele algebra-expressie voor niet toegelaten q-waarden is:

$$\pi_q((\pi_q(R) \times \rho_{S(r)}(S)) - R)$$

De output van de division bestaat uit alle q-waarden van R, behalve de niet toegelaten q-waarden. We passen dit toe en we bekomen de volledige relationele algebra-expressie voor deze division:

$$R \div S = \pi_q(R) - \pi_q((\pi_q(R) \times \rho_{S(r)}(S)) - R)$$

2.4.2 Set containment join

We hanteren dezelfde aanpak als de division om de set containment join uit te drukken in de relationele algebra: we maken gebruik van een intuïtieve uitleg om de volledige relationele algebra-expressie geleidelijk aan op te bouwen. Relaties $R(q, r)$ en $S(s, t)$ met set-valued attributen r en t en key-attributen q en s worden hierbij gebruikt.

De output van de set containment join $R \underset{r \subseteq t}{\bowtie} S$ bestaat uit de verzameling van alle mogelijke koppeltjes (q, s) die toegelaten zijn. Een koppeltje (q, s) is niet toegelaten als het behoort tot de output van $R \underset{r \not\subseteq t}{\bowtie} S$. De toegelaten koppeltjes (q, s) bestaan uit alle koppeltjes (q, s) buiten de koppeltjes die niet toegelaten zijn. Als we weten dat we alle koppeltjes (q, s) kunnen uitdrukken door $\pi_q(R) \times \pi_s(S)$, dan kunnen deze stap alvast uitdrukken in de relationele algebra:

$$R \underset{r \subseteq t}{\bowtie} S = \pi_q(R) \times \pi_s(S) - R \underset{r \not\subseteq t}{\bowtie} S$$

Het koppeltje (q, s) behoort tot $R \underset{r \not\subseteq t}{\bowtie} S$, als voor diens bijhorende r en t geldt dat r geen subset is van t. Concreet gaan we dus op zoek naar koppeltjes (q, s) uit $R(q, r)$ en $S(s, t)$, waarvoor er een r bestaat (die bij q hoort), die gelijk is aan geen enkele t die bij s hoort. Als we deze niet-containment proberen uitdrukken, krijgen we:

$$R \underset{r \not\subseteq t}{\bowtie} S = \{(q, s) \mid \exists r : R(q, r) \wedge \neg S(s, r)\}$$

We zien dat t niet voorkomt in deze uitdrukking. De klemtoon ligt hier op r, want r mag niet voorkomen in $S(s)$.

We gebruiken de projectie-operator π om attributen q en s te projecteren en we bekomen:

$$R \underset{r \not\subseteq t}{\bowtie} S = \pi_{q,s}(\{(q, r, s) \mid R(q, r) \wedge \neg S(s, r)\})$$

De verzameling triples $\{(q, r, s) \mid R(q, r) \wedge \neg S(s, r)\}$ kunnen we opnieuw beschouwen als alle triples (q, r, s) die toegelaten zijn. Een triple (q, r, s) wordt niet toegelaten als de voorwaarde $R(q, r) \wedge S(s, r)$ geldt voor die bepaalde q, r en s. De toegelaten triples (q, r, s) bestaan uit alle triples (q, r, s) buiten de triples die niet toegelaten zijn. Als we weten dat we alle triples (q, r, s) kunnen uitdrukken door $R \times \pi_s(S)$, dan krijgen we:

$$\{(q, r, s) \mid R(q, r) \wedge \neg S(s, r)\} = R \times \pi_s(S) - \{(q, r, s) \mid R(q, r) \wedge S(s, r)\}$$

We kunnen $\{(q, r, s) \mid R(q, r) \wedge S(s, r)\}$ uitdrukken in de relationele algebra door het cartesisch product van R en S te nemen, te verzekeren dat $r = t$ geldt en dan tenslotte q, r en s te projecteren. We bekomen:

$$\{(q, r, s) \mid R(q, r) \wedge \neg S(s, r)\} = R \times \pi_s(S) - \pi_{q,r,s}(\sigma_{r=t}(R \times S))$$

De volledige relationele algebra expressie voor de set containment join is dus:

$$R \underset{r \subseteq t}{\bowtie} S = \pi_q(R) \times \pi_s(S) - \pi_{q,s}(R \times \pi_s(S) - \pi_{q,r,s}(\sigma_{r=t}(R \times S)))$$

2.4.3 Set disjointness join

Om de set disjointness join uit te drukken in de relationele algebra, kunnen we opnieuw een gelijkaardige aanpak hanteren. We gebruiken opnieuw relaties $R(q, r)$ en $S(s, t)$.

De definitie van de set disjointness join op relaties R en S is:

$$R \underset{r \cap t = \emptyset}{\bowtie} S = \{(q, s) \mid \exists r, \exists t : R(q, r) \wedge S(s, t) \wedge R(q) \cap S(s) = \emptyset\}$$

De output van deze set disjointness join $R \underset{r \cap t = \emptyset}{\bowtie} S$ bestaat uit de verzameling van koppeltjes (q, s) die toegelaten zijn. Een koppeltje (q, s) is niet toegelaten als het behoort tot de output van $R \underset{r \cap t \neq \emptyset}{\bowtie} S$. De toegelaten koppeltjes (q, s) bestaan uit alle koppeltjes (q, s), buiten de koppeltjes die niet toegelaten zijn. We kunnen deze stap alvast uitdrukken in de relationele algebra:

$$R \underset{r \cap t = \emptyset}{\bowtie} S = \pi_q(R) \times \pi_s(S) - R \underset{r \cap t \neq \emptyset}{\bowtie} S$$

We merken meteen op dat de niet-disjointness join $R \underset{r \cap t \neq \emptyset}{\bowtie} S$ eigenlijk de set overlap join is, die wij reeds gezien hebben in paragraaf 2.3.4. We kunnen dus de definitie van de set overlap join gebruiken:

$$R \underset{r \cap t \neq \emptyset}{\bowtie} S = \{(q, s) \mid \exists r, \exists t : R(q, r) \wedge S(s, t) \wedge R(q) \cap S(s) \neq \emptyset\}$$

We gebruiken de projectie-operator π om attributen q en s te projecteren en we bekomen:

$$R \underset{r \cap t \neq \emptyset}{\bowtie} S = \pi_{q,s}(\{(q, r, s, t) \mid R(q, r) \wedge S(s, t) \wedge R(q) \cap S(s) \neq \emptyset\})$$

In paragraaf 2.3.4 hebben wij ook gezien dat de set overlap join eigenlijk een projectie is van de standaard equijoin:

$$R \underset{r \cap t \neq \emptyset}{\bowtie} S = \pi_{q,s}(R \underset{r=t}{\bowtie} S)$$

De definitie van de standaard equijoin is:

$$R \underset{r=t}{\bowtie} S = \{(q, r, s, t) \mid R(q, r) \wedge S(s, t) \wedge r = t\}$$

We kunnen dus de definitie van de standaard equijoin gebruiken om onze uitdrukking verder aan te vullen. We kunnen $\{(q, r, s, t) \mid R(q, r) \wedge S(s, t) \wedge r = t\}$

uitdrukken in de relationele algebra door het cartesisch product van R en S te nemen en te verzekeren dat $r = t$ geldt. We hoeven attributen q, r, s en t niet te projecteren aangezien het cartesisch product $R \times S$ alle attributen van R en S al teruggeeft. De volledige relationele algebra expressie voor de set disjointness join is:

$$R \underset{r \cap t = \emptyset}{\bowtie} S = \pi_q(R) \times \pi_s(S) - \pi_{q,s}(\sigma_{r=t}(R \times S))$$

2.4.4 Set equality join

Zoals reeds aangegeven in paragraaf 2.3.3, kunnen we de set equality join uitdrukken aan de hand van de set containment join:

$$R \underset{r=t}{\bowtie} S = R \underset{r \subseteq t}{\bowtie} S \cap R \underset{r \supseteq t}{\bowtie} S$$

We gebruiken de relationele algebra expressies van de set containment joins $R \underset{r \subseteq t}{\bowtie} S$ en $S \underset{t \subseteq r}{\bowtie} R$ en dan nemen we hun doorsnede om de volledige relationele algebra expressie van de set equality join $R \underset{r=t}{\bowtie} S$ te bekomen.

2.4.5 Set overlap join

In paragraaf 2.3.4 hebben we gezien dat de set overlap join een projectie is van de standaard equijoin:

$$R \underset{r \cap t \neq \emptyset}{\bowtie} S = \pi_{q,s}(R \underset{r=t}{\bowtie} S)$$

De relationele algebra expressie voor de standaard equijoin $R \underset{r=t}{\bowtie} S$ is:

$$\sigma_{r=t}(R \times S)$$

Dus de volledige relationele algebra expressie voor de set overlap join is:

$$R \underset{r \cap t \neq \emptyset}{\bowtie} S = \pi_{q,s}(\sigma_{r=t}(R \times S))$$

2.4.6 Tijdscomplexiteit

Na het uitdrukken van de set joins in de relationele algebra, merken we op dat er altijd opnieuw minstens één cartesisch product van relaties R en S in de uitdrukking voorkomt. Aangezien we elke tuple van beide relaties met elkaar vergelijken, wordt er een kwadratisch tussenresultaat geproduceerd. Dit is onvermijdbaar.

Nu is het zo dat er geen relationele algebra expressie bestaat voor de division of eender welke set join, die niet minstens één kwadratisch tussenresultaat produceert. Dit werd aangetoond in [1].

Dit is niet optimaal en daarom tracht men in de praktijk gespecialiseerde algoritmes te gebruiken om set joins efficiënter te berekenen. In hoofdstuk 3 zullen we deze algoritmes nauwkeurig bestuderen en hun performantie met elkaar vergelijken. De worst-case tijdscomplexiteit van deze algoritmes is weliswaar kwadratisch, maar het is alsnog beter om deze algoritmes te gebruiken dan het rechtstreeks evalueren van de vrij complexe relationele algebra expressies.

3 Algoritmes voor set containment join

In dit hoofdstuk worden enkele set containment join-algoritmes uitgebreid besproken. Hoofdstuk 2 voorziet alle informatie om de algoritmes te kunnen begrijpen.

Elk set-join algoritme dat hier wordt besproken, is gebaseerd op het gelijknamige algoritme uit een bepaalde paper. Paragraaf 3.2 en paragraaf 3.3 zijn gebaseerd op [3], paragraaf 3.4 is gebaseerd op [2] en paragraaf 3.5 is gebaseerd op [13] en op [2]. Sommige van deze papers verwijzen op hun beurt naar een paper waar het algoritme oorspronkelijk werd besproken. Tabel 20 geeft een chronologisch overzicht van deze oorspronkelijke papers en de datum waarop ze gepubliceerd zijn.

Voor het IFJ-algoritme kunnen we opmerken dat het oorspronkelijk door Mamoulis [2] werd ontwikkeld, maar hij vond het inefficiënt. Shaporenkov [13] herbekeek het algoritme later in de context van main-memory database management systems en herinterpreteerde de efficiëntie ervan. Vandaar dat beide referenties in de tabel voorkomen bij IFJ.

Algoritme	Oorspronkelijke referentie	Jaar
SNL	[4]	1997
PSJ	[3]	2000
INL	[2]	2003
IFJ	[2] en [13]	2003 en 2005

Tabel 20: Overzicht van de besproken set containment join algoritmes

De focus van de masterproef ligt op de **main-memory** set join algoritmes. Dit zijn set joins tussen twee relaties, zodat het main memory groot genoeg is om de gehele set join-berekening te behandelen. Onder de gehele set join berekening verstaan we zowel het raadplegen van alle tuples van beide joinende relaties, als de set join-berekening zelf.

Meestal worden tuples van een relatie geïdentificeerd door de row-id van die tuple. Een tuple van een relatie met een set-valued attribuut kan op een andere manier geïdentificeerd worden, namelijk door het **key-attribuut**. We illustereren dit aan de hand van de relatie $Persoon(naam, symptoom)$ uit tabel 8. Zoals gezegd is $naam$ hier het key-attribuut. Als we dit key-attribuut gebruiken om een tuple te identificeren, bestaat deze relatie eigenlijk uit 3 tuples: één tuple met key "An", één tuple met key "Bob" en één tuple met key "Caroline". In de bespreking van de verschillende main-memory set join-algoritmes zal deze manier gebruikt worden om de tuples van een relatie te identificeren.

We passen de set-join toe op de relaties R en S met set-valued attributen $R.r$ en $S.s$. We gebruiken de volgende notatie:

- t_R (t_S) is een tuple van relatie R (S)
- $t_R.key$ ($t_S.key$) is de waarde van het key-attribuut van t_R (t_S)

- n_R (n_S) is de kardinaliteit van de verzameling van waarden van key-attributen van R (S)

- $t_R.set$ ($t_S.set$) is de set van waarden die bij $t_R.key$ ($t_S.key$) horen

- A is de kardinaliteit van de unie van alle set-attributen van R en S gecombineerd

- $sig(t_R.set)$ ($sig(t_S.set)$) is de signature van $t_R.set$ ($t_S.set$)

- $|t_R.set|$ ($|t_S.set|$) is de kardinaliteit van $t_R.set$ ($t_S.set$), de gemiddelde waarden hiervan noteren we met B_R (B_S).

Eerst beschrijven we de werking van elk algoritme voor de set containment join $R \underset{r \subseteq s}{\bowtie} S$.

3.1 Nested Loops (NL)

Het Nested Loops algoritme (**NL**) is een eenvoudig algoritme dat als volgt werkt: voor elke tuple t_R van R en voor elke tuple t_S van S gaan we na of $t_R.set \subseteq t_S.set$. Dit is de eenvoudigste (en naïefste) methode om een set containment join te berekenen tussen twee relaties R en S.

Dit naïeve algoritme werd geïmplementeerd om de performantie ervan te vergelijken met de andere set-join algoritmes. Alle andere set-join algoritmes zouden, minstens voor bepaalde parameterkeuzes, sneller moeten zijn dan het Nested Loops algoritme.

Pseudocode

Algorithm 1 Nested Loops

Input: Relatie R, relatie S
Output: Alle paren ($t_R.key$, $t_S.key$) zodat $t_R.set \subseteq t_S.set$

```
1: result = {}
2: for t_R ∈ R do
3:    for t_S ∈ S do
4:       if t_R.set ⊆ t_S.set then
5:          result = result ∪ (t_R.key, t_S.key)
6:       end if
7:    end for
8: end for
9: return result
```

Voorbeeld

Als voorbeeld beschouwen we relatie R uit tabel 21 en relatie S uit tabel 22.

We gaan elke tuple t_R van R en elke tuple t_S van S af en controleren of $t_R.set \subseteq t_S.set$. Zo zien we dat $\{28, 67, 70\} \subseteq \{28, 67, 70, 90\}$, $\{13, 46\} \subseteq \{13, 46, 96\}$, $\{18, 70\} \subseteq \{18, 67, 70\}$ en $\{5, 11, 27\} \subseteq \{5, 9, 11, 27\}$. Het resultaat bestaat uit

de bijhorende paren van ($t_R.key$, $t_S.key$): (x2, y2), (x4, y4), (x6, y1) en (x7, y3).

Tijdscomplexiteit

Elke combinatie van een tuple van R met een tuple van S wordt bekeken. Voor elke van deze combinaties wordt een set containment (of inclusie) gecontroleerd, waarbij nagegaan wordt of een verzameling met gemiddeld B_R elementen, een deelverzameling is van een verzameling met gemiddeld B_S elementen, waarbij beide verzamelingen gekozen werden uit een verzameling met A elementen.

We kunnen deze inclusiecontrole opvatten als een inclusiecontrole voor elk van de elementen van $t_R.set$, waarbij de controle doorgaat tot wanneer een element gevonden is dat niet in $t_S.set$ zit, of tot wanneer alle elementen van $t_R.set$ behandeld zijn. De duur van zo een inclusiecontrole hangt dan ook af van de keuze van de de parameters A, B_R en B_S. Als bijvoorbeeld B_R en B_S veel kleiner zijn dan A, dan wordt de kans dat een element van de set van een R-tuple ook een element is van de set van een S-tuple klein, en dan zal bijna elke inclusiecontrole neerkomen op de inclusiecontrole van een enkel element. In het andere uiterste, wanneer $B_R \lessapprox B_S \lessapprox A$, zal de inclusiecontrole veel langer duren. Hoe dan ook is het kwadratische karakter in dit algoritme erg duidelijk.

Bewijs van correctheid

Voor het NL-algoritme is dit triviaal: aangezien voor elke combinatie van een R-tuple en een S-tuple expliciet wordt gecontroleerd of $t_R.set \subseteq t_S.set$, wordt elke set inclusion gedetecteerd en zijn er geen valse detecties.

3.2 Signature Nested Loops (SNL)

In deze paragraaf bespreken we de werking van signatures en hoe dergelijke signatures kunnen gebruikt worden om op een efficiënte manier de set containment join te berekenen.

3.2.1 Signature

Definitie

Een **signature** is een string van bits van een vaste lengte en wordt gebruikt om sets exact of ongeveer voor te stellen. Stel dat D het willekeurig geordend domein is van de elementen die kunnen voorkomen in een set en stel dat $|D|$ de kardinaliteit is, dus het aantal elementen in D. Dan kan een set x exact voorgesteld worden door een signature $sig(x)$ van lengte $|D|$. Een "geordend" domein betekent dat er een bepaalde volgorde gerespecteerd wordt. Een voorbeeld van een domein is: alle getallen van 0 tot en met 99. Dit domein heeft kardinaliteit 100. Een ander voorbeeld van een domein is alle ASCII-strings met een lengte van 0 tot en met 10.

De definitie van een **exacte signature** $sig_{ex}(x)$ van een verzameling x is als volgt: voor elke $i, 1 \leq i \leq |D|$, geldt dat de i-de bit van $sig(x)$ 1 is als en slechts

als het i-de element van D in x zit. Anders is die i-de bit 0.

Een exacte signature is zeer duur om op te slaan, vooral als de set sparse (verspreid) is. Stel bijvoorbeeld dat het domein D van een set x bestaat uit de eerste 1000 positieve natuurlijke getallen, dus 0, ... 999. Dus $D = 0, 1, \ldots 999$ en $|D| = 1000$. Dan is de lengte van de signatures van x gelijk aan 1000. Omdat een exacte signature zo duur is om op te slaan (in dit geval 1000 bits, dus 125 bytes per signature), worden er benaderingen (approximations) gebruikt.

We definiëren een **signature (approximation)** $sig_{app}(x)$ als volgt: gegeven **een vaste signature lengte** b, kent een mapping functie elk element van D toe aan een bit positie in $[0, b) = \{0, \ldots, b-1\}$. Eigenlijk is er ook een mapping functie bij een exacte signature. Zij $i \in D$ een getal. Dan is er bij een exacte signature een mapping van i naar de i-de positie in de signature, dus $i \rightarrow i$-de positie. Bij een approximation mapt men i naar een bit positie in $[0, b($. Een praktisch voorbeeld van zo'n mapping functie is de modulo-operator. In dit geval gebeurt er dus een mapping $i \rightarrow i \ \% \ b$-de positie.

We illusteren aan de hand van een voorbeeld: stel dat het domein D uit de eerste 100 getallen bestaat en stel dat de signature lengte $b = 10$. Stel dat de set $x = \{5, 7, 56, 98\}$. Dan zou de **exacte signature** $sig_{ex}(x)$ uit een string van lengte 100 bestaan waarbij de 5de, 7de, 56ste en 98ste positie een 1 is. De rest van deze string bestaat uit nullen. Voor de **approximation** van de signature sig_{app} gebruiken we de mapping-functie modulo 10. We passen deze mapping-functie toe op elk element van x en we krijgen: $5 \ \% \ 10 = 5$, $7 \ \% \ 10 = 7$, $56 \ \% \ 10 = 6$ en $98 \ \% \ 10 = 8$. We gaan dus de 5de, 7de, 6de en de 8ste positie van de string op 1 zetten en de rest op 0. We krijgen dus $sig_{app}(x) = 0000011110$.

Eigenschappen

Stel dat we twee relaties R en S hebben. Dan noemen we t_R een tuple van R en t_S een tuple van S. Beide tuples hebben een set-valued attribuut. We noemen $t_R.r$ het set-valued attribuut van R en $t_S.s$ het set-valued attribuut van S.
Als we de signature van twee sets vergelijken, kunnen we meer te weten komen over de sets. We weten dat $t_R.r \subseteq t_S.s \implies sig(t_R.r) \subseteq sig(t_S.s)$. Volgens de logica geldt: $\neg(sig(t_R.r) \subseteq sig(t_S.s)) \implies \neg(t_R.r \subseteq t_S.s)$. Dus: $sig(t_R.r) \not\subseteq sig(t_S.s) \implies t_R.r \not\subseteq t_S.s$.

We kunnen nagaan of $t_R.r \subseteq t_S.s$. Dan checken we eerst of $sig(t_R.r) \subseteq sig(t_S.s)$, want als dit niet het geval is, dan zegt de eigenschap hierboven dat $t_R.r$ geen deelverzameling is van $t_S.s$. Deze check is zeer eenvoudig op voorwaarde dat de signature length niet gigantisch groot is en gebeurt als volgt: we gaan na op welke posities $sig(t_R.r)$ 1 is. Als $sig(t_S.s)$ op diezelfde posities ook 1 is, dan geldt $sig(t_R.r) \subseteq sig(t_S.s)$. Merk op dat $sig(t_S.s)$ ook op andere posities een 1 kan zijn.

Omdat signatures uit een reeks van bits bestaan, is het heel efficiënt om twee signatures met elkaar te vergelijken. Stel dat we twee signatures hebben $sig(a)$ en $sig(b)$. Dan kunnen we nagaan of $sig(a) \subseteq sig(b)$ door te kijken naar het

resultaat van $sig(a)\&\neg sig(b)$. Als dit resultaat 0 is, dan geldt $sig(a) \subseteq sig(b)$. Dus we concluderen: $sig(a)\&\neg sig(b) == 0 \implies sig(a) \subseteq sig(b)$.

Voorbeelden

Als voorbeeld beschouwen we twee sets $x = \{1, 5, 29, 44\}$ en $y = \{1, 5, 29, 34, 56\}$, signature lengte 10 en mapping functie modulo 10. Dan is $sig(x) = 0100110001$ en $sig(y) = 0100111001$. We zien dat $sig(x) \subseteq sig(y)$ want $sig(y)$ is 1 op alle posities waar $sig(x)$ 1 is, namelijk posities 1, 4, 5 en 9. Omgekeerd geldt dit niet: $sig(y) \not\subseteq sig(x)$ want $sig(x)$ is niet 1 op positie 6 en $sig(y)$ is wel 1 op positie 6. Hieruit leiden we af dat $y \not\subseteq x$.

We tonen dit aan met een nieuw voorbeeld: stel we hebben twee signatures $sig(a) = 100101011$ en $sig(b) = 000101000$ van lengte 9. Als we de methode uit vorige alinea gebruiken, zien we duidelijk dat $sig(a) \not\subseteq sig(b)$ en $sig(b) \subseteq sig(a)$. We passen nu de methode toe met de bitoperaties en we krijgen:

- $sig(a)\&\neg sig(b) = 100101011$ & $111010111 = 100000011 \neq 0$, dus $sig(a) \not\subseteq sig(b)$
- $sig(b)\&\neg sig(a) = 000101000$ & $011010100 = 000000000 = 0$, dus $sig(a) \subseteq sig(b)$

De vergelijking van twee signatures gebeurt heel snel omdat er alleen maar bit operaties gebeuren. Vandaar dat dit de beste methode is om na te gaan of een signature een subset is van een andere signature.

3.2.2 SNL

Het Signature Nested Loops algoritme, dat we vanaf nu **SNL** noemen, probeert het vergelijken van de sets van de waarden van de set-valued attributen zo snel mogelijk te laten gebeuren door de signature van die sets te gebruiken en ze met elkaar te vergelijken. Als deze signatures niet aan bepaalde voorwaarden voldoen, dan weten we dat de ene set geen deelverzameling kan zijn van de andere set en moeten we de echte sets dus niet vergelijken. Als de signatures wel voldoen aan de voorwaarden, dan spreken we van een **drop** en moeten we de echte sets wel gaan vergelijken. Als blijkt dat de ene set geen deelverzameling is van de andere set, dan spreken we van een **false drop**.

Het originele SNL-algoritme zoals het beschreven is in [3] bestaat uit drie fases:

1. Signature construction phase
2. Probing phase
3. Verification phase

Dit algoritme maakt gebruik van de harde schijf om in de signature construction phase informatie over relatie R weg te schrijven naar een file. Aangezien de focus van de masterproef ligt op main-memory set containment join algoritmes, wordt er een main-memory variant van dit SNL-algoritme besproken. In deze variant is er dus geen signature construction phase, want er hoeft geen informatie weggeschreven te worden naar een file. We gaan dus direct over naar de

probing phase.

Een ander belangrijk verschil is dat het main-memory SNL-algoritme gebruik maakt van het key-attribuut om een tuple te identificeren, in plaats van een row-id. De reden waarom werd reeds besproken in hoofdstuk 3.

Probing phase

In deze fase gaan we elke tuple t_R af van relatie R en elke tuple t_S van relatie S. Voor zowel t_R als t_S bepalen we de waarde van het key-attribuut $t_R.key$ ($t_S.key$), de set $t_R.set$ ($t_S.set$), de signature $sig(t_R.set)$ ($sig(t_S.set)$) en de kardinaliteit $|t_R.set|$ ($|t_S.set|$). Vervolgens controleren we of $sig(t_R.set) \subseteq sig(t_S.set)$ en $|t_R.set| \leq |t_S.set|$. Als dat zo is, dan hebben we een **drop**.

Verification phase

Elke keer dat er een drop wordt vastgesteld voor een paar ($t_R.key$, $t_S.key$), gaan we na of $t_R.set \subseteq t_S.set$.

Pseudocode

Algorithm 2 Signature Nested Loops

Input: Relatie R, relatie S, signature lengte b
Output: Alle paren ($t_R.key$, $t_S.key$) zodat $t_R.set \subseteq t_S.set$

```
1:  result = {}
2:  signaturesS = {}
3:  sigR = 0
4:  for t_S ∈ S do
5:      signaturesS = signaturesS ∪ (t_S.key, sig(t_S.set))
6:  end for
7:  for t_R ∈ R do
8:      sigR = sig(t_R.set)
9:      for t_S ∈ S do
10:         if sigR ⊆ signaturesS.get(t_S.key) and |t_R.set| ≤ |t_S.set| then
11:             if t_R.set ⊆ t_S.set then
12:                 result = result ∪ (t_R.key, t_S.key)
13:             end if
14:         end if
15:     end for
16: end for
17: return result
```

Voorbeeld

Als voorbeeld beschouwen we relatie R uit tabel 21 en relatie S uit tabel 22. Hier worden de relaties voorgesteld in een formaat waarbij elke tuple uit een key en uit een set van waarden bestaat die bij die key horen (in plaats van het standaardformaat zoals in tabel 18). We gebruiken dit formaat omdat de tabel

anders zeer lang en onoverzichtelijk zou worden.

We passen SNL toe op relatie R en relatie S. Stel dat de signature lengte 10 is. De signature van elke set van R vinden we terug in tabel 23. De signature van elke set van S vinden we terug in tabel 24. Probing fase: we gaan elke tuple t_R van R af en elke tuple t_S van S en we gaan na of $sig(t_R) \subseteq sig(t_S)$ en of $|t_R.set| \leq |t_S.set|$. Voor volgende paren van ($t_R.key$, $t_S.key$) is dit het geval: (x2, y1), (x2, y2), (x3, y7), (x4, y4), (x6, y1), (x6, y2), (x7, y3). Dit is dus onze *candidates*-verzameling. Verification fase: tenslotte gaan we elk kandidaat-paar $(t_R.key, t_S.key)$ af en gaan we na of $t_R.set \subseteq t_S.set$. De kandidaat-paren (x2, y1), (x3, y7) en (x6, y2) vallen af. We houden dus de paren (x2, y2), (x4, y4), (x6, y1) en (x7, y3) over en dit is het resultaat.

Tabel 21: De relatie R(key, values).

key	values
x1	$\{38, 67, 83, 90, 97\}$
x2	$\{28, 67, 70\}$
x3	$\{5, 10, 15, 20, 25, 49\}$
x4	$\{13, 46\}$
x5	$\{8, 88, 34, 97\}$
x6	$\{18, 70\}$
x7	$\{5, 11, 27\}$

Tabel 22: De relatie S(key, values).

key	values
y1	$\{18, 67, 70\}$
y2	$\{28, 67, 70, 90\}$
y3	$\{5, 9, 11, 27\}$
y4	$\{13, 46, 96\}$
y5	$\{9, 99, 29\}$
y6	$\{8, 88, 34\}$
y7	$\{5, 10, 15, 20, 25, 39\}$

Tabel 23: Relatie R(key, values) met signatures van lengte 10.

key	values	$sig(values)$
x1	$\{38, 67, 83, 90, 97\}$	1001000110
x2	$\{28, 67, 70\}$	1000000110
x3	$\{5, 10, 15, 20, 25, 49\}$	1000010001
x4	$\{13, 46\}$	0001001000
x5	$\{8, 88, 34, 97\}$	0000100110
x6	$\{18, 70\}$	1000000010
x7	$\{5, 11, 27\}$	0100010100

Tabel 24: De relatie S(key, values) met signatures van lengte 10.

key	values	$sig(values)$
y1	$\{18, 67, 70\}$	1000000110
y2	$\{28, 67, 70, 90\}$	1000000110
y3	$\{5, 9, 11, 27\}$	0100010101
y4	$\{13, 46, 96\}$	0001001000
y5	$\{9, 99, 29\}$	0000000001
y6	$\{8, 88, 34\}$	0000100010
y7	$\{5, 10, 15, 20, 25, 39\}$	1000010001

Tijdscomplexiteit

Het opstellen van de signatures gaat in een tijd die lineair is in n_R, resp. n_S. In de probing phase moet nog steeds voor elk van de $n_R \cdot n_S$ combinaties een controle worden uitgevoerd, maar zoals vermeld gaat deze sneller dan de inclusiecontrole uit het NL-algoritme. In een fractie ϵ van de combinaties verkrijgen we een drop, waarna we alsnog de inclusiecontrole moeten uitvoeren. Voor deze combinaties hebben we dezelfde duur als in het NL-algoritme, maar het cruciale punt is dat dit niet meer voor elke combinatie moet gedaan worden, maar slechts voor een fractie ϵ van de combinaties.

Hier duikt een interessant evenwicht op: de keuze van de signature length heeft een invloed op de duur van de twee fases. Bij een grote signature length duurt elke stap van de probing phase langer (er zijn meer bitcontroles nodig), maar het aandeel false drops zal kleiner zijn. Bij een kleinere signature length gelden de omgekeerde effecten en het zal dus nodig zijn om een optimale signature length te vinden.

Het is interessant na te gaan hoe dit algoritme een worst-case scenario afhandelt. Hiermee bedoelen we de situatie zoals in tabel 25, waarin elk koppel van een R-tuple en een S-tuple set containment oplevert, wat makkelijk te simuleren is door $A = B_R = B_S = 1$ te kiezen. Elke signature zal dan dezelfde zijn en ook elke kardinaliteit is gelijk aan 1. Elk koppel van R-tuple en S-tuple komt dus in de kandidatenlijst terecht, waarna de set containment voor elk koppel moet worden gecontroleerd. De facto wordt dus achteraf het NL-algoritme op de volledige dataset uitgevoerd, zodat de run time minstens gelijk zal zijn aan de (kwadratische) run time van het NL-algoritme.

R

Key	**Set**
1	$\{t\}$
2	$\{t\}$
3	$\{t\}$
...	...
n_R	$\{t\}$

S

Key	**Set**
1	$\{t\}$
2	$\{t\}$
3	$\{t\}$
...	...
n_S	$\{t\}$

Tabel 25: Relaties R en S voor een worst case scenario

Bewijs van correctheid

Wanneer gewerkt wordt met niet-exacte signatures, dan impliceert een set inclusion een signature inclusion, maar de omgekeerde implicatie is niet geldig. Dit betekent dat er bij de drops ook false drops zullen zijn. Door in de verification phase deze drops te overlopen en elke combinatie expliciet te controleren, zijn we zeker dat dit algoritme correct is.

3.3 Partitioned Set Join (PSJ)

Het Partitioned Set Join algoritme (**PSJ**) deelt het probleem op in kleinere subproblemen door een opdelingsfunctie te gebruiken. Het doel hiervan is de tijd van de join-fase te verkorten. Er wordt gestart met het opdelen van relatie R in k klassen $R_1, R_2, \ldots, R_k$. Een heel eenvoudige manier om dit te doen is gebruik te maken van de modulofunctie, al zijn er ook andere methoden zoals de universal hashing van Carter en Wegman. Daarna wordt een opdeling gemaakt van relatie S in evenveel klassen $S_1, S_2, \ldots, S_k$, op basis van dezelfde opdelingsfunctie. Dit is echter geen partitie in de wiskundige zin van het woord, aangezien het kan gebeuren dat een tuple van S in verschillende klassen zit. Dit zal duidelijker worden in de verdere beschrijving van het algoritme en in het voorbeeld.

PSJ gebruikt een "two-level partitioning scheme". Het bestaat uit drie fases:

1. Opdelingsfase: uit elk tuple van R wordt een willekeurig element gekozen. Op basis van dit element wordt het tuple naar exact één klasse gestuurd, deze wordt bepaald door een functie h, bijvoorbeeld de modulo-functie %. Daarna wordt elk tuple van S toegewezen aan een aantal klassen door dezelfde h toe te passen op elk element van dit tuple.

2. Join-fase: nu worden de klassen van R en van S met elkaar vergeleken. Wanneer op basis van deze vergelijking containment mogelijk is, dan spreken we van een drop. Aangezien er niet met exacte signatures wordt gewerkt, zijn false drops hier mogelijk.

3. Verficatiefase: de paren van tuples die als drop werden gecatalogeerd in de join-fase, worden gecontroleerd door expliciet de set attributen van beide tuples te controleren op containment. Op deze manier worden de false drops geëlimineerd.

Net zoals bij SNL, beschouwen we hier ook een main-memory variant van het PSJ-algoritme. Deze variant maakt gebruik van het key-attribuut om een tuple te identificeren, in plaats van een row-id. De reden hiervoor werd reeds besproken in hoofdstuk 3.

Opdelingsfase

Eerst wordt beslist in hoeveel klassen R zal worden opgedeeld, dit aantal noemen we k. Dan wordt de opdelingsfunctie h vastgelegd, een erg eenvoudige keuze is dan ook de modulo k-functie waarvoor $h(e_R) = e_R \% k$. We starten de opdeling bij relatie R. Voor elke tuple $t_R \in R$, worden volgende stappen uitgevoerd:

1. Er wordt een triple (c_i, s_i, key_i) berekend. c_i is de kardinaliteit $|t_R.set|$, s_i is de signature van $t_R.set$ en key_i is de key-waarde van de tuple, genaamd $t_R.key$.

2. Er wordt een willekeurig element e_R geselecteerd uit $t_R.set$.

3. Het triple wordt gestuurd naar de klasse die bepaald wordt door $h(e_R)$.

We merken op dat het triple van elke tuple van R naar exact één klasse wordt gestuurd. Dit zal niet het geval zijn voor relatie S. Voor elke tuple $t_S \in S$ worden volgende stappen uitgevoerd:

1. Er wordt een triple (c_i, s_i, key_i) berekend. c_i is de kardinaliteit $|t_S.set|$, s_i is de signature van $t_S.set$ en key_i is de key-waarde van de tuple, genaamd $t_S.key$.

2. *Voor elk element* $e_S \in t_S.set$ wordt het triple gestuurd naar de klasse bepaald door $h(e_S)$.

Merk op: als $t_R.set \subseteq t_S.set$, dan zal de klasse bepaald door $h(e_R)$ de triples bevatten die overeenkomen met t_R en t_S. Dit toont aan dat het algoritme de set containment correct berekent: er zijn geen set containments die niet herkend worden.

Zoals eerder vermeld zijn er andere mogelijkheden om de opdeling uit te voeren. De modulofunctie is erg eenvoudig, maar ze is deterministisch. Dit kan een nadeel zijn, vooral wanneer de elementen niet met eenzelfde waarschijnlijkheid gekozen worden. Wanneer bijvoorbeeld elementen e met $e\%k = 0$ veel vaker voorkomen dan de andere, dan zal de bijhorende klasse ook veel meer triples bevatten, wat verderop tot meer false drops zal leiden. Dit kan opgelost worden door te werken met universal hashing. Dit is een methode waarbij een hash-functie willekeurig gekozen wordt uit een familie, zodat de kans dat verschillende elementen in een zelfde klasse terechtkomen, niet groter is dan $1/k$, wat de kans zou zijn wanneer we met volledig willekeurige toewijzingen werken. Deze methode werd uitgewerkt door Carter en Wegman [12]. Aangezien in de experimenten die in dit werk worden uitgevoerd, de elementen uit uniforme verdelingen worden gekozen, heeft deze methode hier weinig toegevoegde waarde.

Join-fase

Tijdens deze fase wordt elke klasse van R gejoind met een klasse van S. Merk op dat elk triple enkel de signature en de kardinaliteit van de set bevat, dus niet de set zelf: het join-algoritme werkt rechtstreeks op signatures en de kardinaliteit van de set.

Het join-algoritme bestaat uit twee stappen: de *build*-stap en de *probe*-stap.

In de *build*-stap wordt een array A geconstrueerd die even groot is als de signature lengte. Elk element van de array A is een lijst van triples. We starten met een lege lijst A en we scannen achtereenvolgens elke klasse van R en elk triple dat in die klasse zit. We kiezen een willekeurige bitpositie m die op 1 staat in

de signature en we voegen het triple toe aan de lijst $A[m]$. Op het einde van deze stap zit elk triple uit R in exact één lijst $A[m]$.

In de daaropvolgende *probe*-stap worden de klassen van S achtereenvolgens gescand. Voor elk triple (c_j, s_j, key_j) van S wordt de lijst van signatures in $A[n]$ overlopen telkens wanneer bit n op 1 staat in s_j. Voor elk triple (c_i, s_i, key_i) in $A[n]$ controleren we of $s_i \subseteq s_j$ en $c_i \leq c_j$. Als dat zo is, dan hebben we een **drop**.

Verificatiefase

In deze fase gaan we voor elke drop voor een paar ($t_R.key$, $t_S.key$) na of $t_R.set$ in $t_S.set$ zit. Als dit niet het geval is, dan hebben we een false drop.

Pseudocode

Algorithm 3 Partitioned Set Join

Input: Relatie R, relatie S, signature lengte b, aantal klassen k
Output: Alle paren ($t_R.key$, $t_S.key$) zodat $t_R.set \subseteq t_S.set$

```
1: candidates = {}
2: result = {}
3: rPartitions = [k]
4: sPartitions = [k]
5: A = []
6: for t_R ∈ R do
7:    tR(tR.card, tR.sig, tR.key) = new Triplet(|t_R.set|, sig(t_R.set), t_R.key)
8:    e_Rand = t_R.set[rand % tR.c]
9:    rPartitions[e_Rand % k].add(tR)
10: end for
11: for t_S ∈ S do
12:    tS(tS.card, tS.sig, tS.key) = new Triplet(|t_S.set|, sig(t_S.set), t_S.key)
13:    for e ∈ t_S.set do
14:       sPartitions[e % k].add(tS)
15:    end for
16: end for
17: for lR ∈ rPartitions do
18:    for tR ∈ lR do
19:       m = getRandomBitIndex(tR.sig)
20:       A[m].add(tR)
21:    end for
22: end for
23: for lS ∈ sPartitions do
24:    for tS ∈ lS do
25:       indices = getIndicesSetBits(tS.sig)
26:       for i ∈ indices do
27:          for tR ∈ A[i] do
28:             if tR.card ≤ tS.card and tR.sig ⊆ tS.sig then
29:                setR = R.get(tR.key)
30:                setS = S.get(tS.key)
31:                if setR ⊆ setS then
32:                   result = result ∪ (tR.key, tS.key)
33:                end if
34:             end if
35:          end for
36:       end for
37:    end for
38: end for
39: return result
```

Voorbeeld

Als voorbeeld beschouwen we opnieuw relatie R uit tabel 21 en relatie S uit tabel 22. We passen PSJ toe op R en S. Stel dat signature lengte $b = 8$ en

aantal klassen $k = 5$. Eerst bepalen we de signatures van elke set van R (tabel 26) en de signatures van elke set van S (tabel 27).

Tabel 26: Relatie R(key, values) met signatures van lengte 8.

key	values	$sig(values)$
x1	$\{38, 67, 83, 90, 97\}$	01110010
x2	$\{28, 67, 70\}$	00011010
x3	$\{5, 10, 15, 20, 25, 49\}$	01101101
x4	$\{13, 46\}$	00000110
x5	$\{8, 88, 34, 97\}$	11100000
x6	$\{18, 70\}$	00100010
x7	$\{5, 11, 27\}$	01010100

Tabel 27: De relatie S(key, values) met signatures van lengte 8.

key	values	$sig(values)$
y1	$\{18, 67, 70\}$	00110010
y2	$\{28, 67, 70, 90\}$	00111010
y3	$\{5, 9, 11, 27\}$	01010100
y4	$\{13, 46, 96\}$	10000110
y5	$\{9, 99, 29\}$	01010100
y6	$\{8, 88, 34\}$	10100000
y7	$\{5, 10, 15, 20, 25, 39\}$	01101101

We beginnen met de opdelingsfase. Voor elk tuple $t_R \in R$ bepalen we eigenlijk het triplet $(|t_R.set|, sig(t_R.set), t_R.key)$. Dan kiezen we een willekeurig element $e_R \in t_R.set$ en dan voegen we dit triplet toe aan de klasse bepaald door e_R % k. Tabel 28 toont welke e_R gekozen wordt voor elke $t_R \in R$, hier werd $k = 5$ gekozen.

Tabel 28: Relatie $R(key, values)$ met signatures, willekeurig gekozen e_R en klasse bepaald door e_R % 5.

key	values	$sig(values)$	e_R	klasse e_R % 5
x1	$\{38, 67, 83, 90, 97\}$	01110010	83	R_3
x2	$\{28, 67, 70\}$	00011010	70	R_0
x3	$\{5, 10, 15, 20, 25, 49\}$	01101101	5	R_0
x4	$\{13, 46\}$	00000110	46	R_1
x5	$\{8, 88, 34, 97\}$	11100000	34	R_4
x6	$\{18, 70\}$	00100010	18	R_3
x7	$\{5, 11, 27\}$	01010100	27	R_2

De klassen van R bevatten nu volgende triples:

$$R_0 = [(3, 00011010, x_2), (6, 01101101, x_3)]$$

$$R_1 = [(2, 00000110, x_4)]$$

$$R_2 = [(3, 01010100, x_7)]$$

$$R_3 = [(5, 01110010, x_1), (2, 00100010, x_6)]$$

$$R_4 = [(4, 11100000, x_5)]$$

Voor elke tuple $t_S \in S$ bepalen we het triple $(|tS.set|, sig(t_S.set), t_S.key)$. Dan gaan we voor elk element $e_S \in t_S.set$ het triple toevoegen aan de klasse bepaald door $e_S \ \% \ k$. Tabel 29 toont de klassen waaraan elk triple wordt toegevoegd.

Tabel 29: De relatie $S(key, values)$ met signatures en bijhorende klassen bepaald door $e_S \ \% \ 5$.

key	values	$sig(values)$	klassen
y1	$\{18, 67, 70\}$	00110010	S_3, S_2, S_0
y2	$\{28, 67, 70, 90\}$	00111010	S_3, S_2, S_0, S_0
y3	$\{5, 9, 11, 27\}$	01010100	S_0, S_4, S_1, S_2
y4	$\{13, 46, 96\}$	10000110	S_3, S_1, S_1
y5	$\{9, 99, 29\}$	01010100	S_4, S_4, S_4
y6	$\{8, 88, 34\}$	10100000	S_3, S_3, S_4
y7	$\{5, 10, 15, 20, 25, 39\}$	01101101	$S_0, S_0, S_0, S_0, S_0, S_4$

Omdat elk triple in elke klasse slechts één keer geraadpleegd wordt, laten we de dubbels weg. De klassen van S bevatten nu volgende triples:

$$S_0 = [(3, 00110010, y_1), (4, 00111010, y_2), (4, 01010100, y_3), (6, 01101101, y_7)]$$

$$S_1 = [(4, 01010100, y_3), (3, 10000110, y_4]$$

$$S_2 = [(3, 00110010, y_1), (4, 00111010, y_2), (4, 01010100, y_3)]$$

$$S_3 = [(3, 00110010, y_1), (4, 00111010, y_2), (3, 10000110, y_4), (3, 10100000, y_6)]$$

$$S_4 = [(4, 01010100, y_3), (3, 01010100, y_5), (3, 10100000, y_6), (6, 01101101, y_7)]$$

Na de opdelingsfase volgt de join-fase. In de build-stap vullen we array A aan met triples uit de klassen van R. Tabel 30 toont hoe de triples uit de klassen van R worden toegevoegd aan array A. Per triple gebruiken we een willekeurige bitpositie uit de signature die op 1 staat, genaamd m. We voegen het triple toe aan $A[m]$.

Tabel 30: Triples uit klassen van R toevoegen aan A.

klasse van R	triple	m	$A[m]$
R_0	$(3, 00011010, x_2)$	4	$A[4]$
R_0	$(6, 01101101, x_3)$	7	$A[7]$
R_1	$(2, 00000110, x_4)$	5	$A[5]$
R_2	$(3, 01010100, x_7)$	5	$A[5]$
R_3	$(5, 01110010, x_1)$	1	$A[1]$
R_3	$(2, 00100010, x_6)$	6	$A[6]$
R_4	$(4, 11100000, x_5)$	2	$A[2]$

Array A ziet er nu zo uit:

$$A[0] = []$$

$$A[1] = [(5, 01110010, x_1)]$$

$$A[2] = [(4, 11100000, x_5)]$$

$$A[3] = []$$

$A[4] = [(3, 00011010, x_2)]$

$A[5] = [(2, 00000110, x_4), (3, 01010100, x_7)]$

$A[6] = [(2, 00100010, x_6)]$

$A[7] = [(6, 01101101, x_3)]$

In de probe-stap gaan we alle klassen van S af en gaan we op zoek naar de juiste triples uit A. We vergelijken telkens de signatures $tS.sig$ van de triples tS van S met de signatures van de triples van $A[n].sig$ en controleren of $A[n].sig \subseteq tS.sig$ en $|A[n].set| \leq |tS.set|$. Als dat zo is, dan hebben we een **drop** en voegen we het paar $(A[n].key, tS.key)$ toe aan de verzameling van kandidaten.

Bijvoorbeeld voor triple $tS = (3, 00110010, y_1)$ uit S_0 checken we de triples uit $A[2], A[3]$ en $A[6]$. Vervolgens vergelijken we $tS.sig$ met alle signatures $A[i].sig$ uit A en we vergelijken $|tS.set|$ met alle kardinaliteiten $|A[i].set|$ uit A met $i = 2, i = 3$ en $i = 6$. Als $A[i].sig \subseteq tS.sig$ en als $|A[i].set| \leq |tS.set|$ dan voegen we het paar $(A[i].key, tS.key)$ toe aan de verzameling van kandidaten.

De kandidatenverzameling ziet er dan zo uit: (x_6, y_1), (x_6, y_2), (x_2, y_2), (x_7, y_3), (x_3, y_7), (x_7, y_5), (x_4, y_4).

Verificatiefase: tenslotte gaan we elk kandidaat-paar ($A[i].key$, $tS.key$) af en gaan we na of $A[i].set \subseteq tS.set$. De kandidaat-paren ($x_6$, y_2), (x_3, y_7) en (x_7, y_5) vallen af. We houden dus de paren (x_6, y_1), (x_2, y_2), (x_7, y_3) en (x_4, y_4) over en dit is het resultaat.

Tijdscomplexiteit

De enige stappen die een kwadratische tijd vergen in dit algoritme, zijn de probe-stap en de verificatie fase.

Tijdens de probe-stap moet telkens een controle gedaan worden op inclusie voor signatures (wat zeer snel kan, net als bij SNL) en op grootte van reeds opgeslagen kardinaliteiten (wat ook een zeer snelle bewerking is). De vraag is dan hoe vaak deze controle moet worden uitgevoerd: voor elk triple uit een klasse van S moet een aantal arrays bekeken worden, evenveel als er enen staan in de signature van het triplet. Elk van die arrays bevat dan ook nog eens meerdere triplets van R. Het zal dan ook duidelijk zijn dat dit een zeer tijdsintensieve stap kan zijn wanneer een groot aantal bits op 1 staat.

Het aantal drops is het aantal koppels dat expliciet moet vergeleken worden. Ook hier speelt de signature length een rol: een te kleine signature length zal voor een relatief groot aantal false drops zorgen, wat deze laatste stap tijdrovend maakt.

Ook hier kunnen we het worst-case scenario bekijken. Wanneer $A = B_R = B_S = 1$ zal er slechts één klasse van R worden opgevuld en ook enkel dezelfde klasse van S. Er wordt dan ook maar één element van A opgevuld, zodat weer alle koppels van R- en S-triples een drop opleveren. De verificatie achteraf komt

dus, net als bij SNL, neer op het uitvoeren van het NL-algoritme op de volledige dataset. Ook hier krijgen we dus een run time die minstens zo lang is als de kwadratische runtime van het NL-algoritme.

Bewijs van correctheid

Beschouw tuples $t_R \in R$ en $t_S \in S$ waarvoor $t_R.set \subseteq t_S.set$. Van beide wordt de signature opgesteld. Op basis van een willekeurig gekozen element van $t_R.set$ wordt het t_R-triple toegevoegd aan een bepaalde klasse. Gezien de inclusie wordt ook het t_S-triple aan deze klasse toegevoegd. Het t_R-triple wordt dan aan een bepaalde array toegevoegd, op basis van een willekeurig gekozen 1 in de signature (wanneer $t_R.set$ leeg is, is dit niet mogelijk, maar aangezien de lege verzameling een deelverzameling is van elke verzameling valt dit eenvoudig te verhelpen zonder veel extra berekentijd). Tot slot worden de t_S-triples vergeleken met deze arrays, waarbij gekeken wordt naar de kardinaliteit en naar de inclusie van signatures. Aangezien $t_R.set \subseteq t_S.set$ levert dit een positief resultaat op en deze combinatie zal dan ook een drop opleveren.

Door in de verificatie fase voor elke drop expliciet de inclusie te testen, worden de false drops verwijderd en blijven enkel de combinaties over waarvoor $t_R.set \subseteq t_S.set$.

3.4 Indexed Nested Loops (INL)

Het Indexed Nested Loops algoritme (**INL**) berekent een set-containment join door gebruik te maken van een **inverted index**.

De inverted index die gebruikt wordt in het INL-algoritme is gelijkaardig aan de inverted index die gebruikt wordt in document retrieval. In document retrieval bestaat een inverted index uit een verzameling van woord-pointer paren. Het woord is een key voor de inverted index. De pointer wijst naar de documenten waarin het woord voorkomt. Er is dus eigenlijk een (key, value)-mapping waarbij het woord de key is en de value bestaat uit (de pointer naar) de documenten die de key bevat.

Zo'n inverted index kan ook gebruikt worden voor set-valued attributen: we maken voor elk set element e in het domein D een **inverted list**. Deze inverted list bevat alle keys van de tuples wiens set e bevat. Al deze inverted lists vormen tesamen de **inverted file**. De inverted file geeft dus voor elk element aan in welke verzamelingen dit element voorkomt.

Een inverted file is handig voor het bepalen van een set containment-join tussen twee relaties R en S op hun set-valued attributen $t_R.set$ en $t_S.set$. We gaan als volgt te werk:

1. We construeren een inverted file S_{IF} voor relatie S. Voor elk element $e_S \in t_S.set$ van elke tuple t_S van S, houden we de keys bij van alle tuples t zodat $e_S \in t.set$ waarbij $t.set$ de set is die bij tuple t hoort. De geconstrueerde inverted file geeft dus voor elk element e_S van relatie S een (key, $value$)-mapping van key e_S op de verzameling van keys van de

tuples t wiens set $t.set$ het element e_S bevat. Deze verzameling van keys noemen we de *value*.

2. Vervolgens gaan we elke tuple t_R van relatie R af. Dan kijken we voor elk element $e_R \in t_R.set$ of deze als *key* voorkomt in S_{IF}.

   - Als dit niet zo is, dan betekent dit dat element e_R nergens voorkomt in relatie S. Bijgevolg kan $t_R.set$ geen deelverzameling zijn van een set $t_S.set$ en kunnen we tuple t_R dus overslaan.
   - Als e_R wel een key is in S_{IF}, dan vragen we de bijhorende set op en houden we een kopie ervan bij, genaamd cpy_{set}. Voor elk ander element $x_R \in t_R.set$ bepalen we de bijhorende set set_x uit S_{IF}. We nemen nu telkens de intersectie van cpy_{set} met alle set_x en we kennen het resultaat van de intersecties toe aan cpy_{set}. Nadat elk element in $t_R.set$ is overlopen, bevat cpy_{set} de keys van alle tuples t_S zodat de huidige $t_R.set \subseteq t_S.set$. Als cpy_{set} leeg is, dan weten we dat er geen enkele $t_S.set$ bestaat zodat $t_R.set \subseteq t_S.set$, dus $t_R.set$ kan in dit geval geen deelverzameling zijn.

Pseudocode

Algorithm 4 Indexed Nested Loops Join

Input: Relatie R, relatie S
Output: Alle paren ($t_R.key$, $t_S.key$) zodat $t_R.set \subseteq t_S.set$

```
1:  result = {}
2:  Map < key, set > invIndex = {}
3:  found = true
4:  for t_S ∈ S do
5:    for el ∈ t_S.set do
6:      if invIndex.containsKey(el) then
7:        invIndex.get(el).add(t_S.key)
8:      else
9:        set = {}
10:       set = set ∪ {t_S.key{
11:       invIndex.put(el, set)
12:     end if
13:   end for
14: end for
15: for t_R ∈ R do
16:   first = t_R.set[0]
17:   if invIndex.containsKey(first) then
18:     cpy = newHashSet <> (invIndex.get(first))
19:     for i = 1 to |t_R.set| do
20:       next = t_R.set[i]
21:       if cpy ≠ ∅ and found then
22:         found = next ∈ invIndex.key
23:         if found then
24:           cpy = cpy ∩ invIndex.get(next)
25:         end if
26:       end if
27:     end for
28:     for el ∈ cpy do
29:       result = result ∪ (t_R.key, el)
30:     end for
31:   end if
32: end for
33: return result
```

Voorbeeld

Als voorbeeld beschouwen we opnieuw relatie R uit tabel 21 en relatie S uit tabel 22. We passen INL toe op R en S. We maken eerst een inverted file S_{IF} voor relatie S. Deze ziet er zo uit:

Tabel 31: De inverted file S_{IF}.

key	values
5	$\{y_3, y_7\}$
8	$\{y_6\}$
9	$\{y_3, y_5\}$
10	$\{y_7\}$
11	$\{y_3\}$
13	$\{y_4\}$
15	$\{y_7\}$
18	$\{y_1\}$
20	$\{y_7\}$
25	$\{y_7\}$
27	$\{y_3\}$
28	$\{y_2\}$
29	$\{y_5\}$
34	$\{y_6\}$
39	$\{y_7\}$
46	$\{y_4\}$
67	$\{y_1, y_2\}$
70	$\{y_1, y_2\}$
88	$\{y_6\}$
90	$\{y_2\}$
96	$\{y_4\}$
99	$\{y_5\}$

We gaan elke tuple $t_R \in R$ af. Voor elk element $e_R \in t_R.set$ (zie kolom "values" in tabel 21) gaan we na of e_R voorkomt als key in S_{IF}.

1. De eerste tuple van R heeft x_1 als key. De set van deze tuple heeft 38 als eerste element. 38 is geen key in S_{IF} dus deze eerste tuple kunnen we overslaan.

2. De tweede tuple van R heeft x_2 als key. De set van deze tuple bevat elementen 28, 67 en 70. Deze drie zijn allemaal keys in S_{IF} en de bijhorende value is respectievelijk $\{y_2\}, \{y_1, y_2\}$ en $\{y_1, y_2\}$. We berekenen dus volgende intersectie: $\{y_2\} \cap \{y_1, y_2\} \cap \{y_1, y_2\} = \{y_2\}$. Dus weten we dat de set van tuple x_2 van R een deelverzameling is van de set van tuple y_2 van S.

3. De derde tuple van R heeft x_3 als key. Alle elementen van de set van deze tuple zijn keys in S_{IF}, behalve de laatste: 49. Omdat 49 niet in relatie S voorkomt, kan deze set geen deelverzameling zijn van een set uit relatie S.

4. De vierde tuple van R heeft x_4 als key. De set van deze tuple bestaat uit 13 en 46. Beiden zijn keys in S_{IF} en de bijhorende value is telkens $\{y_4\}$. De intersectie van $\{y_4\}$ met zichzelf is opnieuw $\{y_4\}$. Dus weten we dat de set van tuple x_4 van R een deelverzameling is van de set van tuple y_4 van S.

5. De vijfde tuple van R heeft x_5 als key. Alle elementen van de set van deze

tuple zijn keys in S_{IF}, behalve de laatste: 97. Omdat 97 niet in relatie S voorkomt, kan deze set geen deelverzameling zijn van een set uit relatie S.

6. De zesde tuple van R heeft x_6 als key. De set van deze tuple bestaat uit 18 en 70. Beiden zijn keys in S_{IF} en de bijhorende value is respectievelijk $\{y_1\}$ en $\{y_1, y_2\}$. Nu is het zo dat $\{y_1\} \cap \{y_1, y_2\} = \{y_1\}$. Dus weten we dat de set van tuple x_6 van R een deelverzameling is van de set van tuple y_1 van S.

7. De zevende tuple van R heeft x_7 als key. De set van deze tuple bestaat uit 5, 11 en 27. Alle elementen van de set van deze tuple zijn keys in S_{IF} en de bijhorende value is respectievelijk $\{y_3, y_7\}$, $\{y_3\}$ en $\{y_3\}$. Omdat $\{y_3, y_7\} \cap \{y_3\} \cap \{y_3\} = \{y_3\}$, weten we dat de set van tuple x_7 van R een deelverzameling is van de set van tuple y_3 van S.

Ons resultaat bestaat dus uit de paren (x_2, y_2), (x_4, y_4), (x_6, y_1) en (x_7, y_3).

Tijdscomplexiteit

Het opstellen van de inverted file S_{IF} gaat in sub-kwadratische tijd. Daarna moet voor elk tuple van R, element per element worden bekeken of het element in de inverted file zit. Indien dit telkens het geval is, moet een doorsnede worden berekend van een aantal verzamelingen, gelijk aan het aantal elementen dat in die R-tuple zit. Dit is de enige stap in dit algoritme die kwadratische tijd vergt.

Bij het worst-case scenario waarin $A = B_R = B_S = 1$ krijgen we een erg typische inverted file voor relatie S: er is slechts één key maar alle values komen hierbij terecht. Wanneer de tuples van R dan worden overlopen, blijkt dat we telkens de zelfde key moeten controleren en dus ook telkens de zelfde values als resultaat krijgen. Voor elk tuple van R krijgen we dus n_S containments. Het kwadratische karakter is ook hier dus duidelijk, al is er wel een verschil met SNL en PSJ: hier moet niet meer expliciet het NL-algoritme worden toegepast. Alhoewel de run time dus ook hier kwadratisch is, is het niet zeker dat die langer zal zijn dan de run time van NL.

Bewijs van correctheid

Als er sprake is van set containment $t_R.set \subseteq t_S.set$ dan zit elk element van $t_R.set$ in $t_S.set$ en dan zal $t_S.key$ ook in de inverted file zitten van elk element van $t_R.set$ en dus ook in de doorsnede hiervan. De set containment wordt dus inderdaad herkend.

Wanneer er geen set containment is, $t_R.set \not\subseteq t_S.set$, dan zal er een element van $t_R.set$ zijn dat niet in $t_S.set$ zit, zodat $t_S.key$ niet voorkomt in de inverted file van dit element. Aangezien de doorsnede wordt genomen van een aantal verzamelingen, waarvan minstens één deze $t_S.key$ niet bevat, zit die ook niet in de doorsnede en komt het koppel (t_R, t_S) niet voor in de finale lijst met set containments. Het algoritme is correct.

3.5 Inverted File Join (IFJ)

Het Inverted File Join (**IFJ**) algoritme voert een set-containment join uit op relatie R en S door hun inverted files R_{IF} en S_{IF} te combineren.

We construeren eerst een inverted file R_{IF} voor relatie R en een inverted file S_{IF} voor relatie S, net zoals de constructie van S_{IF} in paragraaf 3.4. De inverted file R_{IF} (S_{IF}) mapt elk element e_R (e_S) $\in t_R.set$ ($t_S.set$) van elke tuple t_R (t_S) van R (S) op de verzameling van keys van de tuples t_R (t_S) wiens set $t_R.set$ ($t_S.set$) het element e_R (e_S) bevat. De inverted file R_{IF} (S_{IF}) is dus een (key, $value$)-mapping waarbij key overeenkomt met alle elementen e_R (e_S) en $value$ overeenkomt met de keys van de tuples wiens set e_R (e_S) bevat.

Het idee is dat we nu beide inverted files R_{IF} en S_{IF} gaan overlopen om te weten te komen welke tuples van R en S intersecterende elementen bevatten in hun set-attribuut. Want als er intersecterende elementen zijn, dan kan er sprake zijn van set-containment.

We gaan als volgt te werk:

1. We houden een (key, $value$)-mapping $result$ bij die een key $t_R.key$ op een verzameling van keys $\{t_S.key\}$. Deze mapping vertelt ons dat welke sets $t_R.set$ bevat zijn in sets $t_S.sets$.

2. We beschouwen de (key, $value$)-mapping van de inverted files R_{IF} en S_{IF}. We nemen alle keys van beide inverted files tesamen en we noemen deze verzameling $elements_{RS}$, want de keys van de inverted files zijn eigenlijk de elementen van het set-attribuut van de tuples van de relaties R en S.

3. Vervolgens gaan we elk element $e \in elements_{RS}$ af en gaan we na of e voorkomt in R_{IF}.

    - Als e niet voorkomt in R_{IF}, dan wil dit zeggen dat e nergens voorkomt in relatie R. Bijgevolg kunnen er geen tuples t_R en t_S bestaan zodat $e \in t_R.set$ en $t_R.set \subseteq t_S.set$. Set containment is in dit geval dus onmogelijk.
    - Als e wel voorkomt in R_{IF}, dan vragen we de bijhorende set op en we noemen deze set l_R. Vervolgens gaan we elk element $t_R.key \in l_R$ af en we gaan ook na of e voorkomt in S_{IF}. Als e niet voorkomt in S_{IF}, dan weten we dat e nergens voorkomt in relatie S. Dus dan weten we dat elke set $t_R.set$ die e bevat, geen deelverzameling kan zijn van een set $t_S.set$. Dit geven we aan door ($t_R.key$, $\emptyset$) toe te voegen aan $result$. Als e wel voorkomt in S_{IF} dan vragen we de bijhorende set op en we noemen deze set l_S. Dan gaan we na of $t_R.key$ al voorkomt in $result$. Als $t_R.key$ nog niet voorkomt in $result$, dan is deze nieuw en dus voegen we ($t_R.key$, l_S) toe aan $result$. Als $t_R.key$ wel reeds voorkomt in $result$ dan vragen we zijn bijhorende set van tuples op en we noemen deze L_t. Dit wil zeggen dat de huidige e in een vorige iteratie al eens voorkwam in een set $t_S.set$ waarbij L_t de verzameling van keys $\{t_S.key\}$ identificeert. De huidige e komt in de huidige iteratie ook voor in de sets van de tuples die geïdentificeerd

worden door l_S. De doorsnede van l_S en L_t geeft ons een update van de verzameling van keys $\{t_S.key\}$ zodat $l_S \cap L_t = \{t_S.key\}$ en $t_R.set \subseteq t_S.set$. Dus we voegen $(t_R.key,\ l_S \cap L_t)$ toe aan $result$.

Pseudocode

Algorithm 5 Inverted File Join

Input: Relatie R, relatie S
Output: Alle paren ($t_R.key$, $\{t_S.key\}$) zodat $t_R.set \subseteq t_S.set$

1: $Map < key, set > result = \{\}$
2: $Map < key, set > invIndexR = \{\}$
3: $Map < key, set > invIndexS = \{\}$
4: **for** $t_S \in S$ **do**
5: **for** $el \in t_S.set$ **do**
6: **if** $invIndexS.containsKey(el)$ **then**
7: $set = invIndexS.get(el)$
8: **else**
9: $set = \{\}$
10: **end if**
11: $set = set \cup \{t_S.key\}$
12: $invIndexS.put(el, set)$
13: **end for**
14: **end for**
15: **for** $t_R \in R$ **do**
16: **for** $el \in t_R.set$ **do**
17: **if** $invIndexR.containsKey(el)$ **then**
18: $set = invIndexR.get(el)$
19: **else**
20: $set = \{\}$
21: **end if**
22: $set = set \cup \{t_R.key\}$
23: $invIndexR.put(el, set)$
24: **end for**
25: **end for**
26: $elementsSetR = invIndexR.keySet()$
27: $elementsSetS = invIndexS.keySet()$
28: $elementsRS = valueSetR \cup valueSetS$
29: **for** $e \in elementsRS$ **do**
30: $lR = invIndexR.get(e)$
31: $lS = invIndexS.get(e)$
32: **if** $lR \neq null$ **then**
33: **for** $keyR \in lR$ **do**
34: $Lt = \{\}$
35: **if** $lS \neq null$ **then**
36: **if** $!result.containsKey(keyR)$ **then**
37: $Lt = Lt \cup lS$
38: **else**
39: $Lt = Lt \cup result.get(keyR)$
40: $Lt = Lt \cap lS$
41: **end if**
42: **end if**
43: $result.put(keyR, Lt)$
44: **end for**
45: **end if**
46: **end for**
47: **return** result

Voorbeeld

Als voorbeeld beschouwen we opnieuw relatie R uit tabel 21 en relatie S uit tabel 22. We passen IFJ toe op R en S en maken eerst de inverted files R_{IF} en S_{IF}. Tabel 31 illustreert de inverted file voor relatie S. De inverted file voor relatie R ziet er zo uit:

Tabel 32: De inverted file R_{IF}.

key	values
5	$\{x_3, x_7\}$
8	$\{x_5\}$
10	$\{x_3\}$
11	$\{x_7\}$
13	$\{x_4\}$
15	$\{x_3\}$
18	$\{x_6\}$
20	$\{x_3\}$
25	$\{x_3\}$
27	$\{x_7\}$
28	$\{x_2\}$
34	$\{x_5\}$
38	$\{x_1\}$
46	$\{x_4\}$
49	$\{x_3\}$
67	$\{x_1, x_2\}$
70	$\{x_2, x_6\}$
83	$\{x_1\}$
88	$\{x_5\}$
90	$\{x_1\}$
97	$\{x_1, x_5\}$

Nu de inverted files voor relatie R en S geconstrueerd zijn (zie tot en met regel 25 van de pseudocode), gaan we de werking van het algoritme overlopen. Ons resultaat *result* is een (*key*, *value*)-mapping die een tuple van t_R van R mapt op een verzameling van tuples $\{t_S\}$ van S, zodat $t_R.set \subseteq t_S.set$ voor alle $t_S \in \{t_S\}$.

Volgens de (*key*, *value*)-mapping van R_{IF} en S_{IF}, nemen we alle *keys* van R_{IF} en S_{IF} tesamen en we noemen deze verzameling $elements_{RS}$. Dus $elements_{RS} = \{5, 8, 9, 10, 11, 13, 15, 18, 20, 25, 27, 28, 29, 34, 38, 39, 46, 49, 67, 70, 83, 88, 90, 96, 97, 99\}$.

Nu gaan we elk element $e \in elements_{RS}$ af en we gaan na of e voorkomt in R_{IF}. We zien dat elementen $9, 29, 39, 96$ en 99 niet voorkomen in R_{IF}. Deze elementen kunnen we alvast overslaan. De elementen $38, 49, 83$ en 97 komen voor in R_{IF}, maar niet in S_{IF}. Als een set $t_R.set$ één van deze elementen bevat, dan weten we zeker dat $t_R.set$ met t_R een tuple van R, geen deelverzameling kan zijn van een set $t_S.set$ met t_S een tuple van S. We geven dit aan door (*set*, $\emptyset$) aan *result* toe te voegen voor elke *set* die één van deze elementen bevat:

- Set x_1 bevat 38, dus we voegen (x_1, $\emptyset$) toe aan *result*.
- Set x_3 bevat 49, dus we voegen (x_3, $\emptyset$) toe aan *result*.

- Set x_1 bevat 83, dus we voegen (x_1, $\emptyset$) toe aan *result*.

- Set x_1 bevat 97, dus we voegen (x_1, $\emptyset$) toe aan *result*.

- Set x_5 bevat 97, dus we voegen (x_5, $\emptyset$) toe aan *result*.

Nu bestaat *result* uit de volgende entries: (x_1, $\emptyset$), (x_3, $\emptyset$) en (x_5, $\emptyset$). Aangezien we de elementen $9, 29, 38, 39, 49, 83, 96, 97$ en 99 al verwerkt hebben, moeten we nog deze elementen overlopen: $5, 8, 10, 11, 13, 15, 18, 20, 25, 27, 28, 34, 46, 67, 70, 88$ en 90.

We zoeken 5 op in R_{IF} en dit geeft ons $l_R = \{x_3, x_7\}$. We zoeken 5 op in S_{IF} en dit geeft ons $l_S = \{y_3, y_7\}$. x_3 komt reeds voor in *result* en heeft bijhorende set $L_t = \emptyset$. Aangezien $l_S \cap L_t = \{y_3, y_7\} \cap \emptyset = \emptyset$, voegen we ($x_3$, $\emptyset$) toe aan *result*. Hieruit leiden we af dat we alle elementen kunnen overslaan wiens bijhorende set in R_{IF} gelijk is aan $\{x_1\}, \{x_3\}$ of $\{x_5\}$. We slaan dus volgende elementen over: $8, 10, 15, 20, 25, 34, 88$ en 90. x_7 komt nog niet voor in *result*, dus we voegen (x_7, $\{y_3, y_7\}$) toe aan *result*.

Deze elementen moeten we nog overlopen: $11, 13, 18, 27, 28, 46, 67$ en 70.

- We zoeken 11 op in R_{IF} en in S_{IF}. Dit geeft $l_R = \{x_7\}$ en $l_S = \{y_3\}$. x_7 komt al voor in *result* en heeft bijhorende set $L_t = \{y_3, y_7\}$, dus $l_S \cap L_t = \{y_3\} \cap \{y_3, y_7\} = \{y_3\}$. We voegen $(x_7, \{y_3\})$ toe aan *result*.

- We zoeken 13 op in R_{IF} en in S_{IF}. Dit geeft $l_R = \{x_4\}$ en $l_S = \{y_4\}$. x_4 komt nog niet voor in *result*, dus we voegen (x_4, $\{y_4\}$) toe aan *result*.

- We zoeken 18 op in R_{IF} en in S_{IF}. Dit geeft $l_R = \{x_6\}$ en $l_S = \{y_1\}$. x_6 komt nog niet voor in *result*, dus we voegen (x_6, $\{y_1\}$) toe aan *result*.

- We zoeken 27 op in R_{IF} en in S_{IF}. Dit geeft $l_R = \{x_7\}$ en $l_S = \{y_3\}$. x_7 komt al voor in *result* en heeft bijhorende set $L_t = \{y_3\}$, dus $l_S \cap L_t = \{y_3\}$. We voegen $(x_7, \{y_3\})$ toe aan *result*.

- We zoeken 28 op in R_{IF} en in S_{IF}. Dit geeft $l_R = \{x_2\}$ en $l_S = \{y_2\}$. x_2 komt nog niet voor in *result*, dus we voegen (x_2, $\{y_2\}$) toe aan *result*.

- We zoeken 46 op in R_{IF} en in S_{IF}. Dit geeft $l_R = \{x_4\}$ en $l_S = \{y_4\}$. x_4 komt al voor in *result* en heeft bijhorende set $L_t = \{y_4\}$, dus $l_S \cap L_t = \{y_4\}$. We voegen $(x_4, \{y_4\})$ toe aan *result*.

- We zoeken 67 op in R_{IF} en in S_{IF}. Dit geeft $l_R = \{x_1, x_2\}$ en $l_S = \{y_1, y_2\}$. x_1 komt al voor in *result* en heeft bijhorende set $L_t = \emptyset$, dus $l_S \cap L_t = \emptyset$. We voegen (x_1, $\emptyset$) toe aan *result*. x_2 komt al voor in *result* en heeft bijhorende set $L_t = \{y_2\}$, dus $l_S \cap L_t = \{y_2\}$. We voegen $(x_2, \{y_2\})$ toe aan *result*.

- We zoeken 70 op in R_{IF} en in S_{IF}. Dit geeft $l_R = \{x_2, x_6\}$ en $l_S = \{y_1, y_2\}$. x_2 komt al voor in *result* en heeft bijhorende set $L_t = \{y_2\}$, dus $l_S \cap L_t = \{y_2\}$. We voegen $(x_2, \{y_2\})$ toe aan *result*. x_6 komt al voor in *result* en heeft bijhorende set $L_t = \{y_1\}$, dus $l_S \cap L_t = \{y_1\}$. We voegen $(x_6, \{y_1\})$ toe aan *result*.

Tabel 33: *result* op het einde van het IFJ-algoritme.

key tuple R	keys tuples S
x_1	$\emptyset$
x_2	$\{y_2\}$
x_3	$\emptyset$
x_4	$\{y_4\}$
x_5	$\emptyset$
x_6	$\{y_1\}$
x_7	$\{y_3\}$

Op het einde van het algoritme bevat *result* de output. Dit wordt geïllustreerd in tabel 33. Het resultaat bestaat dus uit volgende paren: (x_2, y_2), (x_4, y_4), (x_6, y_1) en (x_7, y_3).

Tijdscomplexiteit

Het opstellen van de inverted file gaat in sub-kwadratische tijd. Daarna worden de elementen overlopen. Elk element is naar verwachting aanwezig in $\frac{n_R B_R}{A}$ sets van R en in $\frac{n_S B_S}{A}$ sets van S. Elke tuple t_R wordt daarna nog $B_R - 1$ keer bekeken, waarbij telkens een doorsnede wordt gemaakt voor wat betreft de verzameling van keys $\{t_S.key\}$. Aangezien die doorsnede steeds kleiner wordt naarmate het algoritme vordert, wordt deze intersectiebewerking steeds sneller.

Het worst-case scenario waarbij $A = B_R = B_S = 1$ leidt tot twee inverted files die er gelijkaardig uitzien: slechts één key, met alle values erbij. Bij het overlopen gaan we dan telkens een koppel $(x_i, \{y_1, \ldots, y_n\})$ toevoegen aan de lijst *results*. Hier blijkt opnieuw het kwadratische karakter van alleen al het opstellen van deze lijst met resultaten. Ook in dit algoritme is het niet meer nodig om nog een expliciete uitvoering van de verificatiefase te doen, zoals bij SNL en PSJ.

Bewijs van correctheid

Als er een set containment is, dus $t_R.set \subseteq t_S.set$, dan zit $t_S.key$ in de inverted file van elk element van $t_R.set$, dus zal het ook in de doorsnede zitten en zal het algoritme de set containment van dit koppel herkennen.
Indien er geen set containment is, is er een element $x \in t_R.set$ waarvoor $x \notin t_S.set$. In de inverted file zal bij dat element dus wel $t_R.key$ voorkomen, maar niet $t_S.key$. Er wordt telkens een doorsnede genomen, dus $t_S.key$ kan nooit meer opduiken in result.

4 Set joins in SQL en XQuery

In dit hoofdstuk gaan we na in welke mate de querytalen SQL en XQuery de verschillende soorten set joins uit paragraaf 2.3 ondersteunen. We leggen hierbij de nadruk op de query die de set containment join uitdrukt, die we vanaf nu de "set containment query" noemen en we vergelijken de performantie van die set containment query in SQL en in XQuery met elkaar.

De geïmplementeerde set join queries worden losgelaten op een zelfgemaakte database genaamd "Masterproef". Deze database bevat drie tabellen "Thesis", "Student"en "Thesisvakken" die we vanaf nu relaties noemen. De relaties Thesis en Student hebben twee integer attributen: "ID" en "Vak". Het ID-attribuut wordt gebruikt om een thesis of student te identificeren. Het vak-attribuut in de Thesis-relatie definieert de vakken die men moet beheersen voordat men een bepaald thesisonderwerp mag nemen. Het vak-attribuut in de Student-relatie definieert de vakken die een bepaalde student reeds heeft afgelegd. De relatie Thesisvakken wordt alleen gebruikt om de division Student $\div$ Thesisvakken uit te voeren, die nog zal besproken worden. Tabel 34 illustreert deze database. Voor deze tabel wordt het attribuut "ID" op dezelfde manier gebruikt als in tabel 7 van paragraaf 2.2.2.

Thesis

ID	Vak
0	0
0	1
1	1
2	2
2	3

Student

ID	Vak
0	2
0	3
0	4
1	0
1	1
2	2

Thesisvakken

Vak
2
3

Tabel 34: De relaties "Thesis","Student" en "Thesisvakken" uit database "Masterproef".

4.1 Set joins in SQL

SQL (Structured Query Language) is een querytaal die gebruikt wordt om data in databases op te slaan, te manipuleren en op te vragen [14]. Om met SQL te kunnen werken, moet men gebruik maken van een database query processor. We werken met een gratis versie van de database query processor DB2, genaamd DB2 Express-C [15]. Om de set join queries uit te voeren, maken we gebruik van de DB2 commandline processor (CLP). Dit is een krachtige tool die ons toelaat SQL statements uit te voeren.

In deze paragraaf gaan we na hoe de verschillende soorten set joins in SQL kunnen uitgedrukt worden.

Het resultaat van elke set join query (behalve de division) bestaat steeds uit koppeltjes van "Thesis.ID" en "Student.ID". Het resultaat van de division bestaat uit een lijst van "Student.ID".

4.1.1 Set containment

We passen nu de set containment query toe op relaties Thesis en Student. Een SQL query om de set containment join $Thesis \underset{Thesis.Vak \subseteq Student.Vak}{\bowtie} Student$ uit te voeren, is:

```
SELECT Thesis.ID AS "Thesis.ID", Student.ID AS "Student.ID"
FROM Thesis, Student
EXCEPT
SELECT T.ID, S1.ID
FROM Thesis AS T, Student AS S1
WHERE T.Vak NOT IN (SELECT Vak FROM Student AS S WHERE S1.ID = S.ID);
```

We merken op dat deze SQL query ongeveer dezelfde werkwijze heeft als de relationele algebra-expressie voor de set containment join van paragraaf 2.4.2: we selecteren namelijk alle ID-koppels van Thesis en Student die toegelaten zijn. Dit zijn dus alle ID-koppels buiten de ID-koppels die niet toegelaten zijn, vandaar dat hier het SQL statement EXCEPT wordt gebruikt. Een ID-koppel van Thesis en Student wordt niet toegelaten als er een vak van Thesis niet behoort tot de vakken van Student. We gebruiken NOT IN om dit resultaat te bekomen.

4.1.2 Division

Hier bespreken we de division query voor de relaties Student en Thesisvakken. Een SQL query om de division $Student \div Thesisvakken$ uit te voeren, is:

```
SELECT Student.ID AS "Student.ID"
FROM Student
EXCEPT
SELECT S1.ID
FROM ThesisDivision, Student AS S1
WHERE ThesisDivision.Vak NOT IN (SELECT Vak
 FROM Student AS S
 WHERE S1.ID = S.ID);
```

In deze query wordt de tabel ThesisDivision gebruikt. Dit is gewoon een andere naam voor de relatie Thesisvakken uit tabel 34.

4.1.3 Set equality

We passen nu de set equality query toe op relaties Thesis en Student. Een SQL query om de set equality join $Thesis \underset{Thesis.Vak = Student.Vak}{\bowtie} Student$ uit te voeren, is:

```
(SELECT DISTINCT Thesis.ID AS "Thesis.ID", Student.ID AS "Student.ID"
FROM Thesis, Student
EXCEPT
SELECT DISTINCT T.ID, S1.ID
FROM Thesis AS T, Student AS S1
WHERE T.Vak NOT IN (SELECT Vak
FROM Student AS S
```

```
WHERE S1.ID = S.ID))
INTERSECT
(SELECT DISTINCT Thesis.ID AS "Thesis.ID", Student.ID AS "Student.ID"
FROM Thesis, Student
EXCEPT
SELECT DISTINCT T1.ID, S.ID
FROM Thesis AS T1, Student AS S
WHERE S.Vak NOT IN (SELECT Vak
FROM Thesis AS T
WHERE T1.ID = T.ID));
```

Uit paragraaf 2.3.3 weten we dat we de set equality join aan de hand van de set containment join kunnen definiëren. We gebruiken deze werkwijze ook in de set equality query door de doorsnede (INTERSECT) te nemen van de set containment query's $Thesis \underset{Thesis.Vak \subseteq Student.Vak}{\bowtie} Student$ en $Student \underset{Student.Vak \subseteq Thesis.Vak}{\bowtie} Thesis$.

4.1.4 Set overlap (standaard equijoin)

De set overlap query op relaties Thesis en Student komt eigenlijk overeen met een standaard equijoin op het Vak-attribuut. Dit hebben we reeds opgemerkt in paragraaf 2.3.4.

De eenvoudigste SQL query om de set overlap join

$$Thesis \underset{Thesis.Vak \cap Student.Vak \neq \emptyset}{\bowtie} Student$$

uit te voeren, is:

```
SELECT DISTINCT Thesis.ID AS "Thesis.ID", Student.ID AS "Student.ID"
FROM Thesis, Student
WHERE Thesis.Vak = Student.Vak;
```

Het is overduidelijk dat deze set overlap query overeenkomt met de standaard equijoin.

4.1.5 Set disjointness

We passen nu de set disjointness query toe op relaties Thesis en Student. Een SQL query om de set disjointness join $Thesis \underset{Thesis.Vak \cap Student.Vak = \emptyset}{\bowtie} Student$ uit te voeren, is:

```
SELECT Thesis.ID AS "Thesis.ID", Student.ID AS "Student.ID"
FROM Thesis, Student
EXCEPT
SELECT T.ID, S.ID
FROM Thesis AS T, Student AS S
WHERE S.Vak = T.Vak;
```

De set disjointness query is eigenlijk het "tegengestelde"van de set overlap query. In dit opzicht kunnen we het resultaat van de set overlap query beschouwen als de niet-toegelaten ID-koppels. Het resultaat van de set disjointness query bestaat dan uit alle ID-koppels behalve de niet-toegelaten ID-koppels. We gebruiken opnieuw het SQL statement EXCEPT om dit resultaat te bekomen.

4.2 Set joins in XQuery

XQuery is een querytaal die gebruikt wordt om elementen en attributen op te vragen te vinden uit XML-documenten. Het wordt vooral gebruikt om query's toe te passen op XML-data. Je zou dus kunnen zeggen dat XQuery en SQL gelijkaardig zijn: SQL gebruikt data uit databases en XQuery gebruikt data uit XML-documenten [19].

Net zoals bij SQL, heeft men ook een database query processor nodig om met XQuery te kunnen werken. De XQuery processor waarmee we werken is BaseX en meer bepaald de Standalone Mode [20]. Dit is een commandline console mode van waaruit alle XQuery database commando's kunnen uitgevoerd worden [21].

In deze paragraaf gaan we na hoe we de verschillende set joins in XQuery kunnen uitdrukken.

Aangezien XQuery op XML-files werkt, zetten we de relaties Thesis en Student van tabel 34 om in onderstaande XML-code, genaamd "ThesisStudent.xml".

```
<?xml version="1.0" encoding="UTF-8"?>
<masterproef>
        <thesis>
                <id>0</id>
                <vak>0</vak>
                <vak>1</vak>
        </thesis>
        <thesis>
                <id>1</id>
                <vak>1</vak>
        </thesis>
        <thesis>
                <id>2</id>
                <vak>2</vak>
                <vak>3</vak>
        </thesis>
        <student>
                <id>0</id>
                <vak>2</vak>
                <vak>3</vak>
                <vak>4</vak>
        </student>
        <student>
                <id>1</id>
                <vak>0</vak>
                <vak>1</vak>
        </student>
        <student>
                <id>2</id>
                <vak>2</vak>
        </student>
</masterproef>
```

Om de division te kunnen uitvoeren, zetten we de relaties Student en Thesis-vakken van tabel 34 om in onderstaande XML-code, genaamd "divisionThesis-Student.xml".

```
<?xml version="1.0" encoding="UTF-8"?>
<masterproef>
        <thesis>
                <vak>2</vak>
                <vak>3</vak>
        </thesis>
        <student>
                <id>0</id>
                <vak>2</vak>
                <vak>3</vak>
                <vak>4</vak>
        </student>
        <student>
                <id>1</id>
                <vak>0</vak>
                <vak>1</vak>
        </student>
        <student>
                <id>2</id>
                <vak>2</vak>
        </student>
</masterproef>
```

Bij het bespreken van de set join query's zullen we deze xml-bestanden als leidraad gebruiken.

Het resultaat van elke set join query (behalve de division) heeft dit formaat:

```
<results>
        <pair>
                <thesisID>0</thesisID>
                <studentID>1</studentID>
        </pair>
        <pair>
                <thesisID>1</thesisID>
                <studentID>1</studentID>
        </pair>
        <pair>
                <thesisID>2</thesisID>
                <studentID>0</studentID>
        </pair>
</results>
```

Het resultaat van de division heeft dit formaat:

```
<results>
        <studentID>0</studentID>
</results>
```

4.2.1 Set containment

We passen nu de set containment query toe op relaties Thesis en Student, die voorkomen in "ThesisStudent.xml". Een XQuery query om de set containment join $Thesis \underset{Thesis.Vak \subseteq Student.Vak}{\bowtie} Student$ uit te voeren, is:

```
<results>
{
let $masterproef := doc('ThesisStudent.xml')/masterproef
for $t in $masterproef/thesis
for $s in $masterproef/student
where fn:deep-equal($t/vak[. = $s/vak], $t/vak)
return (<pair>{<thesisID>{$t/id/data()}</thesisID>, <studentID>{$s/id/data()}</studentID>}
}
</results>
```

Het belangrijkste statement is

```
fn:deep-equal($t/vak[. = $s/vak], $t/vak)
```

De functie "fn:deep-equal()" wordt gebruikt om de volledige inhoud van twee sequences met elkaar te vergelijken [16]. De operatie $t/vak[. = $s/vak] geeft alle elementen terug uit $t/vak die voorkomen in $s/vak [17]. Containment van $t/vak in $s/vak kunnen we dus nagaan door het volledige statement toe te passen.

4.2.2 Division

Hier bespreken we de division query voor de relaties Student en Thesisvakken, die voorkomen in "divisionThesisStudent.xml". Een XQuery query om de division $Student \div Thesisvakken$ uit te voeren, is:

```
<results>
{
let $masterproef := doc('divisionThesisStudent.xml')/masterproef
let $thesisvakken := $masterproef/thesis/vak
for $s in $masterproef/student
where fn:deep-equal($thesisvakken[. = $s/vak], $thesisvakken)
return (<studentID>{$s/id/data()}</studentID>)
}
</results>
```

Net zoals bij de set containment query, is ook hier

```
fn:deep-equal($thesisvakken[. = $s/vak], $thesisvakken)
```

het belangrijkste statement. Er wordt opnieuw gebruik gemaakt van de "fn:deep-equal()" functie en de operatie [. =] om de division te bepalen.

4.2.3 Set equality

We passen nu de set equality query toe op relaties Thesis en Student, die voorkomen in "ThesisStudent.xml". Een XQuery query om de set equality join $Thesis \underset{Thesis.Vak = Student.Vak}{\bowtie} Student$ uit te voeren, is:

```
<results>
{
let $masterproef := doc('ThesisStudent.xml')/masterproef
for $t in $masterproef/thesis
for $s in $masterproef/student
where fn:deep-equal($t/vak, $s/vak)
return (<pair>{<thesisID>{$t/id/data()}</thesisID>, <studentID>{$s/id/data()}</studentID>}
}
</results>
```

We gebruiken opnieuw de functie "fn:deep-equal" om de gelijkheid van de sequences $t/vak en $s/vak na te gaan.

4.2.4 Set overlap (standaard equijoin)

De set overlap query op relaties Thesis en Student (uit "ThesisStudent.xml") komt overeen met een standaard equijoin op het Vak-attribuut.

De eenvoudigste XQuery query om de set overlap join

$$Thesis \underset{Thesis.Vak \cap Student.Vak \neq \emptyset}{\bowtie} Student$$

uit te voeren, is:

```
<results>
{
let $masterproef := doc('ThesisStudent.xml')/masterproef
for $t in $masterproef/thesis
for $s in $masterproef/student
where $t/vak = $s/vak
return (<pair>{<thesisID>{$t/id/data()}</thesisID>, <studentID>{$s/id/data()}</studentID>}
}
</results>
```

De test $t/vak = $s/vak geeft true als de sequences $t/vak en $s/vak minstens 1 gemeenschappelijk element hebben, wat dus exact overeenkomt met de voorwaarde van de set overlap join of de standaard equijoin [17].

4.2.5 Set disjointness

We passen nu de set disjointness query toe op relaties Thesis en Student uit "ThesisStudent.xml". Een XQuery query om de set disjointness join $Thesis \underset{Thesis.Vak \cap Student.Vak = \emptyset}{\bowtie} Student$ uit te voeren, is:

```
<results>
{
let $masterproef := doc('ThesisStudent.xml')/masterproef
for $t in $masterproef/thesis
for $s in $masterproef/student
where not($t/vak = $s/vak)
return (<pair>{<thesisID>{$t/id/data()}</thesisID>, <studentID>{$s/id/data()}</studentID>}
}
</results>
```

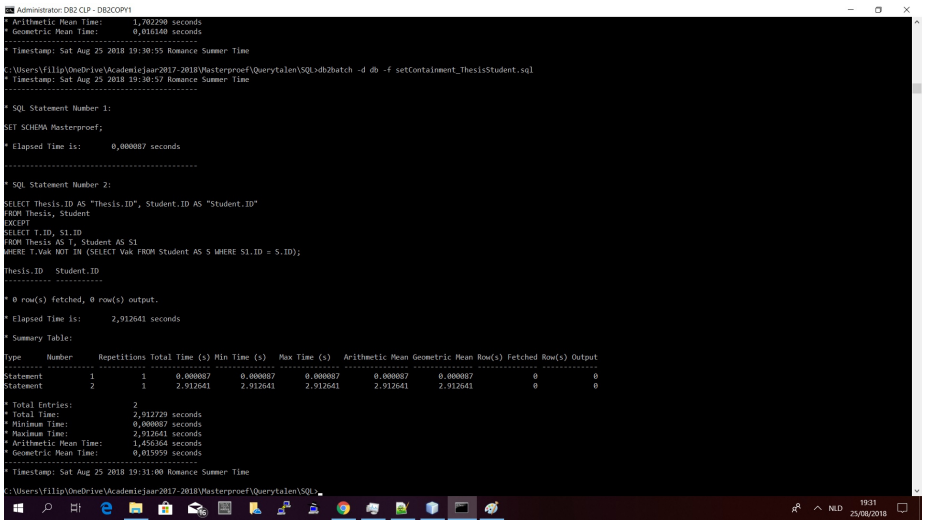
De set disjointness query is eigenlijk het "tegengestelde" van de set overlap query, zoals reeds besproken in 4.1.5. De test not($t/vak = $s/vak) geeft true als de sequences $t/vak en $s/vak geen gemeenschappelijke elementen hebben [18]. Dit is dus exact wat we nodig hebben om set disjointness te bepalen.

4.3 Performantie van de set containment query in SQL en XQuery

Hier gaan we de performantie van de set containment query in SQL en XQuery met elkaar vergelijken.

Het db2-commando db2batch maakt het mogelijk om de performantie van SQL-statements na te gaan [22]. Hierdoor kunnen we een beeld krijgen van de performantie van onze set containment query. De afbeelding hieronder is een screenshot van de DB2 commandline processor, na het toepassen van db2batch op de set containment query. Volgende parameters werden ingesteld voor het uitvoeren van deze query:

- Aantal verschillende studenten in de Student-relatie: 100
- Aantal verschillende thesissen in de Thesis-relatie: 100
- Aantal vakken voor elke thesis: 5
- Aantal vakken dat elke student volgt: 20
- Aantal vakken waaruit gekozen kan worden: 100



Uit de screenshot leiden we af dat het ongeveer 2,91 seconden duurt om deze set containment query in SQL uit te voeren.

In de XQuery processor BaseX moet men geen expliciet commando uitvoeren om de performantie van een query na te gaan. Telkens wanneer er een query wordt uitgevoerd, vermeldt de commandline processor de uitvoeringstijd van

A	|Thesis.Vak|	|Student.Vak|	N	runtime SQL(s)	runtime XQuery (s)
100	5	20	100	2.91	0.126
100	5	20	250	50.679	
100	5	20	500	402.914	3.436
100	5	20	1000		24.027
100	5	20	1500		29.023
100	5	20	2000		99.032

Tabel 35: Uitvoeringstijden van de set containment query in SQL en XQuery

die bepaalde query.

We maken een overzicht van de uitvoeringstijd van de set containment query in SQL en XQuery en we doen dit voor verschillende parameters:

- |Thesis.Vak| is het aantal vakken voor elke thesis
- |Student.Vak| is het aantal vakken dat elke student volgt
- N is het aantal studenten en het aantal thesissen
- A is het aantal vakken waaruit gekozen kan worden

Tabel 35 geeft de uitvoeringstijden weer. Uit de tabel leiden we af dat de set containment query in SQL vanaf N = 500 zeer traag begint te lopen. We zien duidelijk dat de set containment query in XQuery veel sneller wordt uitgevoerd.

We besluiten dat de set containment query in XQuery veel performanter is dan in SQL.

n_R	n_S	A	B_S	B_R	run time [s]
10000	10000	100	20	5	7.338
5000	5000	100	20	5	1.439
10000	10000	40	20	5	8.788
5000	5000	1	1	1	10.137
10000	10000	1	1	1	/
5000	5000	20	20	20	23.094

Tabel 36: Resultaten van de experimenten voor het NL-algoritme.

5 Experimenten

Dit hoofdstuk beschrijft de resultaten van de experimenten die werden uitgevoerd met verschillende set join algoritmes. Gezien de impact op de performantie van de implementaties van de set-join algoritmes, bespreken we eerst de systeemspecificaties. Nadien worden de vijf besproken algoritmes apart bekeken om te zien hoe goed ze presteren bij verschillende keuzes van de parameters. Tot slot worden de algoritmes tegen elkaar uitgespeeld om te zien welk algoritme de beste keuze is bij bepaalde keuzes van de parameters.

5.1 Maatstaf voor performantie

Om de performantie nauwkeurig te kunnen meten, wordt de "runtime" als maatstaf gebruikt: de hoeveelheid tijd die een set join algoritme nodig heeft om twee relaties R en S te joinen door middel van set containment.

5.2 Systeemspecificaties

Bij het testen van de snelheid van een implementatie, is het belangrijk om rekening te houden met de capaciteiten van het systeem waarop dit gebeurt. Zo zal een systeem met meer RAM-geheugen of met een betere CPU de implementatie sneller uitvoeren.

Een overzicht van de specificaties van het gebruikte systeem:

- Besturingssysteem: Windows 10 professional
- Processor: Intel(R) Core(TM) i7 CPU
- Werkgeheugen (RAM): $8,00$ GB
- Type systeem: 64 bitsbesturingssysteem
- Java versie: 8 update 51 (dus JDK 1.8.0_51)

5.3 Individuele bespreking van de verschillende algoritmes

5.3.1 Het NL-algoritme

De opmerkingen over tijdsefficiëntie worden bevestigd door de experimenten: enkele resultaten zijn opgenomen in Tabel 36. Merk op dat wanneer n_R en n_S

Algoritme	n_R	n_S	A	B_S	B_R	b	run time [s]
NL	10000	10000	100	20	5		7.338
SNL	10000	10000	100	20	5	5 (int)	7.845
SNL	10000	10000	100	20	5	10 (int)	7.026
SNL	10000	10000	100	20	5	31 (int)	2.847
SNL	10000	10000	100	20	5	32 (long)	2.853
SNL	10000	10000	100	20	5	47 (long)	2.567
SNL	10000	10000	100	20	5	63 (long)	2.533
SNL	10000	10000	100	20	5	100 (string)	5.093
SNL	5000	5000	1	1	1	1 (int)	19.518
SNL	5000	5000	1	1	1	1 (long)	10.854
SNL	5000	5000	1	1	1	1 (string)	9.137
SNL	5000	5000	20	20	20	20 (int)	23.775
SNL	5000	5000	20	20	20	20 (long)	14.780
SNL	5000	5000	20	20	20	20 (string)	11.729

Tabel 37: Resultaten van de experimenten voor het SNL-algoritme.

halveren, de tijd ongeveer door vier wordt gedeeld. Wanneer we de eerste en de derde lijn vergelijken, zien we dat het algoritme langer moet lopen wanneer zelfde aantallen gekozen worden uit een kleinere verzameling: hierdoor duurt het gemiddeld langer om een inclusie te controleren.

De vierde tot zesde lijn zijn voorbeelden van een worst case scenario: elke $t_R.set$ is deelverzameling van elke $t_S.set$ (omdat alle sets gelijk zijn). Wanneer n_R en n_S te groot zijn, vraagt dit zodanig veel geheugen dat we een foutmelding krijgen, zoals in de vijfde regel.

5.3.2 Het SNL-algoritme

Enkele resultaten van de experimenten voor het SNL-algoritme zijn te vinden in Tabel 37.

De trend is duidelijk: een kortere signature length zorgt voor een langere run time. Een opmerking is hierbij wel belangrijk: het is van belang hoe de signature wordt bijgehouden. Zoals eerder besproken is de signature een bit string die geïnterpreteerd wordt als getal. Wanneer dit getal als integer (int) wordt bijgehouden, is het gelimiteerd tot 31 bits. Elementen groter dan 31 worden dan genegeerd, zodat het in deze implementatie onmogelijk is met exacte signatures te werken, indien $A > 31$. Dit effect wordt uiteraard verschoven tot 63 wanneer we met long integers werken. Er is geen significant verschil merkbaar tussen het werken met long en int bij een gelijke signature length, zoals we kunnen zien wanneer we de run times voor $b = 31$ (met int) voor $b = 32$ (met long) vergelijken.

Door middel van strings kunnen we wel met exacte signatures werken, maar dit is beduidend trager voor de parameterwaarden die hier werden behandeld. Een voordeel is dat bij exacte signatures de verificatiefase niet meer nodig is, maar de tijdwinst die hierdoor geboekt wordt, is veel kleiner dan het tijdverlies

n_R	n_S	A	B_S	B_R	b	k	PSJ [s]	PSJCW [s]	NL [s]
10000	10000	100	20	5	10 (int)	5	159.799	164.790	7.338
10000	10000	100	20	5	32 (long)	5	14.437	14.604	7.338
10000	10000	100	20	5	100 (string)	5	22.654	20.164	7.338
10000	10000	800	2	1	31 (int)	10	1.427	1.476	3.034
10000	10000	800	2	1	8 (int)	10	4.281	4.250	3.034
5000	5000	1	1	1	1 (int)	10	10.302	10.789	10.137
1000	1000	20	20	20	20 (int)	10	8.017	7.260	0.327

Tabel 38: Resultaten van de experimenten voor het PSJ-algoritme.

veroorzaakt door het werken met strings.

Dat het voordelig is te werken met langere signatures is geen verrassing: bij korte signature length worden er snel heel veel false drops gecreëerd, een veelvoud van de echte drops. De controle hiervan is erg tijdsintensief. Wanneer de signature length te klein wordt gekozen, is SNL zelfs trager dan NL (vergelijk de eerste en de tweede lijn uit de tabel). Dit betekent dus dat er zo veel false drops zijn, dat de verificatie ervan, samen met het voorafgaand opstellen van de signatures en de probing phase, langer duurt dan alle koppels te controleren. In het voorbeeld van de tweede lijn is $B_S = 20$ en $b = 5$, wat betekent dat een tuple van S heel vaak de signature 11111 zal opleveren en dus voor elk tuple van R een drop zal genereren.

De onderste zes lijnen zijn instanties van het worst case scenario, waarbij elke combinatie een drop oplevert. Het is opvallend en contra-intuïtief dat int hier de traagste keuze is, gevolgd door long en string.

5.3.3 Het PSJ-algoritme

Tabel 38 geeft enkele resultaten van experimenten met het PSJ-algoritme. Vooral bij verspreide (sparse) relaties is PSJ sneller dan NL, althans wanneer de signature length b en het aantal klasses k goed worden gekozen. Vooral een te kleine b zorgt zoals verwacht voor een snel groeiende run time.

Voorts valt op dat de verschillen tussen de klassieke PSJ en de Carter-Wegmann PSJ (CWPSJ) erg klein zijn, wat niet hoeft te verbazen aangezien de gegenereerde data random en uniform verdeeld zijn.

De onderste twee lijnen geven een worst case scenario weer, waarbij elke combinatie een set containment oplevert. Vooral de onderste lijn is opvallend: hier werd $n_R = n_S = 1000$ gekozen om de tijd beperkt te houden, maar PSJ is hier duidelijk veel trager dan NL.

5.3.4 Het INL-algoritme

In Tabel 39 zien we dat INL voor de gekozen parameterwaarden beduidend sneller is dan de naïeve NL-benadering. Het halveren van n_R en n_S zorgt voor

n_R	n_S	A	B_S	B_R	run time INL [s]	run time NL [s]
10000	10000	100	20	5	2.228	7.338
5000	5000	100	20	5	0.510	1.439
10000	10000	800	20	5	0.333	7.068
10000	10000	40	20	5	6.000	9.585
10000	10000	30	20	5	11.289	13.226
5000	5000	1	1	1	6.245	10.137
5000	5000	20	20	20	15.478	23.094

Tabel 39: Resultaten van de experimenten voor het INL-algoritme.

n_R	n_S	A	B_S	B_R	run time IFJ [s]	run time NL [s]
10000	10000	100	20	5	4.709	7.338
5000	5000	100	20	5	1.353	1.324
10000	10000	800	20	5	0.527	7.068
10000	10000	40	20	5	18.577	9.585
10000	10000	30	20	5	/ (*)	13.226
4000	4000	30	20	5	4.064	1.171
10000	10000	100	3	1	0.794	3.397
5000	5000	1	1	1	6.017	10.137
5000	5000	20	20	20	13.807	23.094

Tabel 40: Resultaten van de experimenten voor het IFJ-algoritme.

een ongeveer vier keer kortere run time, wat het kwadratische karakter aangeeft (vergelijk de eerste en de tweede lijn). Als we de eerste en derde lijn vergelijken, zien we dat wanneer A groter wordt gekozen, het algoritme sneller gaat, wat logisch is: elk element zit gemiddeld in veel minder tuples van S en R. Deze winst wordt in het NL-algoritme niet geboekt. Bij een veel kleinere A (vierde en vijfde lijn) zien we dat het INL-algoritme duidelijk trager wordt: elk element zit dan in een groot aantal tuples, waardoor doorsnedes van in verhouding grotere verzamelingen moeten worden gemaakt.

Ook bij de worst case scenario's in de onderste twee lijnen zien we dat INL iets sneller is dan NL, al is het verschil niet erg groot. Dit lijkt logisch aangezien een groot deel van de tijd hier gebruikt zal worden om de resultaten (alle koppels) weg te schrijven.

5.3.5 Het IFJ-algoritme

Uit Tabel 40 blijkt het kwadratische gedrag van de run time: wanneer n_R en n_S worden gehalveerd, wordt de run time bij benadering gedeeld door vier. Verder is ook duidelijk dat bij een verspreide (sparse) database (met minder verwachte set containments, zoals wanneer A groot is vergeleken met B_R en B_S) het algoritme veel sneller gaat. Dit is logisch aangezien één van de twee verzamelingen waarvan doorsnedes moeten berekend worden, snel kleiner wordt. Wanneer dit niet het geval is, zoals bij $A = 30$, $B_S = 20$, $B_R = 5$, lukt het wegens geheugen-

tekort niet het programma te runnen bij $n_R = n_S = 10000$, dit werd in de tabel aangeduid met (*). Bij $n_R = n_S = 4000$ lukt het wel, maar blijkt IFJ trager te zijn dan NL.

Het worst case scenario wordt weergegeven in de onderste twee lijnen, waaruit blijkt dat IFJ ook hier beduidend sneller is dan NL.

5.4 Vergelijking van de verschillende algoritmes

Het kwadratische karakter van de run time bij elk algoritme werd in de afzonderlijke analyse al duidelijk. Het blijkt ook uit Figuur 1, waarin voor de verschillende algoritmes de run times zijn weergegeven voor de parameterkeuze $A = 100, B_R = 4, B_S = 24$ en voor keuzes van $n_R = n_S$ die variëren van 5000 tot 40000 met toenamefactor 2. Bij IFJ en INL kwam er een foutmelding wegens geheugentekort bij $n_R = n_S = 40000$. Het feit dat de log-log plots bijna rechten zijn, toont aan dat het machtsverband op kleine afwijkingen na, gevolgd wordt.

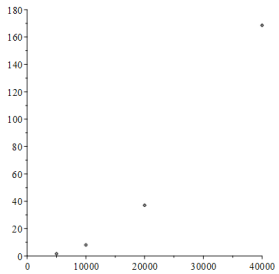
In Tabel 41 wordt voor verschillende waarden van n_R, n_S, A, B_R en B_S de run time van de verschillende algoritmes vergeleken. Voor SNL en PSJ wordt telkens de tijd weergegeven voor de optimale keuze van signature length (en aantal klassen).

Het is duidelijk dat de keuze voor het juiste algoritme samenhangt met de keuze van de parameterwaarden. Zo blijken de algoritmes met inverted files, INL en IFJ, vaak efficiënt, maar wanneer B_R en B_S groot zijn in vergelijking met A zijn ze soms, trager dan SNL.

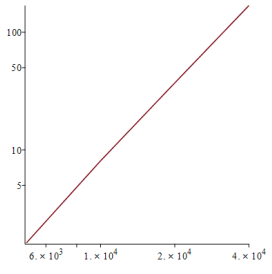
Opvallend is ook hoe PSJ soms erg lang duurt en ruim trager is dan NL. Ook IFJ is in sommige gevallen trager dan de naïeve benadering van NL. Toch zijn er parameterkeuzes waarbij PSJ de snelste keuze is.

In het algemeen blijkt dat INL een prima keuze is voor een brede waaier aan parameterwaarden. Het is niet altijd het snelste algoritme, maar wanneer het dat niet is, is het verschil met de winnaar niet groot.

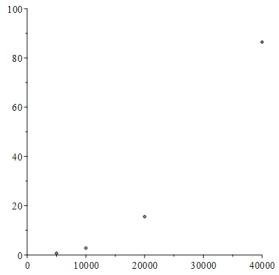
Een vergelijking maken met resultaten uit de papers is niet eenvoudig aangezien de systeemspecificaties verschillen. Het is dan ook geen goed idee zomaar run times uit de literatuur te vergelijken met onze resultaten. Wel kunnen we op zoek gaan naar tendenzen in de literatuur en kijken of ze al dan niet bevestigd worden door onze experimenten. Zo zien we dat volgens paragraaf 5.6 van [3] het PSJ-algoritme consistent beter is dan SNL. Dit bleek niet uit onze experimenten: PSJ is daarin vaak veel trager. Dit zou te maken kunnen hebben met het feit dat we hier werken met de main-memory variant van deze algoritmes, wat niet het geval was in [3]. Een observatie die wel bevestigd wordt, is het effect van de signature length op de run time van het SNL-algoritme: een te kleine signature length zorgt voor een grotere run time, maar eens de signature length groot genoeg is, blijft de run time ongeveer constant. Deze observatie uit paragraaf 5.8 van [3] vinden we ook terug in tabel 37.



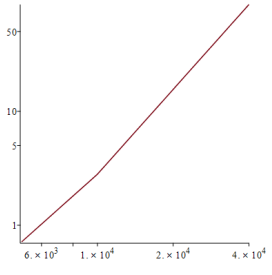
(a) Run time NL



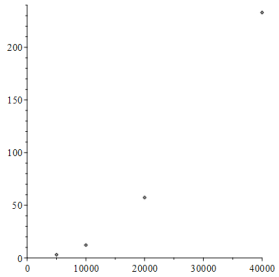
(b) Run time NL (log-log)



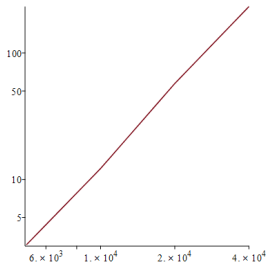
(c) Run time SNL



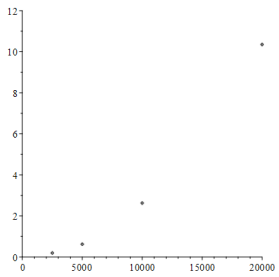
(d) Run time SNL (log-log)



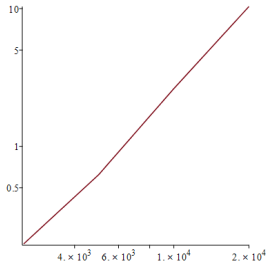
(e) Run time PSJ



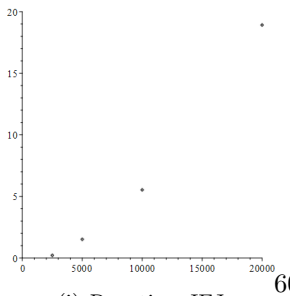
(f) Run time PSJ (log-log)



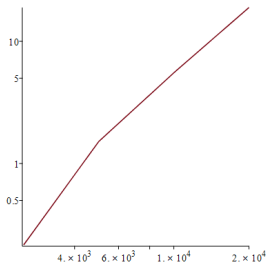
(g) Run time INL



(h) Run time INL (log-log)



(i) Run time IFJ



(j) Run time IFJ (log-log)

Figuur 1: Run times voor de verschillende algoritmes, in functie van $n_R = n_S$, bij $A = 100, B_R = 4, B_S = 24$.

id	n_R	n_S	A	B_S	B_R	NL	SNL	PSJ	INL	IFJ
1	10000	10000	100	20	5	7.224	2.538	5.751	**1.780**	5.471
2	10000	10000	1000	20	5	6.733	2.049	4.541	0.389	**0.330**
3	5000	5000	30	20	5	1.918	**1.171**	23.633	2.355	4.775
4	10000	10000	30	10	10	5.520	**2.515**	2.646	2.828	10.461
5	10000	10000	300	10	10	4.795	2.514	0.557	**0.295**	0.622
6	10000	10000	60	10	1	10.530	7.143	63.981	**1.828**	3.500
7	10000	10000	60	5	3	3.209	2.215	**0.528**	0.674	2.170
8	5000	5000	100	50	3	3.168	2.316	77.085	**1.458**	3.919
9	10000	10000	100	1	1	2.924	2.070	0.388	**0.062**	0.090

Tabel 41: Vergelijking van de run times (in seconden). Voor SNL en PSJ werden de resultaten weergegeven voor de optimale keuze van signature length en aantal klasses.

6 Conclusie

In deze masterproef werden set join operaties op database-relaties bestudeerd: division, set containment join, set disjointness join, set equality join en set overlap join. Deze operaties werden uitgedrukt in de relationele algebra en geïmplementeerd in de querytalen SQL en XQuery.

Daarna werd de focus gelegd op de set containment join. Wij hebben gezien dat de relationele algebra - uitdrukking voor deze set containment join niet effciënt is. Vandaar de nood aan efficiëntere set containment join algoritmes. Enkele van deze algoritmes werden in detail behandeld: het naïeve NL, waarbij elke tuple van R met elke tuple van S wordt vergeleken; SNL en PSJ, die steunen op signatures; INL en IFJ die gebruik maken van de inverted file.

Deze algoritmes werden geïmplementeerd in Java en getest op database-relaties met een grootte van typisch 10000 tuples. Dit geeft 10^8 combinaties van tuples, wat typisch in enkele seconden kon worden uitgevoerd.

De resultaten van deze experimenten bevestigen de tijdscomplexiteit die kon worden vermoed op basis van de werking van de algoritmes. Zo is duidelijk dat een verdubbeling van de grootte van beide databases ruwweg leidt tot een vervierdubbeling van de run time. Ook het worst-case scenario, waarbij elke combinatie ($t_R.set$,$t_S.set$) een set containment oplevert, is hierbij interessant omdat in deze extreme situatie de werking en output van de algoritmes erg duidelijk is.

Een andere conclusie is dat de parameterwaarden belangrijk zijn om te beslissen welk algoritme optimaal is. Zo blijkt INL vaak het snelste algoritme te zijn, maar dit is niet meer het geval bij vrij dichte (dense) datasets (dit is wanneer B_R en B_S relatief groot zijn ten opzichte van A). Ook PSJ en IFJ zijn voor sommige parameterwaarden soms de snelste keuze, maar hebben in andere situaties dan weer een veelvoud van de tijd van INL nodig. Wanneer deze algoritmes in een grote real-life situatie gebruikt zouden worden, zou het dus interessant zijn op voorhand het juiste algoritme te kiezen aan de hand van de aard van de database.

Wanneer de tijden van de query's in SQL en XQuery enerzijds (tabel 35) worden vergeleken met de tijden van de in Java geïmplementeerde algoritmes anderzijds (tabel 41), bemerken we enorme verschillen: SQL en XQuery zijn veel trager. Dit toont aan dat het in de praktijk bij grotere datasets erg nuttig kan zijn deze Java-implementatie te gebruiken.

Referenties

[1] Dirk Leinders en Jan Van den Bussche, *On the complexity of division and set joins in the relational algebra*, J. Comput. Syst. Sci. **73** (4), 538–549 (2007).

[2] Nikos Mamoulis, *Efficient Processing of Joins on Set-valued Attributes*, Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, 157–168 (2003).

[3] Karthikeyan Ramasamy, Jignesh M. Patel, Jeffrey F. Naughton en Raghav Kaushik, *Set Containment Joins: The Good, The Bad and The Ugly*, VLDB '00 Proceedings of the 26th International Conference on Very Large Data Bases, 351-362 (2000).

[4] Sven Helmer en Guido Moerkotte, *Evaluation of Main Memory Join Algorithms for Joins with Set Comparison Join Predicates*, VLDB '97 Proceedings of the 23rd International Conference on Very Large Data Bases, 386–395 (1997).

[5] *Decision Support System (DSS)*, `http://searchcio.techtarget.com/definition/decision-support-system`.

[6] *TPC-H*, `http://www.tpc.org/tpch/`.

[7] Hector Garcia-Molina, Jeffrey D. Ullman en Jennifer Widom, *Database Systems: The Complete Book (Second Edition)*, Pearson (2008).

[8] Raghu Ramakrishnan en Johannes Gehrke. *Database Management Systems Second Edition*, McGraw-Hill, 92–106 (2000).

[9] Adrienne Watt, *Chapter 7 The Relational Data Model*, `https://opentextbc.ca/dbdesign01/chapter/chapter-7-the-relational-data-model/`.

[10] Caleb Curry, *Atomic Values*, `https://www.calebcurry.com/atomic-values/`.

[11] Adrienne Watt, *Chapter 8 The Entity Relationship Data Model*, `https://opentextbc.ca/dbdesign01/chapter/chapter-8-entity-relationship-model/`.

[12] Larry Carter en Mark N. Wegman, *Universal Classes of Hash Functions*, J. Comput. Syst. Sci. **18** (2), 143-–154 (1979).

[13] Dmitry Shaporenkov, *Efficient Main-Memory Algorithms for Set Containment Join Using Inverted Lists*. In: J. Eder, H.M. Haav, A. Kalja, J. Penjam (eds) Advances in Databases and Information Systems. ADBIS 2005. Lecture Notes in Computer Science, vol 3631. Springer, Berlin, Heidelberg (2005).

[14] *W3schools SQL Support*, `https://www.w3schools.com/sql/`.

[15] *DB2 Express-C*, `http://db2express.com/en/`.

[16] *IBM XQuery support*, `https://www.ibm.com/support/knowledgecenter/en/SSEPGG_10.5.0/com.ibm.db2.luw.xml.doc/doc/xqrfndpe.html`.

[17] Stack Overflow, https://stackoverflow.com/questions/13979691/xquery-how-to-try-if-a-list-contains-a-given-string.

[18] Microsoft Docs, https://docs.microsoft.com/en-us/sql/xquery/functions-on-boolean-values-not-function?view=sql-server-2017.

[19] *W3schools XQuery Support*, https://www.w3schools.com/xml/xquery_intro.asp.

[20] *BaseX*, www.basex.org/.

[21] *BaseX Standalone*, http://docs.basex.org/wiki/Standalone_Mode.

[22] *DB2Batch*, https://www.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.admin.cmd.doc/doc/r0002043.html.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Set joins in database query processing

Richting: **master in de informatica-databases**
Jaar: **2018**

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,

Van Assche, Filip

Datum: **27/08/2018**