Hoofdstuk 3

Een relationele database ontwerpen door normalisatie

- De student kan voor een gegeven database bepalen in welke normaalvorm deze database staat en waarom.
- De student kan voor elke normaalvorm aangeven wat de onderhoudsproblemen zijn. Hij kan die aangeven bij een gegeven voorbeeld of zelf een voorbeeld geven.
- De student kan voor elke normaalvorm bij een gegeven database aangeven waarom deze database in deze normaalvorm staat en hij kan de database omzetten naar een hogere normaalvorm.
- De student kan bij een gegeven casus de database stap voor stap normaliseren tot derde normaalvorm. Hij kan elke stap verantwoorden door de normaalvorm te bepalen en de reden waarom.
- De student kan zijn finale oplossing weergeven in een tupelschema en in een visueel diagram (ERD of datastructuurdiagram).
- De student kan uitleggen waarom denormalisatie in sommige gevallen te verantwoorden is en hij kan hierbij een voorbeeld geven.

In de vorige hoofdstukken hebben we gezien hoe we gegevens kunnen opslaan in een relationele database. Daarbij hebben we uitvoerig kennis gemaakt met het relationele model. In dit deel zullen wij ons concentreren op het ontwerp van een relationele database. Uitgaande van de informatiebehoefte van een klant of opdrachtgever zullen wij bepalen welke relaties (tabellen) we nodig hebben en wat het tupelschema van elke relatie is. We leggen tegelijk ook de primary keys en foreign keys vast en bepalen het domein van ieder attribuut.

Het ontwerpen van een databank gebeurt typisch in een aantal stappen waarbij het belangrijk is zoveel mogelijk informatie te verzamelen over wat er verwacht wordt van de databank (de *requirements*), door de gebruiker of klant. Hieronder valt bijv.

- Welke data of informatie moet er opgeslagen worden?
- Welke rapporten moeten er kunnen gegenereerd worden op basis van de databank?
- In welke context wordt de databank gebruikt? Zal de databank telkens door een enkel pro-

gramma worden aangesproken, of zal de databank rechtstreeks aangesproken worden door duizenden individuen?

- Wat is het kennisniveau van de gebruikers? Bouw je de databank voor een doorgewinterde databank expert, of voor iemand die oppervlakkige toegang nodig heeft voor een simpele website?

- In hoeverre moet de databank future-proof zijn? Als de databank dient om een toepassing te sturen die nooit meer geupdate wordt is dit gemakkelijker dan een databank te bouwen die meerdere updates moet overleven, over de jaren heen.

Eens er bekend is *wat* er moet opgeslagen worden gaan we typisch een *normalisatieproces* doorlopen om ervoor te zorgen dat we deze opslag op een correcte manier kunnen verzorgen. Hierdoor wordt er onder andere voorkomen dat er duplicate informatie wordt opgeslagen. Het normalisatieproces kan je op twee manieren aanpakken:

- Alle attributen definiëren alsof ze in één tabel zitten en dan stap voor stap de normalisatieregels toepassen. Dit is de manier die we in deze cursus zullen toepassen. Dit is een goede manier bij kleine databases en voor onervaren ontwerpers.

- De attributen direct in entiteiten groeperen en de relaties bepalen. In een iteratief (d.w.z. zich steeds herhalend) proces ga je ontwerpfouten detecteren door de normalisatieregels toe te passen, te normaliseren en zo nieuwe tabellen te creëren. Dit proces herhaal je steeds weer tot je ontwerp juist is. Je koppelt ook geregeld terug naar je klant. Deze manier is de manier die ervaren ontwerpers in de realiteit zullen toepassen.

In het vorige hoofdstuk hebben we de begrippen gegevensintegriteit en redundantie behandeld. We hebben toen vastgesteld dat redundantie, het feit dat één informatie-element op twee plaatsen in de database opgeslagen wordt, ervoor kan zorgen dat gegevens inconsistent worden en dat dus de informatie op de twee locaties niet meer met elkaar overeenkomt. Wanneer wij bijvoorbeeld het telefoonnummer van een klant in twee verschillende tabellen gaan stockeren, bestaat het gevaar dat we het telefoonnummer in de ene tabel wijzigen, maar dat in de andere tabel vergeten te doen. We krijgen dan een inconsistente database. Wanneer we bijvoorbeeld bij een departementsnummer van een werknemer ook overal de departementsnaam gaan herhalen, dan lopen we het risico dat we ergens bij hetzelfde nummer een andere naam vermelden.

Van een tabel kunnen we bepalen in welke *normaalvorm* deze staat. De normaalvorm (of *normal form*, afgekort tot **NF**) vertelt ons iets over de mogelijke inconsistentieproblemen die er kunnen optreden bij het gebruik van die relatie. We zeggen dat een tabel *conform* is aan een bepaalde normaalvorm, of in een bepaalde normaalvorm staat. Normaalvormen bestaan in gradaties, van 1NF tot 6NF, waarbij elke hogere normaalvorm telkens een bepaald type inconsistentie verhelpt. Een hogere normaalvorm is echter moeilijker te implementeren, dus er is hier sprake van een evenwicht zoeken tussen accuraatheid en veiligheid, en gebruiksgemak.

In dit hoofdstuk zullen we leren hoe we een database kunnen ontwerpen die voldoet aan een zekere normaalvorm (3NF). We noemen dit in de praktijk een genormaliseerde database, aangezien 3NF typisch als een gezonde balans aanzien wordt tussen bruikbaarheid en correctheid. De komende secties behandelen de begrippen *atomaire attributen* en *functionele afhankelijkheden*. Daarna worden de verschillende normaalvormen besproken. Tot slot wordt de methode behandeld om stap voor stap van de nulde naar de derde normaalvorm over te gaan: het eigenlijke normalisatieproces.

3.1 Atomaire attributen

Een atomaire waarde is er één die je niet kan opsplitsen in andere waarden. Typisch beschouwen we de datatypes van een databank als atomair, zoals een getal, string of datum. Niet-atomaire waarden zijn bijvoorbeeld arrays, lijsten, sets, samengestelde objecten. Een atomair attribuut is een attribuut dat enkel atomaire waarden kan aannemen.

Merk op dat een correct gevormde *relatie* enkel atomaire waarden mag bevatten. Dit concept is belangrijk bij normalisatie, aangezien een tabel in 1NF enkel atomaire attributen mag bevatten.

Stel dat je een lijst bijhoudt van alle studenten van het tweede jaar samen met hun examenresultaten per onderwijsactiviteit, voor die onderwijsactiviteiten waarvan al een examen werd afgelegd. Een illustratie van een deel van die lijst staat in Tabel 3.1.

id	last_name	first_name	id_class	results
45612	Lauwers	Eveline	2TIA	{ "Besturingssystemen": 9, "Systeemanalyse": 13, "Datacommunicatie": 12 }
14556	Peeters	Koen	2TIA	{ "Besturingssystemen": 11, "Systeemanalyse": 12, "Datacommunicatie": 14, "Administratieve inf.": 15 }

Tabel 3.1: Illustratie niet-atomaire waarden in kolom 'resultaten'.

We zouden deze gegevens niet zomaar in één relatie kunnen zetten. Per student hebben we immers in de laatste kolom meerdere examenresultaten en attribuutwaarden zijn in een relationele database enkelvoudig. De laatste kolom met meerdere waarden voor vakken en resultaten bevat dus niet-atomaire waarden. De tabel staat dus niet in 1NF, maar in 0NF.

In het voorbeeld van Tabel 3.1 nemen de niet-atomaire waarden de vorm aan van een JSON-object: een reeks van key-value-paren. Niet-atomaire waarden kunnen nog andere vormen aannemen zoals arrays, sets en geneste structuren. Deze worden geïllustreerd in Tabel 3.2.

1NF-inbreuk	voorbeeldwaarde
array	["Besturingssystemen", "Systeemanalyse"]
set	("rood", "groen", "geel")
object	{ "Databanken": 12, "Analyse": 14 }
subtabel	Databanken | 9 Web Development | 12

Tabel 3.2: Illustratie types van niet-atomaire waarden.

Het concept van atomaire waarden is belangrijk wanneer we het hebben over de eerste normaalvorm. De definitie ervan steunt erop (zie Sectie 3.3.1).

3.2 Functionele afhankelijkheden

Het concept van functionele afhankelijkheden (afgekort tot f.a.) helpt ons in te zien hoe attributen zich onderling gedragen. Met een f.a. kunnen we voor een attribuut (of verzameling attributen) bepalen welke attributen we nodig hebben om deze uniek te bepalen. Dit is belangrijk wanneer we kandidaatsleutels willen bepalen, en wordt algemeen gebruikt in het normalisatieproces.

Definitie 1 (Functionele afhankelijkheid) *Een verzameling attributen B is **functioneel afhankelijk** van een verzameling attributen A als de waarden van B uniek bepaald worden door de waarden van A. Als je de waarden van A dus kent, ben je ondubbelzinnig zeker over de waarden van B. Dit noteren we als $A \rightarrow B$. A noemen we de **determinant** van de f.a.*

Definitie 2 (Notatie ontbrekende functionele afhankelijkheid) *Hoewel het ontbreken van een functionele afhankelijkheid simpelweg betekent dat we dit niet noteren, is het soms nuttig om hier expliciet over te kunnen zijn. Als B* **niet** ***functioneel afhankelijk*** *is van A schrijven we dit als volgt: $A \nrightarrow B$.*

Eens we voor een verzameling attributen de functionele afhankelijkheden hebben bepaald (gebaseerd op de betekenis en de constraints van deze attributen) kunnen we dus normaliseren, wat mogelijks ervoor zorgt dat een relatie verder wordt opsplitst in meerdere relaties. Het is belangrijk om te weten dat het normalisatieproces geen f.a. doet verloren gaan. Met andere woorden: na het normalisatieproces moeten alle f.a. uit de oorspronkelijke relatie nog steeds gelden en aanwezig zijn[1].

`(product_number, product_name)`

In de veronderstelling dat een product (geïdentificeerd door `product_number`) slechts één naam heeft, is `product_name` functioneel afhankelijk van `product_number`, oftewel:

`(product_number)` $\rightarrow$ `(product_name)`

`(product_number, supplier)`

In de veronderstelling dat een product (geïdentificeerd door `product_number`) door een willekeurig aantal leveranciers kan geleverd worden, is `supplier` niet functioneel afhankelijk van `product_number`.

`(product_number)` $\nrightarrow$ `(supplier)`

3.2.1 Volledige functionele afhankelijkheid

Een functionele afhankelijkheid is een relatief *breed* concept. Neem de volgende relatie als voorbeeld.

`User(id, email, first_name, last_name)`

Hierbij kan je de betekenis van de attributen in achting nemen: een `(id)` is uniek, bedoeld als identifier voor de relatie. `(email)` is ook een uniek attribuut voor een persoon. De volgende opmerkingen illustreren dan wat een f.a. al dan niet is:

- **`(id)`** $\rightarrow$ **`(email)`**
  Wanneer je een id kent, weet je dat er maximaal één email aan gekoppeld kan worden. Deze f.a. geldt dus perfect.

[1] De enige uitzondering hierop gebeurt bij het oplossen van 1NF-inbreuken (dus niet-atomaire attributen omzetten naar atomaire attributen), waar attributen van betekenis kunnen veranderen waardoor de f.a. ook wijzigen (zie Sectie 3.4.1).

- **`(first_name)` $\nrightarrow$ `(last_name)`**
  Deze f.a. geldt *niet*. Wanneer je een voornaam kent, kunnen er immers meerdere achternamen mee geassocieerd worden in de context van de relatie *User*.

- **`(id, first_name)` $\rightarrow$ `(last_name)`**
  Deze f.a. geldt *wel*. Als je immers beide `(id)` én `(first_name)` kent, dan kan daar maar één achternaam mee geassocieerd worden. Achternaam is dus uniek bepaald door id en voornaam.

- **`(id)` $\rightarrow$ `(first_name, last_name, email)`**
  Als een verzameling attributen meerdere attributen uniek bepaalt noteren we dit gewoonweg op deze bondige manier, in plaats van ze apart uit te schrijven.

In de bovenstaande voorbeelden staat verstopt wat dan een *volledige* functionele afhankelijkheid is: een f.a. die een *minimale determinant* heeft. Met andere woorden, de verzameling attributen A in $A \rightarrow B$ kan niet meer verkleind worden door er een attribuut uit te halen, zonder de f.a. ongeldig te maken.

Definitie 3 (Volledige functionele afhankelijkheid) *Een verzameling attributen B is* **volledig *functioneel afhankelijk*** *van een verzameling attributen A als B functioneel afhankelijk is van A en bovendien geldt dat B niet functioneel afhankelijk is van een deel van A.*

Als $A \rightarrow B$ geldt, maar B is niet volledig f.a. van A, dan zeggen we dat B **partieel functioneel afhankelijk** is van A. Dit concept is belangrijk in de context van 2NF (zie Sectie 3.3.2).

In de bovenstaande voorbeelden, gebaseerd op de relatie *User*, kan je het volgende stellen:

- **`(id)` $\rightarrow$ `(email)`**
  Dit is een *volledige* f.a., aangezien `(id)` een verzameling van één attribuut is. Dat kan je niet verkleinen.

- **`(id, first_name)` $\rightarrow$ `(last_name)`**
  Dit is een *partiële* f.a., geen volledige, aangezien je `(first_name)` kan weghalen uit de determinant zonder de functionele afhankelijkheid te schaden. `(id)` $\rightarrow$ `(last_name)` geldt immers ook.

3.2.2 Volledige transitieve functionele afhankelijkheid

Wanneer we een situatie hebben waarin er attributen functioneel afhankelijk zijn van een niet-sleutelattribuut kan het zijn dat we te maken hebben met een *transitieve* functionele afhankelijkheid. De interessantste vorm hiervan is een *volledige* transitieve functionele afhankelijkheid. Dit type van f.a. is belangrijk in het normaliseren naar de derde normaalvorm (zie Sectie 3.3.3). Aangezien een niet-volledige transitieve functionele afhankelijkheid vaak niet interessant is voor onze doeleinden (normaliseren) zullen we vanaf nu **altijd** een *volledige transitieve f.a.* bedoelen wanneer we het hebben over een *transitieve f.a.*

Definitie 4 (Volledige transitieve functionele afhankelijkheid) *Een volledige functionele afhankelijkheid $A \rightarrow C$ is* ***transitief*** *wanneer er een B bestaat waarvoor geldt dat $A \rightarrow B$ en $B \rightarrow C$.*
Hierbij zijn A, B en C verzamelingen attributen. B mag zelf ***geen*** *kandidaatsleutel zijn*[2].

Zulk een transitieve f.a. kan je dus vaak simpel identificeren doordat er een functionele afhankelijkheid bestaat van een attribuut op een niet-sleutelattribuut.

[2] Is B wel een kandidaatsleutel, dan geldt $B \rightarrow A$.

In dit voorbeeld gebruiken we de relatie *Order*.

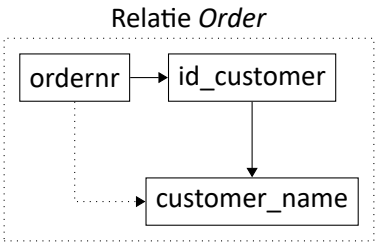
`Order(id_order, id_customer, customer_name)`

In de veronderstelling dat een order toebehoort aan een welbepaalde klant, dat een klant een willekeurig aantal orders kan hebben geplaatst en met een klant (geïdentificeerd door zijn `id_customer`) één enkele `customer_name` overeenstemt, geldt dat:

`(id_order)` $\rightarrow$ `(id_customer)`
`(id_customer)` $\rightarrow$ `(customer_name)`

`(id_customer)` is geen kandidaatsleutel, waardoor `(customer_name)` — via `(id_customer)` — volledig transitief afhankelijk is van `(id_order)`.

Transitieve functionele afhankelijkheden vermelden (of tekenen) we vaak niet omdat deze *vervat* zitten in andere functionele afhankelijkheden. Dit kan je visueel zien in Figuur 3.1 uit het bovenstaande voorbeeld, waarin de f.a. op de kandidaatsleutel wel gevisualiseerd wordt.



Figuur 3.1: Illustratie transitieve functionele afhankelijkheid.

Om te illustreren waarom het belangrijk is dat het tussenattribuut B geen kandidaatsleutel is in deze definitie kunnen we de relatie `Person` als voorbeeld nemen:

`Person (id, email, name)`

Deze relatie heeft twee kandidaatsleutels: `(id)` en `(email)`. De volgende volledige functionele afhankelijkheden gelden:

`(id)` $\rightarrow$ `(email)`
`(email)` $\rightarrow$ `(id)`
`(id)` $\rightarrow$ `(name)`
`(email)` $\rightarrow$ `(name)`

Aangezien `(id)` $\rightarrow$`(email)` $\rightarrow$`(name)` geldt zou je kunnen denken dat `(id)` $\rightarrow$`(name)` een transitieve f.a. is. Dit is echter niet het geval, aangezien `(email)` een kandidaatsleutel is!

3.3 Normaalvormen

Nu we de concepten van functionele afhankelijkheden en kandidaatsleutels kennen, kunnen we starten met het effectieve normalisatieproces. Zoals vermeld in de introductie normaliseren we relaties om bepaalde onderhoudsproblemen te voorkomen of te elimineren. Er zijn goed gedefinieerde *normaalvormen* (of *normal form (NF)*), gaande van de eerste normaalvorm (1NF) tot de zesde normaalvorm (6NF). Elke normaalvorm beschrijft een aantal regels waaraan een relatie

kan voldoen. Een *hogere* normaalvorm is telkens strenger ten opzichte van de *lagere*, en elimineert steeds een extra soort onderhoudsprobleem. In deze cursus, wat typisch ook in de echte wereld gebeurt, stoppen we het normalisatieproces bij 3NF. Dit vormt een goed evenwicht tussen accuraatheid en bruikbaarheid.

3.3.1 1NF

De eerste normaalvorm vormt de basis van het normalisatieproces. Informatie in tabelvorm is, als het geen niet-atomaire waarden bevat, een volwaardige relatie in 1NF.

Definitie 5 (1NF) *Een tabel staat in 1NF als er enkel atomaire attributen in voorkomen.*

Typisch starten we ons normalisatieproces met een 1NF relatie, aangezien we bij het ontwerpproces kunnen kiezen om geen niet-atomaire attributen toe te laten. Soms worden we echter geconfronteerd met een bestaande situatie waar er *wel* niet-atomaire attributen voorkomen, dus het is belangrijk om te weten hoe we hier dan mee om kunnen gaan.

Beschouw de relatie `Hobbies`, waarvoor voorbeelddata getoond wordt in Tabel 3.3. In dit voorbeeld bevat de kolom `(hobbies)` niet-atomaire waarden. De tabel staat dus niet in 1NF: `(hobbies)` is een niet-atomair attribuut.

email	hobbies
jan@provider.be	["zwemmen", "lopen"]
jean@fournisseur.fr	["nager"]

Tabel 3.3: 1NF voorbeelddata: relatie *Hobbies*

Onderhoudsprobleem

De eerste normaalvorm zorgt ervoor dat we een basistabel hebben die voldoet aan de relationele algebra. De tabel is dus een *relatie* waar enkel atomaire waarden in voorkomen en die steeds goed leesbaar is, zonder nog een waarde te moeten interpreteren.

Stel dat je een attribuut hebt dat een JSON-object als waarde kan hebben. Dat zou er zo kunnen uitzien:

```
{
  "name": "Jef",
  "birthdate": "12/12/2012",
  "hobbies": ["swimming", "running"]
}
```

Het uitlezen van zulk een waarde (en dus het attribuut) is niet evident. Je moet weten wat de structuur is, of alle waarden dezelfde structuur respecteren, wat voor datatypes er in het JSON-object verstopt zitten, of er delen kunnen ontbreken, enzovoort. Door te normaliseren naar 1NF zorgen we ervoor dat deze extra stappen en nood tot naverwerking

vermeden worden: de waarde van een attribuut is steeds eenduidig uit te lezen.

3.3.2 2NF

De tweede normaalvorm steunt op twee concepten: partiële functionele afhankelijkheden en kandidaatsleutels. 2NF voorkomt voornamelijk dat er een vorm van duplicatie van data optreedt, wat zijn voordelen heeft wanneer er aanpassingen in de relaties worden gedaan.

Definitie 6 (2NF) *Een relatie R staat in 2NF als deze in 1NF staat en wanneer geen enkel niet-sleutelattribuut partieel functioneel afhankelijk is van* een *kandidaatsleutel.*

Merk op dat, wanneer er meerdere kandidaatsleutels aanwezig zijn in een relatie, deze allemaal moeten gecheckt worden of er attributen partieel functioneel afhankelijk van zijn. Merk ook op dat een relatie enkel een 2NF-inbreuk kan hebben wanneer er sprake is van een *samengestelde kandidaatsleutel*, aangezien er enkel dan een partiële functionele afhankelijkheid kan zijn.

Om te checken of een relatie in 2NF staat moeten we weten 1) wat de kandidaatsleutels zijn van de relatie 2) wat de functionele afhankelijkheden zijn die gelden tussen de attributen van de relatie.

Minimaal voorbeeld van 2NF-inbreuk

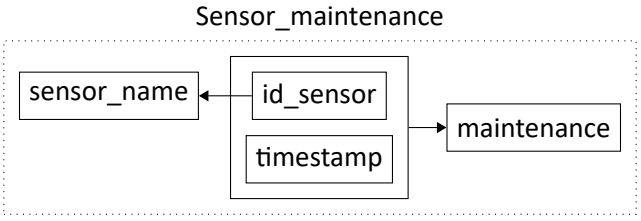
Als een minimaal voorbeeld kunnen we de relatie *Sensor_maintenance* beschouwen, die beschrijft wanneer we voor een sensor onderhoud gedaan hebben. De relatie bevat het unieke id van de beschouwde sensor, de naam van de sensor, het tijdstip van het onderhoud en een klein tekstje dat het onderhoud beschrijft.

```
Sensor_maintenance (id_sensor, sensor_name, timestamp, maintenance)
```

We bepalen eerst al de volledige functionele afhankelijkheden van *Sensor_maintenance*, die vertellen ons alles dat we nodig hebben om de kandidaatsleutels en de normaalvorm te bepalen van de relatie.

```
(id_sensor, timestamp) → (maintenance)
(id_sensor) → (sensor_name)
```

Uit deze f.a. (zie ook Figuur 3.2) kunnen we de kandidaatsleutel(s) afleiden: de minimale verzameling attributen die we nodig hebben om alle attributen van de relatie te bepalen. In dit geval is er maar één kandidaatsleutel: `(id_sensor, timestamp)`, aangezien we hiermee beide `(maintenance)` en `(sensor_name)` kunnen bepalen (en de kandidaatsleutel bepaalt uiteraard zichzelf).



Figuur 3.2: Functioneel afhankelijkheidsdiagram “Minimaal voorbeeld 2NF-inbreuk”

Nu kunnen we ook gemakkelijk zien dat er een partiële f.a. is:

```
(id_sensor, timestamp) → (sensor_name)
```

`(sensor_name)` wordt immers uniek bepaald door `(id_sensor)`, wat een deel is van een kandidaatsleutel. Hierdoor weten we dat de gegeven relatie *niet* in 2NF staat. De bovenstaande partiële f.a. is een *2NF-inbreuk*. Hoe we de relatie opwaarderen van 1NF naar 2NF zien we later.

Complexer voorbeeld van 2NF-inbreuk

Aangezien 2NF steunt op het concept van kandidaatsleutels is het ook belangrijk om te zien wat er gebeurt wanneer er verschillende kandidaatsleutels in de relatie voorkomen. Om dit te illustreren maken we een relatie om notities bij te houden. Een persoon heeft een uniek id en een email, en kan notities maken die gesorteerd zijn op tijd. Een persoon kan geen twee notities maken op hetzelfde moment. Het systeem houdt ook bij of een email *geverifieerd* is.

`Notitie(id_person, timestamp, email, verified, note)`

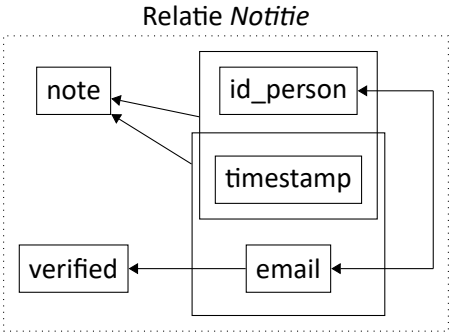
De volgende functionele afhankelijkheden gelden in deze relatie:

```
(email, timestamp) → (note)
(id_person, timestamp) → (note)
(email) → (verified)
(id_person) → (email)
(email) → (id_person)
```

De kandidaatsleutels zijn gemakkelijker te zien in de visuele representatie (zie Figuur 3.3). Elke kandidaatsleutel is een verzameling attributen waarmee je alle attributen in de relatie kan bereiken via een functionele afhankelijkheid. Aangezien `(email)` en `(id_persoon)` van elkaar afhankelijk zijn krijgen we twee kandidaatsleutels:

```
(id_person, timestamp)
(email, timestamp)
```

Een 2NF-inbreuk moet gecheckt worden op *beide* kandidaatsleutels: in dit geval is de f.a. `(email)` → `(verified)` wel degelijk een partiële f.a. van een kandidaatsleutel, en dus een 2NF-inbreuk.



Figuur 3.3: Functioneel afhankelijkheidsdiagram “Complexer voorbeeld 2NF-inbreuk”

Onderhoudsprobleem

Om te illustreren wat een mogelijk onderhoudsprobleem kan zijn in een relatie die wel in 1NF, maar niet in 2NF staat, kunnen we Tabel 3.4 met voorbeelddata bekijken uit de *Sensor_maintenance* relatie. Daar is duidelijk dat er duplicate data (redundantie) wordt opgeslagen: alle sensornamen worden telkens herhaald voor elke onderhoudsbeurt, wat ervoor zorgt dat er bij updates mogelijk inconsistenties opduiken. Als je bijvoorbeeld in één rij de sensornaam aanpast dan krijg je tegenstrijdige informatie: welke rij bevat dan de waarheid?

Bijkomstig zijn er nog problemen die opduiken wanneer je waarden wil toevoegen of verwijderen. Als je een sensor wil toevoegen zonder dat je er al een onderhoud voor doet introduceer je null values in tijdstip, wat niet mag aangezien dat in een kandidaatsleutel zit. Wanneer je overigens het enige onderhoud van een bepaalde sensor verwijdert uit de tabel ga je ook ineens alle informatie over die sensor kwijtspelen.

id_sensor	sensor_name	timestamp	maintenance
1	temperatuur living	6/3/2020 09:00	calibratie
1	temperatuur living	20/5/2020 11:00	nogmaals calibratie
1	temperatuur living	1/7/2020 08:30	kabel vervangen
2	ketel druk	20/5/2020 11:10	nakijken waarde
~~3~~	~~licht buiten~~	~~null~~	~~null~~

Tabel 3.4: Voorbeelddata voor de *Sensor_maintenance* relatie

3.3.3 3NF

De derde normaalvorm bouwt verder op de tweede en voegt hierbij een extra restrictie toe. Ze is dus strenger en voorkomt wederom een mogelijk onderhoudsprobleem.

Definitie 7 (3NF) *Een relatie R staat in 3NF als ze in 2NF staat en wanneer er geen niet-sleutelattribuut A volledig transitief functioneel afhankelijk is van een kandidaatsleutel van R.*

Met andere woorden, wanneer we een 3NF-inbreuk zoeken, dan zoeken we de volgende situatie:

A $\rightarrow$ B $\rightarrow$ C

Hierbij geldt dat A een kandidaatsleutel is, B en C verzamelingen van niet-sleutelattributen zijn en de f.a. allemaal *volledig* zijn. Daardoor geldt dus ook dat:

B $\nrightarrow$ A

Anders zou immers B een kandidaatsleutel zijn.

Minimaal voorbeeld van 3NF-inbreuk

Als een minimaal voorbeeld kunnen we de relatie *Course* beschouwen, die kortweg beschrijft voor een cursus wie de titularis ervan is (in dit geval is er maar één titularis per cursus). De geboortedag van de titularis wordt ook opgeslagen.

`Course (course, id_teacher, birthdate)`

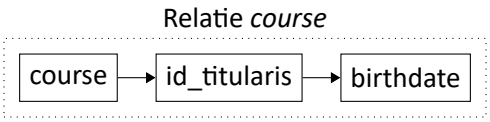
We bepalen eerst al de volledige functionele afhankelijkheden van *Course*. Die vertellen ons alles wat we nodig hebben om de kandidaatsleutels en de normaalvorm te bepalen van de relatie.

`(course)` $\rightarrow$ `(id_teacher)`
`(id_teacher)` $\rightarrow$ `(birthdate)`

Deze[3] staan visueel vermeld in Figuur 3.4. Hier is duidelijk te zien dat er één enkelvoudige kandidaatsleutel is: `(course)`.

Echter, `(birthdate)` is f.a. van `(id_titularis)` en dat attribuut is op zijn beurt f.a. van `(course)`. Deze keten van afhankelijkheden voldoet aan de eigenschappen van een volledig transitief functionele afhankelijkheid, waardoor het een 3NF-inbreuk is: *Course* staat in 2NF (wegens enkelvoudige kandidaatsleutel), maar niet in 3NF.

[3] Eigenlijk is `(course)` $\rightarrow$`(birthdate)` ook een volledige f.a. maar aangezien deze impliciet reeds aanwezig is door de vermelde afhankelijkheden schrijven we deze niet op.



Figuur 3.4: Functioneel afhankelijkheidsdiagram in “Minimaal voorbeeld 3NF-inbreuk”

Onderhoudsprobleem

De relatie *Course* is voorzien van voorbeelddata, terug te vinden in Tabel 3.5. Hier is duidelijk gemaakt wat mogelijke onderhoudsproblemen zijn in een relatie die in 2NF staat, maar niet in 3NF.

- Duplicate informatie (redundantie): Frank’s geboorte moet tweemaal opgeslagen worden. Anders zou je op basis van een rij waar de geboorte niet in staat kunnen besluiten dat Frank’s geboorte onbekend is. Dit kan voor problemen zorgen wanneer we Frank zijn geboorte willen aanpassen, het moet dan overal gedaan worden, om inconsistenties te vermijden.
- Opslagrestricties: doordat vak- en titularisinformatie samen in één relatie staan, en dat *course* de primary key is, kunnen we geen titularis toevoegen zonder deze ineens aan een vak te koppelen. Dit kan soms wenselijk zijn, maar is vaak onbedoeld.

course	**id_titularis**	**birthdate**
Analyse 1	frank1	06/03/1985
Analyse 2	frank1	06/03/1985
Databanken	jos2	12/02/1986
~~Programmeren~~	~~null~~	~~null~~
~~null~~	~~jef1~~	~~10/10/1990~~

Tabel 3.5: Illustratie 3NF onderhoudsproblemen in relatie *Course*

3.4 Normaliseren

Nu we weten wat kandidaatsleutels, functionele afhankelijkheden en normaalvormen zijn kunnen we *normaliseren*: een relatie die in een bepaalde normaalvorm staat omzetten in een hogere normaalvorm. Typisch gaan we elke relatie in 3NF omzetten, wat een goede trade-off tussen bruikbaarheid en accuraatheid bereikt. Dit is in de echte wereld meestal ook waar een grens getrokken wordt. Voor een relatie R doorlopen we steeds het volgende proces:

- Bepaal de functionele afhankelijkheden van R.
- Bepaal de kandidaatsleutels van R, op basis van de f.a.
- Bepaal de huidige normaalvorm.
  - Als we reeds in 3NF staan kunnen we stoppen: R is *genormaliseerd* (genoeg).
  - Als we nog niet in 3NF staan identificeren we een inbreuk op 3NF (dit kan een 1NF, 2NF of 3NF-inbreuk zijn).
- Splits de 3NF-inbreuk in R af naar een nieuwe relatie, waardoor R vervangen wordt door twee relaties S en T.

- Herhaal het normalisatieproces voor S en T: deze moeten beide in 3NF terecht komen.

Na het normalisatieproces bekomen we één (wanneer R reeds in 3NF staat) of meerdere relaties $R_{1..n}$ (resultaat van de afsplitsingen) die in 3NF staan. Deze relaties zijn dan klaar om geïmplementeerd te worden in een echte databank, waarbij we zeker zijn dat alle vermelde onderhoudsproblemen vermeden worden.

Opgelet: bij kleine relaties of in speciale gevallen kan het voorvallen dat na normalisatie een relatie R_i volledig vervat zit in R_j (dus alle attributen van R_i komen voor in R_j). Dan mag je R_i weglaten, aangezien het overtollig is.

Hoewel het concept van een foreign key tijdens het normalisatieproces niet ter sprake komt is het wel belangrijk te beseffen dat, na het splitsen van een relatie in twee andere relaties, de link ertussen uiteindelijk zal kunnen afgedwongen worden door een foreign key constraint, wanneer de gekozen databank dit ondersteunt. We zullen dit in de komende voorbeelden ook vermelden.

In de volgende secties wordt uitgelegd hoe je precies de afsplitsingen uitvoert waarmee je van een lage naar een hoge normaalvorm gaat.

3.4.1 1NF-inbreuk oplossen

Wanneer een relatie R een 1NF-inbreuk bevat weten we dat het gaat om een niet-atomair attribuut (of meerdere, maar dan los je deze samen op analoge manier op). Het niet-atomair attribuut A beschrijft, per definitie, meerdere dingen (of attributen). De *inbreuk* in R is effectief het attribuut A, en zolang dat attribuut in deze vorm in R blijft staan zal de relatie niet in 1NF kunnen staan. Als we dus willen normaliseren naar 1NF moeten we het attribuut verwijderen uit de relatie. Voer de volgende stappen uit:

- Bepaal de kandidaatsleutel van R.
- Vorm het niet-atomair attribuut A om in een aantal (1 of meer) atomaire attributen $B_{1..n}$. **Opgelet**, we veranderen attributen, dus deze actie past ook functionele afhankelijkheden aan.
- Maak een nieuwe relatie S aan, waar je $B_{1..n}$ naar verplaatst. A verwijder je uit R.
- Zorg dat S een correcte kandidaatsleutel heeft, waarmee je de originele relatie mee kan reconstrueren. Typisch kopieer je hiervoor de kandidaatsleutel van R over naar S. Doordat de functionele afhankelijkheden veranderd zijn moet je hier ook rekening mee houden.

Minimaal voorbeeld

We lossen de 1NF-inbreuk op in Tabel 3.3 (zie p.48). De kandidaatsleutel van deze tabel is `(email)`. Het niet-atomair attribuut is `(hobbies)`, wat dus moet verdwijnen uit de relatie om het in 1NF te krijgen. We maken een nieuwe relatie `Hobby` aan. Het niet-atomair attribuut `(hobbies)` splitsen we op zodat er enkel atomaire waarden in voorkomen. In dit geval volstaat het om het om te

vormen naar een attribuut `(hobby)`, waar er telkens één hobby als waarde in voorkomt. De originele kandidaatsleutel uit `Hobbies` kopiëren we mee naar `Hobby`, aangezien we moeten kunnen reconstrueren dat een bepaald persoon wel degelijk een zekere hobby uitoefent.

De nieuwe relatie is dan als volgt:

```
Hobby (email, hobby)
```

De nieuwe tabel Hobby ziet eruit zoals in Tabel 3.6.

email	hobby
jan@provider.be	zwemmen
jan@provider.be	lopen
jean@fournisseur.fr	nager

Tabel 3.6: Minimaal voorbeeld 1NF-oplossen: de nieuwe relatie *Hobby*

We moeten nu enkel nog de functionele afhankelijkheden en de kandidaatsleutel van `Hobby` bepalen. Aangezien we een attribuut hebben *aangepast* veranderen de originele functionele afhankelijkheden. `(email)` $\rightarrow$ `(hobbies)` geldt niet meer, aangezien `(hobbies)` is verdwenen. In onze nieuwe relatie kunnen we geen functionele afhankelijkheid meer identificeren. De kandidaatsleutel is dus effectief de verzameling van alle attributen: `(email, hobby)`.

Om het verhaal accuraat te houden moeten we ook nog het *restje* van dit proces bekijken: we hebben `(hobbies)` uit de relatie `Hobbies` gehaald, waardoor er enkel nog het attribuut `(email)` in overblijft (zie Tabel 3.7). We zitten hier dus in een speciaal geval waarin de relatie `Hobbies` een subset is van de relatie `Hobby`: `(email)` zit vervat in `(email, hobby)`. `Hobbies` brengt dus niks meer bij aan onze kennis en mogen we elimineren. Het resultaat van ons normalisatieproces is één relatie `Hobby`, die in 1NF staat.

email
jan@provider.be
jean@fournisseur.fr

Tabel 3.7: Minimaal voorbeeld 1NF-oplossen: de aangepaste relatie *Hobbies*

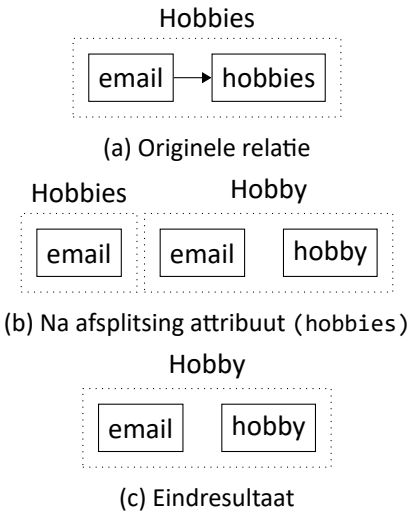
Visueel kan je het bovenstaande proces ook zien in Figuur 3.5.

3.4.2 2NF-inbreuk oplossen

Wanneer een tabel in 1NF staat maar niet in 2NF kan je de inbreuk vinden en deze afsplitsen naar een nieuwe relatie. De volgende stappen gebruik je om relatie R te splitsen in relaties S en T.

- Bepaal functionele afhankelijkheden en kandidaatsleutels van R
- Bepaal de 2NF-inbreuk (een[4] *probleemattribuut* A dat partieel afhankelijk is van kandidaatsleutel K)
- Splits A af naar een nieuwe relatie S
- Kopieer het deel van K waar A volledig functioneel afhankelijk van is over naar S (dit deel van K wordt een kandidaatsleutel van S)

[4] Dit kan ook een verzameling attributen zijn, maar dat los je analoog op.



(a) Originele relatie

(b) Na afsplitsing attribuut (hobbies)

(c) Eindresultaat

Figuur 3.5: Minimaal voorbeeld 1NF-inbreuk oplossen

Deze stappen resulteren in twee relaties die elk een eigen normaalvorm hebben. Als er nog 2NF-inbreuken zijn in wat overblijft van R kan je deze op dezelfde manier oplossen. Als er attributen zijn die op dezelfde manier partieel afhankelijk zijn van dezelfde kandidaatsleutel kan je deze uiteraard ook in één keer afsplitsen.

Minimaal voorbeeld

Bij wijze van voorbeeld gebruiken we de relatie *Sensor_maintenance* die eerder werd gebruikt om de tweede normaalvorm te illustreren (zie Tabel 3.4).

```
Sensor_maintenance (id_sensor, sensor_name, timestamp, maintenance)
```

De functionele afhankelijkheden waren hiervan:

```
(id_sensor, timestamp) → (maintenance)
(id_sensor) → (sensor_name)
```

Er was één kandidaatsleutel: `(id_sensor, timestamp)`. De 2NF-inbreuk die geïdentificeerd werd was dat `(sensor_name)` partieel f.a. is van de kandidaatsleutel doordat `(sensor_name)` functioneel afhankelijk is van `(id_sensor)`. We lossen nu deze inbreuk op door het probleemattribuut `(sensor_name)` af te splitsen naar een nieuwe relatie `Sensor`. Aangezien `(sensor_name)` f.a. is van `(sensor_id)` kopiëren we dit attribuut ook mee naar `Sensor`. Dit resulteert dan in de volgende twee relaties:

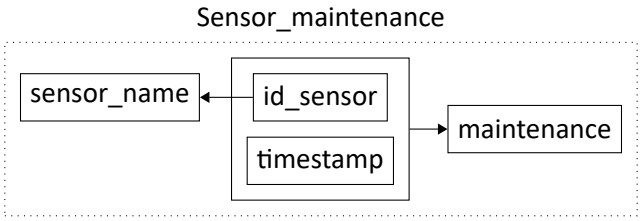
```
Sensor_maintenance (id_sensor, timestamp, maintenance)
Sensor (id_sensor, sensor_name)
```

De kandidaatsleutels zijn hier respectievelijk `(id_sensor, timestamp)` en `(id_sensor)`. Beide relaties staan in 1NF en er zijn geen 2NF-inbreuken meer te vinden in beide relaties: ze staan dus beide in 2NF. Visueel kan je dit proces ook terugvinden in Figuur 3.6.

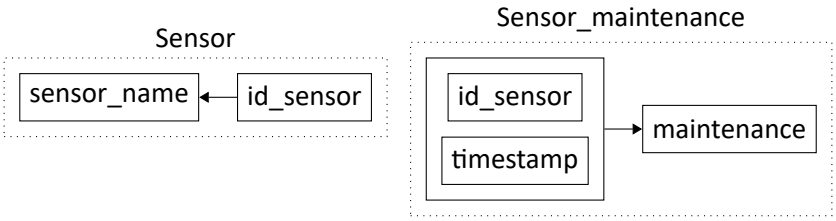
Als we de relaties van dit voorbeeld zouden implementeren in een databank leggen we indien mogelijk een foreign key constraint:

```
FK van Sensor_maintenance (id_sensor) naar Sensor(id_sensor)
```

De *richting* van de constraint is belangrijk: we verwijzen *naar* `Sensor` aangezien daar de



(a) Originele relatie



(b) Eindresultaat na afsplitsing attribuut sensor_name

Figuur 3.6: Minimaal voorbeeld 2NF-inbreuk oplossen

sensor broninformatie staat.

3.4.3 3NF-inbreuk oplossen

De derde normaalvorm steunt op de definitie van de tweede normaalvorm en het concept van transitieve functionele afhankelijkheden. Neem aan dat relatie R reeds in 2NF staat. Als we een niet-sleutelattribuut hebben gevonden dat (volledig) transitief functioneel afhankelijk is van een kandidaatsleutel, is dat een *probleemattribuut*. Zolang dat attribuut in R blijft staan, kan R niet conform zijn aan 3NF. Analoog aan hoe we een relatie van 1NF naar 2NF brengen gaan we dat probleemattribuut afsplitsen naar een nieuwe relatie.

Als we de functionele afhankelijkheden kennen van R is het probleemattribuut gemakkelijk te herkennen. Stel dat we de volgende f.a. hebben, met A een kandidaatsleutel[5]:

A $\rightarrow$ B
B $\rightarrow$ C

Het attribuut (of verzameling attributen) C is dan het probleem dat we moeten elimineren uit de relatie om conform te zijn aan 3NF. Laten we hier B het *tussenattribuut* noemen van deze transitieve functionele afhankelijkheid. De stappen die we ondernemen bij het oplossen van een 3NF-inbreuk zijn de volgende:

- Bepaal functionele afhankelijkheden en kandidaatsleutels van de relatie R (die reeds in 2NF staat).
- Bepaal de 3NF-inbreuk.
- Splits het probleemattribuut af naar een nieuwe relatie S.
- Kopieer het *tussenattribuut* waar A volledig functioneel afhankelijk van is over naar S (dit wordt een kandidaatsleutel van S).

[5] Merk op dat hier $A \rightarrow B$ enkel kan wanneer A een volledige kandidaatsleutel is. We gaan immers ervan uit dat de relatie reeds in 2NF staat.

Minimaal voorbeeld

Neem de voorbeeldrelatie *Course* om te illustreren hoe we een 3NF-inbreuk oplossen (zie Figuur 3.4 op p.52). De relatie zag er als volgt uit:

```
Course (course, teacher, birthdate)
```

De volledige functionele afhankelijkheden van *Course* zijn de volgende:

```
(course) → (teacher)
(teacher) → (birthdate)
```

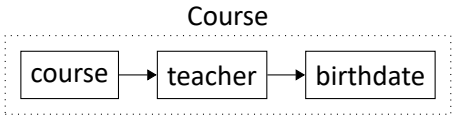
Course heeft één kandidaatsleutel: `(course)`. We zien dat er een situatie bestaat die voldoet aan een 3NF-inbreuk: een niet-sleutelattribuut dat transitief f.a. is van een kandidaatsleutel, via het niet-sleutelattribuut `(teacher)`.

```
(course) → (teacher) → (birthdate)
```

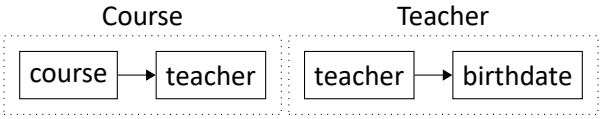
De 3NF-inbreuk is hier dus `(birthdate)`: dit attribuut moet verdwijnen uit de relatie vooraleer ze in 3NF kan staan. We splitsen het attribuut dus af naar een nieuwe relatie en kopiëren het attribuut waarvan het direct functioneel afhankelijk van was. We bekomen de volgende twee relaties:

```
Course (course, teacher)
Teacher (teacher, birthdate)
```

Beide relaties staan in 3NF. Deze procedure kan je ook visueel volgen in Figuur 3.7



(a) Originele relatie



(b) Eindresultaat na afsplitsing attribuut 'birthdate'

Figuur 3.7: Minimaal voorbeeld 3NF-inbreuk oplossen

3.4.4 Compleet voorbeeld

Stel dat we een opdracht krijgen om een databank te maken die voor werknemers bijhoudt in welk department ze werken en voor welke projecten ze allemaal gewerkt hebben. We krijgen de gevraagde requirements en bouwen de volgende relatie op:

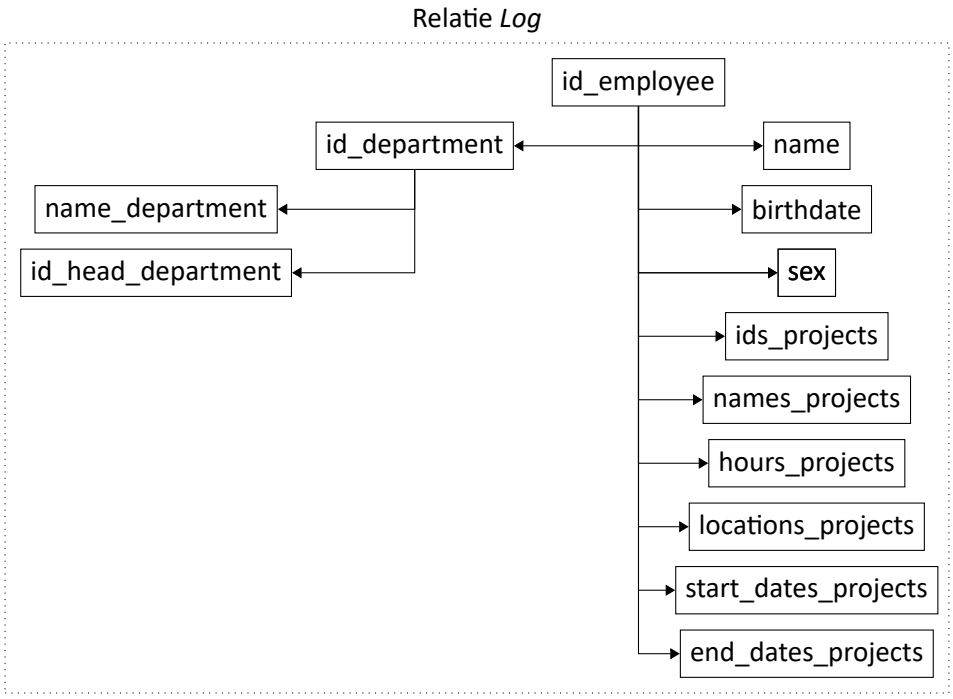
```
Log (id_employee, name, birthdate, sex, id_department, name_department,
    id_head_department, id_supervisor, ids_projects, hours_projects,
    names_projects, locations_projects, start_dates_projects,
    end_dates_projects)
```

Relatie *Log* normaliseren: voorbereiding

De eerste stap die we moeten ondernemen is een overzicht maken van de functionele afhankelijkheden. Dat gaat ons in staat stellen om correct de kandidaatsleutels te bepalen, en om in te zien wat de betekenis van de attributen exact is.

```
(id_employee) → (name, birthdate, sex, ids_projects, hours_projects,
    names_projects, locations_projects, start_dates_projects,
    end_dates_projects, id_department)
(id_department) → (name_department, id_head_department)
```

Bovenstaande functionele afhankelijkheden kan je ook visueel terugvinden in Figuur 3.8. Hier is duidelijk te zien dat er één kandidaatsleutel is: `(id_employee)`. Dit attribuut bepaalt immers alle andere attributen op unieke wijze.



Figuur 3.8: Beginsituatie voor relatie *Log*

Relatie *Log* normaliseren

De relatie *Log* staat niet in 1NF. Er zijn een aantal niet-atomaire attributen te bespeuren: alles dat project gerelateerd is bevat *meerdere waarden*. We normaliseren *Log* dus naar 1NF door deze probleemattributen af te splitsen. Aangezien ze allemaal projectgerelateerd zijn houdt het steek dit in één keer te doen en de relatie *Project* te noemen.

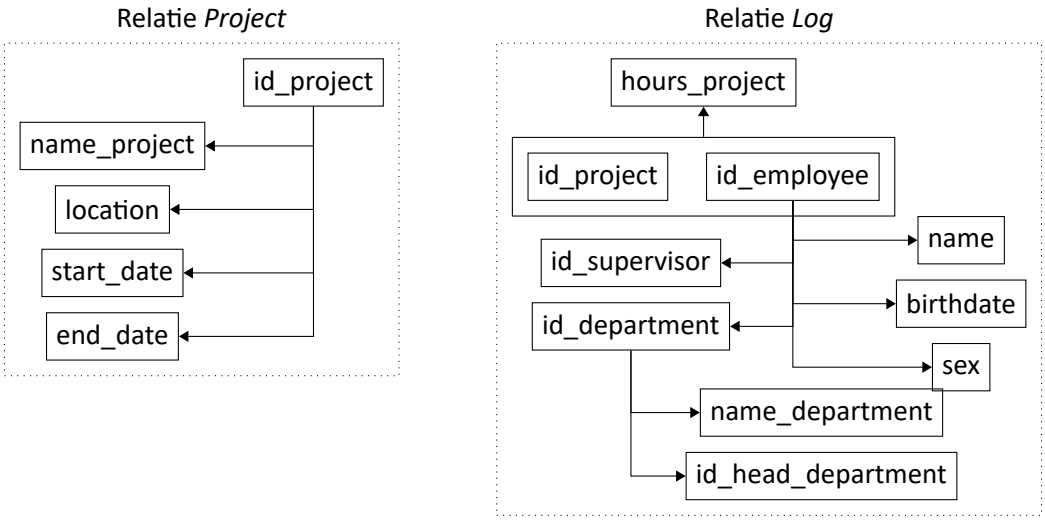
Er is één attribuut waar we extra aandacht aan moeten geven: `(hours_projects)`. Wanneer we dit enkelvoudig maken merken we op dat het attribuut afhankelijk wordt van de combinatie van `(id_project)`en `(id_employee)`. Dit blijft dus enkelvoudig staan in de relatie *Log*[6]. Moesten we het attribuut mee verhuizen naar `Project` zouden we de link met de werknemer immers verliezen.

Vanaf nu hebben we wel degelijk atomaire attributen en komen we niet meer aan hun betekenis. Alle komende normalisatiestappen zullen dus kunnen uitgaan van 1NF relaties. De uit deze normalisatiestap resulterende relaties en de functionele afhankelijkheden die erin gelden kan je zien in Figuur 3.9.

```
(id_employee) → (name, birthdate, sex, id_department, name_department,
    id_head_department, id_supervisor)
```

[6]Deze aanpassingen in de functionele afhankelijkheden zijn nodig omdat we de betekenis van de attributen moeten veranderen: ze worden enkelvoudig in plaats van meervoudig. De normalisatiestappen in 2NF en 3NF zijn aanzienlijk eenvoudiger op dat gebied.

```
(id_department) → (name_department, id_head_department)
(id_project) → (name_project, locatie, start_date, end_date)
```



Figuur 3.9: Situatie na normalisatie tot 1NF

De nieuwe versie van *Log* (te zien in Figuur 3.9) staat nog niet in 2NF. Na de 1NF normalisatiestap hebben we de attributen gewijzigd, dus de kandidaatsleutel is ook aangepast:

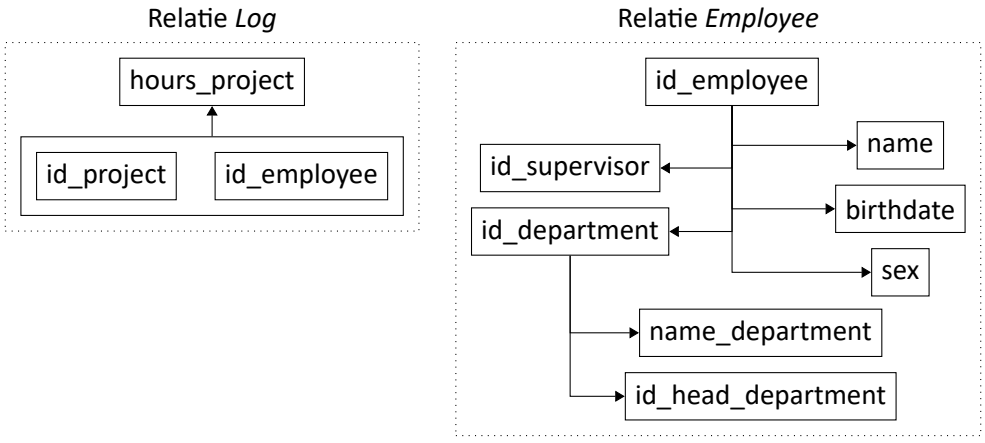
```
(id_employee, id_project)
```

Er zijn partiële f.a. te vinden:

```
(id_employee) → (name, birthdate, sex, id_department, name_department,
    id_head_department, id_supervisor)
```

Deze werknemerspecifieke attributen zijn functioneel afhankelijk van een deel van de enige kandidaatsleutel, en moeten dus verdwijnen op de relatie in 2NF te brengen. We splitsen ze af naar een nieuwe relatie *Employee*. Het resultaat van deze normalisatiestap kan je zien in Figuur 3.10. De kandidaatsleutel van *Log* blijft `(id_project, id_werknemer)` en de kandidaatsleutel van *Employee* wordt `(id_werknemer)`.

Wat resteert van de relatie *Log* (zie Figuur 3.10) staat nu ook in 3NF. Er is geen partiële f.a. of transitieve f.a. meer te bekennen.



Figuur 3.10: Situatie na normalisatie tot 2NF

Relatie *Project* normaliseren

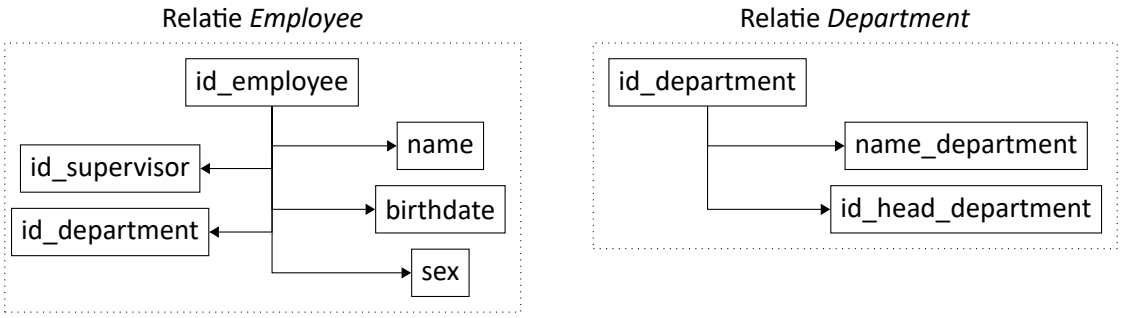
De relatie project, zoals je ze kan zien in Figuur 3.9, staat reeds in 3NF. De kandidaatsleutel ervan is `(id_project)`, dus enkelvoudig, wat geen partiële functionele afhankelijkheden tot gevolg kan hebben. Er zijn ook geen afhankelijkheden van niet-sleutelattributen, dus er kan geen 3NF-inbreuk zijn. Met deze relatie zijn we klaar!

Relatie *Employee* normaliseren

De relatie *Employee*, te zien in Figuur 3.10, staat in 2NF. De kandidaatsleutel is een enkelvoudige: `(id_werknemer)`. De relatie staat echter niet in 3NF. Er is een transitieve f.a. van de departementsattributen, via `(id_department)`, van de kandidaatsleutel. Deze moeten we dus afsplitsen naar een nieuwe relatie. We kopiëren het tussenattribuut `(id_department)` mee.

```
Employee (id_employee, name, birthdate, sex, id_department, id_supervisor)
Department (id_department, id_head_department, name_department)
```

De resulterende relaties worden ook visueel voorgesteld, met hun functionele afhankelijkheden, in Figuur 3.11. De relatie *Employee* staat nu wel in 3NF, analoog aan de relatie *Project*.



Figuur 3.11: Situatie na normalisatie tot 3NF

Relatie *Department* normaliseren

Analoog aan de relatie *Project* staat deze relatie in 3NF.

ERD diagram

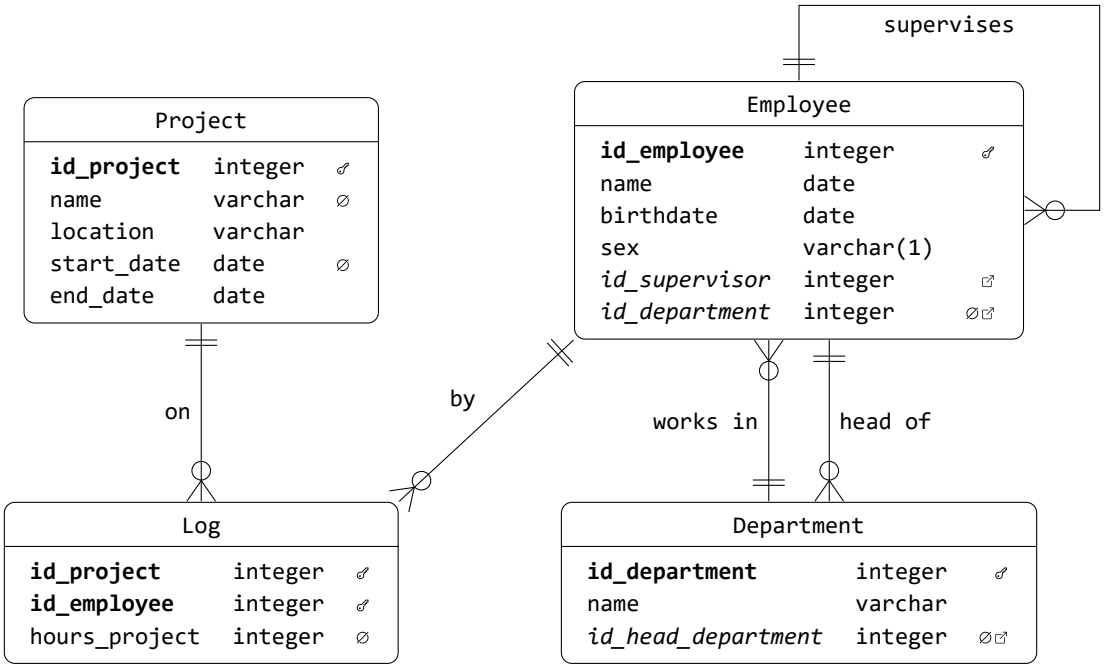
Om de resulterende relaties in een databank te implementeren helpt het om een ERD-diagram op te stellen waarin de attributen datatypes krijgen en waarin foreign key constraint worden opgesteld. Dit kan je vinden in Figuur 3.12.

3.5 Oefeningen op normaliseren

3.5.1 Inleidende oefeningen

Bestudeer onderstaande voorbeelden en beantwoord telkens volgende vragen:

- In welke normaalvorm staat dit voorbeeld?



Figuur 3.12: Datastructuurdiagram van het voorbeeld.

- Waarom?
- Zet om naar een hogere normaalvorm?
- Teken het datastructuurdiagram.

Sensoren

Tabel 3.8 illustreert het opslagen van metingen van sensoren. Latitude (lat) en longitude (lon) staan voor respectievelijk breedtegraad en lengtegraad van de locatie. Het geïmplementeerde tupelschema is het volgende:

```
Sensor_measurements(id, lat, lon, name, timestamp, temperature)
```

id_sensor	lat	lon	name	timestamp	temperature
201345	50.8102	3.2771	XPO	12-10-2016 9:02:17	4,3
				12-10-2016 9:03:52	5,2
				12-10-2016 10:02:25	5,2
210343	50.8162	3.2827	Mimosa	12-10-2016 9:03:23	4,4
				12-10-2016 10:02:25	5,6
				12-10-2016 12:00:02	8,2

Tabel 3.8: Voorbeelddata sensor en metingen.

Statuswijzigingen machine

In deze database worden de statuswijzigingen van machines opgeslagen. Machines kunnen overgaan van een status stop naar opwarmen, van opwarmen naar bezig, van bezig naar pauze, van

pauze naar bezig, van bezig naar stop en van pauze naar stop. Alle statuswijzingen van elke machine worden bijgehouden, met het tijdstip waarop de wijziging plaatsvond. Voorbeelddata kan je vinden in Tabel 3.9. Het geïmplementeerde tupelschema is het volgende:

```
Status_machine(id_machine, name, description, timestamp, status_start,
    status_end)
```

id_machine	name	description	timestamp	status_start	status_end
AX983	Zortrax M200	3D-printer Zortrax klein	18-10-2016 9:02:45	Stop	Opwarmen
LS214	Trotec Laser	Laser Cutter	18-10-2016 9:03:23	Bezig	Pauze
LS214	Trotec Laser	Laser Cutter	18-10-2016 9:08:34	Pauze	Bezig
AX983	Zortrax M200	3D-printer Zortrax klein	18-10-2016 9:10:03	Opwarmen	Printing
LS214	Trotec Laser	Laser Cutter	18-10-2016 9:23:03	Bezig	Stop
AX983	Zortrax M200	3D-printer Zortrax klein	18-10-2016 9:48:02	Bezig	Stop

Tabel 3.9: Voorbeelddata statuswijzigingen machine.

Titels en auteurs

We zien hier (voorbeelddata in Tabel 3.10) een lijst van titels. Elke titel kan meerdere auteurs hebben, in een verschillende rol. Om niet-atomaire attributen te vermijden werden de auteurs in een aparte tabel gezet: `auteurs_titels`. Het geïmplementeerde tupelschema is het volgende:

```
Book(id_book, isbn, title, publisher, year)
Book_author(id_book, id_author, author_name, author_role)
```

id_book	isbn	title	publisher	year
04321	2070386708	Caligula	Gallimard	1993
90893	0141022280	Caligula	Pocket Penguin	2005
90234	0520248953	Caligula: A Biography	University of California Press	2011

titels

id_book	id_author	author_name	author_role
04321	10345	Albert Camus	auteur
04321	15436	Pierre-Louis Rey	editor
90893	32455	Robert Graves	auteur
90234	12543	Aloys Winterling	auteur
90234	54321	Deborah Lucas	vertaler
90234	32165	Glenn W. Most	vertaler
90234	11265	Paul Psoinos	vertaler

`auteurs_titels`

Tabel 3.10: Voorbeelddata titels en auteurs.

Lezers en gelezen boeken

We hebben een lijst van lezers en de boeken die ze gelezen hebben. Wanneer hij begint met het boek te lezen, wordt de begindatum geregistreerd. Wanneer hij het boek heeft uitgelezen, wordt de einddatum geregistreerd. De lezer kan een beoordeling op vijf geven.

De tabel `Book_author` verwijst nog naar een vorige oefening. Een titel kan meerdere auteurs hebben (in een diverse rol). Voorbeelddata kan je vinden in Tabel 3.11.

```
Reader(id_reader, name, surname, birthdate, email)
Rating(id_reader, id_book, date_start, date_end, rating, title, year,
    publisher)
Book_author(id_book, id_author, author_role)
```

id_reader	name	surname	birthdate	email
542133	Peter	Thiers	12-03-1970	peter.thiers@outlook.com
542315	Magda	Ockier	23-98-1956	magda.ockier@telenet.be

Reader

id_reader	id_book	date_start	date_end	rating	title	year	publisher
542133	04321	12-04-2016	28-04-2016	4	Caligula	1993	Gallimard
542133	90234	12-04-2016	16-04-2016	3	Caligula: A Biography	2011	Uni. of Cali. Press
542315	90234	02-04-2016	05-04-2016	4	Caligula: A Biography	2011	Uni. of Cali. Press
542315	89321	12-11-2016	19-11-2016	4	De Bekeerlinge	2016	De Bezige Bij

Rating

id_titel	id_auteur	rol_auteur
89321	67435	auteur
90234	12543	auteur
90234	54321	vertaler
90234	32165	vertaler
90234	11265	vertaler
04321	10345	auteur
04321	15436	editor

Book_author

Tabel 3.11: Voorbeelddata lezers en gelezen boeken.

Leden van een sportclub

We krijgen een lijst van leden van een sportclub. Elk lid behoort tot nul of één ploeg. Het tupelschema is het volgende:

```
Member(id_member, name, surname, birthdate, id_team, team_name,
    lowest_birthyear, highest_birthyear)
```

Voorbeelddata kan je vinden in Tabel 3.12.

id_member	name	surname	birthdate	id_team	team_name	lowest birth year	highest birth year
12345	Pieter	Devroe	18-10-2007	U10	pagadders	2007	2008
43256	Karel	Descamps	23-05-2007	U10	pagadders	2007	2008
43257	Kenny	Deketele	04-10-2008	U10	pagadders	2007	2008
32124	Tore	Devos	01-01-2005	U12	welpen	2005	2006
54798	Pieter	Samyn	19-11-2006	U12	welpen	2005	2006

Tabel 3.12: Voorbeelddata leden van een sportclub.

Leden van een sportfederatie

We krijgen een lijst van leden van een sportfederatie. Elk lid behoort tot één club. Elke club heeft slechts één "thuisgemeente".

```
Member (id_member, name, surname, birthdate, id_club, club_name, club_town)
```

Voorbeelddata kan je vinden in Tabel 3.13.

id_member	name	surname	birthdate	id_club	club_name	club_town
12345	Pieter	Devroe	18-10-2007	908	Apolloon	Kortrijk
54789	Pieter	Samyn	19-11-2006	908	Apolloon	Kortrijk
54792	Anja	Devroe	23-09-1983	802	Sporting Nelo	Neerpelt
56789	Pjotr	Teuninck	12-10-1987	802	Sporting Nelo	Neerpelt

Tabel 3.13: Voorbeelddata leden van een sportclub.

3.5.2 Studentenhuizen

Eigenaars van een studentenhuis verhuren meerdere studentenkoten. Binnen elk studentenhuis krijgt elk kot een opeenvolgend nummer, dit telkens vanaf nummer 1. Binnen een studentenhuis kan de verhuurprijs van elk kot verschillend zijn.
We gebruiken hiervoor volgende tabel:

```
Student_room (id_house, house_owner, address, town, id_room1, price1, rented1,
    id_room2, price2, rented2, ...)
```

Doe of beantwoord het volgende:

- In welke normaalvorm staat deze tabel?
- Normaliseer tot de derde normaalvorm.
- Teken het datastructuurdiagram.

3.5.3 Monitoraat

Veronderstel dat de deelname van studenten aan monitoraten als volgt in een tabel geregistreerd wordt. Hierbij wordt een student gekoppeld aan een monitoraat (council). We weten van een student ook in welke klas hij zit en wat zijn naam is.

```
Study_council (id_student, name, id_council, council_description, class)
```

Doe of beantwoord het volgende:

- In welke normaalvorm staat deze tabel?
- Normaliseer tot de derde normaalvorm.
- Teken het datastructuurdiagram.
- Indien ook het aantal afwezigheden (*absences*) moet bijgehouden worden, waar moet dit attribuut dan toegevoegd worden?

3.5.4 Studentenhuizen [2]

De studentendienst houdt de gegevens van studentenkoten in de buurt van de hogeschool bij in volgende tabel. Het kamer id (`id_room`) betreft hier een uniek nummer over alle studentenhuizen heen.

```
Student_room (id_house, house_owner, address, town, id_room, room_price,
    rented)
```

Doe of beantwoord het volgende:

- In welke normaalvorm staat deze tabel?
- Normaliseer tot de derde normaalvorm.
- Teken het datastructuurdiagram.

3.5.5 Archeologische vondsten

Beschouw een inventaris van archeologische vondsten. Deze voorwerpen zitten in kisten. Per kist beschrijven we de voorwerpen in deze kist. Elk voorwerp heeft ook een type.

```
Object (id_object, description, height, width, weight, id_type, type_name,
    type_description, id_chest, chest_open_date, chest_location)
```

Doe of beantwoord het volgende:

- In welke normaalvorm staan deze tabel?
- Normaliseer tot de derde normaalvorm.
- Teken het datastructuurdiagram.

3.5.6 Lijst uitleningen

In bedrijf X zijn er een aantal speciale toestellen (digitale fototoestellen, videocamera's, projectoren, …) die aan de werknemers (voor hun werk) uitgeleend kunnen worden. De bediende die de uitleningen beheert wil op elk moment de huidige toestand (uitgeleend of niet) en de huidige en vorige uitleners kennen.
Voorbeelden vind je in Tabel 3.14 en Tabel 3.15

Gevraagd:
Maak een relatie die deze informatie voorstelt. Normaliseer ze tot 3NF. Verantwoord elke stap.

Device ID: 1245
Description: digital camera Nokia
Borrowers:

code	name	function	start_date	return_date
TOLLU	De Tollenaere Luc	Productiechef	02-02-2015	15-02-2015
SPLMA	Desplenter Marc	Ingenieur R&D	22-02-2015	24-02-2015
LEUKA	Deleu Katrien	Developer	25-02-2015	---

Tabel 3.14: Voorbeeldddata uitleningen van toestellen

Device ID: 1246
Description: video camera Hitachi
Borrowers:

code	name	function	start_date	return_date
TOLLU	De Tollenaere Luc	Productiechef	15-02-2015	15-02-2015
ACXJO	Acx Johan	Personeeldirecteur	01-03-2015	---
LEUKA	Deleu Katrien	Developer	15-02-2015	16-02-2015
TOLLU	De Tollenaere Luc	Productiechef	16-02-2015	20-02-2015

Tabel 3.15: Voorbeeldddata uitleningen van toestellen

3.5.7 Inkooporders

In Tabellen 3.16 en 3.17 vind je twee informatiebehoeften met betrekking tot inkooporders. Ga uit van elke informatiebehoefte apart en normaliseer tot de derde normaalvorm.

Doe dit stap voor stap. Verantwoord iedere stap. Verenig dan de resultaten van beide normalisaties tot één relationeel databasemodel.

Order ID	8376			Date: 16/10/15
Supplier details:				
3921	Quick and Cheap			
	Industryroad 217			
	Gouda			
Delivery date:	24/11/15			
Item ID	Description	Amount	Price each	Total price
3216	Radio	20	50,00	1.000,00
4189	Televisie	5	400,00	2.000,00
7283	Mixer	10	20,00	200,00
			Total:	3200,00

Tabel 3.16: Oefening inkooporders: order 1

Item ID	3216		Radio	
Amount available:	6			
Orders				
Order ID	Supplier	Price each	Delivery date	Amount
8376	Quick and Cheap	50	24/11/15	20
7125	Fast and Reasonable	48	30/11/15	10
			Ordered amount:	30

Tabel 3.17: Oefening inkooporders: order 2

3.5.8 Vrachtwagenverhuur

Het bedrijfje CAMRENT verhuurt kleine vrachtwagens aan particulieren. Een voorbeeld van hun verhuurdocumenten kan je vinden in Tabel 3.18.

Ontwerp een kleine database voor deze informatiebehoefte door te normaliseren tot de derde normaalvorm.

Truck number:	125864
Brand:	Mercedes
Year:	2014
Chassisnumber:	…
Kilometers:	56.893
Remarks:	…
Customer:	4586 Peeters André Graaf Karel de Goedelaan 12 8500 Kortrijk (056) 45.56.23
Date departure:	27-01-2015
Date return:	28-01-2015
Date actual return:	…
Km. at start:	56.893
Km. at arrival:	…
Special remarks:	…

Tabel 3.18: Vrachtwagenverhuur: verhuurdocument van CAMRENT.

Opmerkingen vooraf:

- Merk op dat een vrachtwagen in de loop van een jaar meerdere keren aan dezelfde klant verhuurd kan worden.
- Een klant kan meerdere vrachtwagens op hetzelfde moment huren.
- We verhuren niet tweemaal dezelfde vrachtwagen aan dezelfde klant op dezelfde dag. Dat wordt als één verhuring beschouwd.
- We kunnen de vrachtwagen eventueel wel op dezelfde dag aan twee verschillende klanten verhuren
- *Truck number* is uniek voor elke vrachtwagen, *Customer number* is uniek voor elke klant.

3.5.9 Vakken in een hogeschool

Gegeven de volgende informatiebehoefte: in een hogeschool houdt men van ieder vak een fiche bij met de belangrijkste gegevens over het vak: de code, de naam, de beschrijving, de klassen die dat vak krijgen en de code en de naam van de docent die dat vak aan die klas geeft. Bij iedere klas geeft men de code van de afdeling, de naam van de afdeling en de coördinator van de afdeling. Een voorbeeld kan je vinden in Tabel 3.19.

Ontwerp hiervoor een kleine database door te normaliseren tot de derde normaalvorm.

Course ID:	IT1002		
Name:	Lab SQL		
Description:	Aanleren van de databasequerytaal SQL.		
Taught by/to:			
LTOL	L. De Tollenaere	2TIA	TI (Toegepaste Informatica (Arne Vandenbussche))
LTOL	L. De Tollenaere	2TIB	TI (Toegepaste Informatica (Arne Vandenbussche))
MSPL	M. Desplenter	2TIC	TI (Toegepaste Informatica (Arne Vandenbussche))

Tabel 3.19: Vakken in een hogeschool: voorbeeld data.

3.5.10 Informatica-afdeling

Som van het onderstaande probleem de relevante attributen op, bepaal de functionele afhankelijkheden en normaliseer tot de derde normaalvorm.

- In een informatica-afdeling wil men gegevens over software-ontwikkelingsprojecten bijhouden. Ieder project heeft een unieke naam en een omschrijving van het project.
- Verder is het van belang wie de projectleider van het project is, wat de begindatum en de einddatum van het project was. Men wil ook bijhouden welke werknemers voor het project gewerkt hebben, in welke functie (C++-programmeur, analist, tester, enz.) en hoeveel uur ze in deze functie voor het project gewerkt hebben.
- Eén persoon kan in één project meerdere functies vervuld hebben.
- Van iedere medewerker in de informatica-afdeling willen we zijn naam (die niet noodzakelijk uniek is), zijn voornaam en zijn specialiteit (gegevensanalyse, procesanalyse, networking, C++-programmeur, enz.) kennen. Van iedere specialiteit wordt ook een korte omschrijving gegeven.

3.5.11 Voorraadbeheer

Hieronder vind je een korte probleembeschrijving.

- Een hersteldienst heeft een eenvoudige database voor voorraadbeheer. In die database vind je een lijst met alle onderdelen (niet de individuele stukken, maar iedere onderdeelsoort is een element in de lijst). Ieder onderdeel heeft een code. Die code is voor ieder onderdeel verschillend. Daarnaast heeft men ook een beschrijving van het onderdeel, de verkoopprijs van het onderdeel, de huidige voorraad en de minimumvoorraad.
- Ieder onderdeel is van één bepaald type. Er worden ongeveer twaalf types onderscheiden. Ieder type heeft een naam, een beschrijving en een sectienummer van het magazijn. De artikels zijn in het magazijn immers per type geordend.
- Voor ieder onderdeel houdt men één of meerdere leveranciers bij.
- Iedere leverancier heeft per onderdeel een rangordenummer. De beste leverancier voor dit onderdeel heeft het rangordenummer (`rank`) 1, de tweede beste rangordernummer 2, enzovoort. Eventueel wordt ook de prijs waartegen die leverancier dit onderdeel kan leveren bijgehouden (als die prijs bekend is).
- Van iedere leverancier kent men de naam, het adres (straat, huisnummer, postnummer, gemeente), het telefoonnummer en het faxnummer.

Tot nu toe hield men die gegevens bij in een aantal eenvoudige tekstbestanden. Men besluit die gegevens bij te houden in een relationele PC-database. Een amateurknutselaar heeft hiervoor de volgende database ontworpen, die bestaat uit twee tabellen.

```
Part (id_part, description, price, stock, minimum_stock, type_name,
    type_description, section_number)
Supplier (id_part, supplier_name, address, postal_code, town, phone, fax,
    rank, price)
```

Ons hart staat stil bij het zien van een dergelijke niet-genormaliseerde database. Bekijk de attributen, bepaal alle functionele afhankelijkheden en maak dan een *genormaliseerde* versie van deze databank. Teken vervolgens het ERD diagram.

3.5.12 Bibliotheek

Een kleine bibliotheek wil van iedere lezer een overzicht kunnen krijgen. Bekijk de voorbeelddata uit Tabel 3.20.

Ontwerp hiervoor een relationele database door te normaliseren naar derde normaalvorm.

Reader ID:	1024578
Name:	Piet Paaltjens
Address:	Zeeweg 4 4058 Bilderdijk
Phone:	(025)35.26.54
Profession:	Onbekend

Copy	ISBN	Author	Title	Publisher	Year	Date loan	Date actual return
542368	9780321227317	Hulsens Eric	Waarom lusten kinderen nog reuzen?	Infodok	1980	01-03-2015	23-03-2015
457850	4589275623248	Hoppenbrouwers Cor	Jongerentaal	Stubeg	1991	01-03-2015	
478958	5278556611458	Spillebeen Willy	Cortés of de Val	Houtekiet	1994	23-03-2015	
789456	4585263252345	Istendael Geert van	Bekentenissen van een Reactionair	Atlas	1994	23-03-2015	

Tabel 3.20: Voorbeelddata bibliotheek

Opmerkingen vooraf:

- Een lezer kan meerdere keren eenzelfde boek lenen.
- Een exemplaar kan op een dag maar aan één lezer uitgeleend worden.

3.5.13 Garagebedrijf

Bestudeer onderstaande voorbeelden van informatiebehoeften in een garagebedrijf. **Normaliseer** elke informatiebehoefte apart en voeg dan de resultaten samen tot één **genormaliseerde database**.

In een garagebedrijf werken meerdere mecaniciens. Een te repareren auto kan door meerdere mecaniciens hersteld worden. Elke mecanicien stuurt, wanneer hij onderdelen nodig heeft, een bon naar het centrale magazijn.

- Een mecanicien kan op 1 dag meerdere keren aan dezelfde wagen gewerkt hebben (zie Tabel 3.21).
- Elke mecanicien stuurt een bon naar de dienst administratie met het gepresteerde werk (zie Tabel 3.22).
- De administratie heeft een lijst met de tarieven voor alle onderdelen. De verkoopprijs wordt berekend uit de inkoopprijs met een vaste winstmarge (zie Tabel 3.23).
- De administratie heeft een lijst met van alle klanten hun betreffende wagens (zie Tabel 3.24).
- De garagehouder wil van elke werknemer een overzicht van het gepresteerde werk in de laatste maand (zie Tabel 3.25).

Plate:	Xxxxxxxx	
Employee ID:	Xxxxxxxx	
Date:	Xx/xx/xx	
Article number.	Description	Amount
Xxxx	Xxxxxxxxx	xxx
Xxxx	Xxxxxxxxx	xxx
Xxxx	Xxxxxxxxx	xxx

Tabel 3.21: Beschrijving werk van mecanicien aan auto.

Plate:	Xxxxxxxx		
Employee ID:	Xxxxxxxx		
Date:	xx/xx/xx		
Description labour	Start time	End time	Duration
Xxxx	Xxxxxxxxx	xxxx	xxx

Tabel 3.22: Bon gepresteerd werk.

Article number	Description	Purchase price	Sale price
xxxxx	Xxxxxxxxxxxx	xxxxx	xxxxxxx

Tabel 3.23: Tarieven

Client ID	Name	Address	Plate	Brand	Model
Xxxx	Xxxxxxx	Xxxxxxxxxx	xxxxx	xxxxx	xxxxx
			xxxxx	xxxxx	xxxxx
			xxxxx	xxxxx	xxxxx
Xxxx	Xxxxxxxx	Xxxxxxxxxx	xxxxx	xxxxx	xxxxx

Tabel 3.24: Wagens van klanten

Employee ID:	Xxxxxxxx			
Employee name:	Xxxxxxxxxx			
Date	Start hour	End hour	Duration	Plate
xx/xx/xx	Xxxxx	xxxxx	xxxxx	xxxxx
	Xxxxx	xxxxx	xxxxx	xxxxx
	Xxxxx	xxxxx	xxxxx	xxxxx
xx/xx/xx	Xxxxxxxxx	xxxx	xxx	xxxxx
		Totaal:	xxxxx	

Tabel 3.25: Overzicht gepresteerd werk

3.6 Denormalisatie en optimalisatie

In dit hoofdstuk beschreven we het normalisatieproces en lieten we zien waarom dit normalisatieproces zo belangrijk is. In sommige gevallen heeft het geen zin om tot de derde normaalvorm te gaan of is het om efficiëntieredenen zelfs beter om te gaan denormaliseren.

Genormaliseerde relaties vermijden modificatie-anomalieën en worden daarom meestal geprefereerd boven ongenormaliseerde relaties. Het kan echter voorkomen dat het normalisatieproces meer kost dan het uiteindelijk opbrengt. Stel dat de volgende relatie gegevens is:

`KLANT (Klantnummer, Klantnaam, Deelgemeente, Provincie, Postcode)`

Deze relatie staat niet in de DK/NF omdat zij de volgende functionele afhankelijkheid bevat:

`Postcode` $\rightarrow$ `(Deelgemeente, Provincie)`

Die afhankelijkheid is geen sleutelafhankelijkheid. We kunnen de relatie dus splitsen:

```
KLANT (Klantnummer, Klantnaam, Postcode)
CODE (Postcode, Deelgemeente, Provincie)
```

De vraag is of dit echt een verbetering is. Bij het raadplegen van klantgegevens moeten nu steeds twee tabellen benaderd worden. Dit vertraagt het raadplegen en dat is waarschijnlijk nadeliger dan het eventueel vaker voorkomen van de combinatie `(Deelgemeente, Provincie)`.

Het kan dus in het belang van efficiënte verwerking soms verstandig zijn een relatie niet te normaliseren of eerst te normaliseren en vervolgens weer te 'ontnormaliseren'. Vaak betekent dit wel dat we in de programmacode extra controles moeten inbouwen om bij modificaties geen inconsistenties te krijgen.

De Powerpointpresentatie voor dit hoofdstuk kan je terugvinden in de Toledocursus.

Op Toledo vind je een zelftoets "Oefentoets normaliseren". Los deze toets op.

Op Toledo vind je ook een individuele opdracht "Vakken in een hogeschool" die je zelf oplost en via Toledo inlevert.