3. Gegevens opvragen uit meerdere tabellen

3.5 Implementatie van joins

Cartesische producten en joins zijn de relationele operaties die relatief het meeste impact hebben op de uiteindelijke verwerkingstijd T van een query. Dikwijls leidt het frequent uitvoeren van joins op steeds dezelfde tabellen, en de systeembelasting die dit veroorzaakt tot de beslissing om het databankontwerp te denormaliseren.

Methodes hiervoor:

1. 0pslag van een kindtabel, verbreed met beschrijvende attributen van de oudertabel.

2. Opslag van een kindtabel, verbreed met een (naar een grootoudertabel) verwijzende sleutel van de oudertabel, zodat men 1 of meer niveau's kan **kortsluiten**.

3. Opslag van doorsnedetabellen, verbreed met attributen van de tabellen met een onderlingen veel-op-veel relatie.

4. Opslag van een oudertabel, verbreed met statistische informatie over kindtabellen en andere tabellen waarmee deze een relatie hebben.

5. Opslag van een tabel, verbreed met resultaten bekomen uit een of meerdere auto-joins van de tabel zelf.

Een wat extreme techniek om de belastende impact van specifieke joins te reduceren noemt men **pre-joining** of soms **clustering**. Meestal bevatten diskblokken enkel rijen van dezelfde tabel. Bij pre-joining daarentegen verzamelt men in een blok rijen van verschillende tabellen, die frequent met een bepaald join-predikaat aan elkaar gekoppeld worden. Hierdoor kan men het aantal I/O-operaties voor een specifieke query beperken. Alle andere queries worden daarentegen minder efficiënt uitgevoerd.

Methodes voor het verwezenlijken van joins
Bij elke query bepaalt de query optimizer welke strategie effectief gevolgd zal worden. Dit gebeurt deels op basis van een aantal heuristische en syntactische regels, toegepast op het navigatiepad van de query en de gebruikte predicaten, deels op basis van een raming van de uitvoeringstijd en de noodzakelijke geheugenbronnen.
Om voor elk van de methodes een raming van de werkekingstijd te krijgen, wordt verondersteld dat men slechts 2 tabellen (X en Y) joint, dat deze $\#_x$ en $\#_y$ aantal rijen hebben. De aantallen I/O operaties (aantal diskblokken) vereist om de tabellen op te slaan: β_x en β_y.
(Bij elk van deze methodes is het idee van producttabellen conceptueel: de query processor voert join operaties meestal uit zonder de producttabel ook fysiek te maken)

1 **block nested loop join** (meest eenvoudig algoritme)
In **4 geneste lussen** wordt elke rij van X met elke rij van Y vergeleken. De 2 meest uitwendige lussen lezen blokken van elke tabel in het geheugen in, terwijl de 2 inwendige lussen elke rij van een blok van de ene tabel met elke rij van een blok van de andere tabel met elkaar vergelijken.
Verwerkingstijd: $T \sim \beta_x + (\beta_x \times \beta_y)$
De term β_x zorgt ervoor dat men voor de meest uitwendige lus best de tabel met het kleinst aantal blokken kiest. In de meeste RDBMS'en komt dit doorgaans neer op de linkeroperand van de join-operatie

2 **index nested loop join**: kan gebruikt worden als alle in het join-predikaat optredende attributen van 1 van de tabellen geïndexeerd zijn.
Hier worden **3 geneste lussen** gebruikt.
De meest uitwendige lus leest blokken in van de niet-geïndexeerd veronderstelde tabel X, een middenste lus loopt doorheen elke rij van dergelijke blokken en de meest

inwendige lus zoekt via de indices naar corresponderende rijen van Y
Verwerkingstijd: $T \sim \beta_x + \#_x \times f_y$
Waarbij f_y een functies is van de gebruikte indexmethode en van de aantallen rijen die aan het join-predikaat voldoen.
Vooral voor equi-joins zijn index nested loop joins veel efficiënter dan block nested loop joins.
De query optimizer kan in de praktijk enkel tot het gebruik van index nested loop joins beslissen indien de operanden van het join-predikaat hetzij kolomnamen, hetzij constanten zijn. (Van zodra kolommen in het join-predicaat bewerkingen ondergaan, kan het resultaat van deze bewerkingen enkel bij uitvoering bepaald worden, en niet bij evaluatie van de optimale strategie)

3 **sort-merge join**
Als men van beide tabellen X en Y mag veronderstellen dat ze gesorteerd zijn op de attributen van het equi-join predikaat, dan volstaat het om beide tabellen **gemerged** in te lezen.
Op elk ogenblik tijdens de verwerking wordt van elke tabel 1 blok in het geheugen gehouden, en beschouwt men van elk blok 1 rij. Afhankelijk van de relatieve volgorde van deze 2 rijen met betrekking tot het join-predikaat, schuift men 1 rij verder in 1 of beide tabellen. Overschrijdt men hierbij de grens van een blok, dan wordt een nieuw blok ingelezen voor die tabel. De procedure wordt herhaald tot alle rijen verwerkt zijn.
Verwerkingstijd: $T \sim \beta_x + \beta_y$
Indien 1 of beide tabellen vooraf gesorteerd moeten worden, dan moet $\beta_x \times \ln_2(\beta_x)$ en/of $\beta_y \times \ln_2(\beta_y)$ bij de verwerkingstijd opgeteld worden.

4 **Hash join**: verwerking gebeurt in 2 stappen:

1 de tabellen worden onafhankelijk van elkaar ingelezen en op elke rij wordt een hashfunctie toegepast. Equivalente waarden voor het join-predikaat leveren dezelfde hash-waarde op. Alle rijen van een tabel met dezelfde hash-waarde worden gegroepeerd in een **partitie**, en in blokken van partities weggeschreven. (partities met een bepaalde hash-waarde kunnen rijen met diverse waarden voor het join-predicaat bevatten)

2 Voor elke partitie wordt een join uitgevoerd. Van beide tabellen worden de overeenkomstige partities ingelezen. Op de kleinste partitie (**build partitie**) wordt in het geheugen opnieuw een hashtabel gecreëerd, nu met een andere hashfunctie. De rijen van de andere partitie (**probe partitie**) worden rij per rij verwerkt: telkens wordt op de rij de hashfunctie toegepast, om via de hashtabel de beperkte verzameling kandidaatrijen van de build partitie te vinden die aan het join-predikaat voldoen.

Verwerkingstijd: $T \sim 3 \times (\beta_x + \beta_y)$ in ideale omstandigheden.
Indien er onvoldoende geheugen ter beschikking is om de hashtabel voor de kleinste partitie volledig te bevatten, dan moet men het algoritme voor die partitie recursief uitvoeren. De verwerkingstijd neemt toe met $\sim \beta \times \ln(\beta)$
Indien 1 van de tabellen in zijn geheel klein genoeg is, kan de RDBMS de eerste stap van het algoritme volledig achterwege laten.

5. Gegevens wijzigen

5.3 Wijzigbare views

Het wijzigen van een inline view of een CTE is een alternatief om een update via een join van van diverse tabellen uit te voeren.

Dergelijke bewerkingen zijn wel aan restricties onderworpen, die kunnen verschillen tussen de verschillende RDBMS'en.

Beperkingen:

1. Een kolom van een CTE (of inline view) is over het algemeen enkel wijzigbaar indien wijzigingen zich éénduidig laten herleiden tot een bewerking op één enkele tabel.

2. De kolommen van de CTE moeten de attributen van minstens 1 kandidaatsleutel van die tabel bevatten, om de te wijzigen rijen te kunnen identificeren.
   Een CTE die daaraan beantwoordt, wordt **key-preserved** genoemd.

3. Bij gebruik van het sleutelwoord distinct, of bij groeperen en het gebruik van statistische functies is er zeker geen eenduidig verband tussen de resultaatrijen en de rijen van de brontabel, zodat kolommen in de CTE die daardoor beïnvloedt worden, niet wijzigbaar zijn.

4. De kolomwaarden die gewijzigd worden, mogen niet berekend worden op basis van bewerkingen op andere kolommen of scalaire subqueries.

5. Kolommen die het resultaat zijn van samengestelde select opgrachten met set-operatoren kan men niet wijzigen.

Om updates via een CTE uit te voeren kan men de CTE beperken tot:
- Kolommen die de rijen van de doeltabel kunnen identificeren
- Kolommen die attributen bevatten die gewijzigd moeten worden. Deze kolommen zijn onderworpen aan de vorige beperkingen.
- Kolommen die de nieuwe waarden bevatten. Geen beperkingen.

Zo'n CTE kan ook gebruikt worden om een **preview** te produceren voor dat men de wijzigingen effectief doorvoert.

```
WITH r as (……)
UPDATE r
SET name = cname
WHERE ….
```

Een optionele where-clausule in de CTE of de update opdracht kan de bewerking beperken tot de rijden die effectief een verandering moeten ondergaan.
De CTE kan voor de berekening van de nieuwe waarden een beroep doen op willekeurige bewerkingen op andere kolommen, zoals subqueries en andere CTE's, set-operatoren, groeperingen en statistische functies.
De CTE die gewijzigd wordt, mag ook verwijzen naar een recursieve CTE.

6. DDL aspecten

6.1 Tabel-en kolomdefinities

DDL-code (Data Definition Language) beschrijft de structuur van tabellen, samen met de regels die ervoor gelden. Er zijn hierin grote syntax-verschillen tussen de diverse RDBMS implementaties. SQL voorziet de opdrachten **create**, **alter** en **drop**.

De beschrijving van tabellen wordt opgeslagen in een verzameling administratietabellen van de databank, de **systeemcatalogus** of **datadictionary**. Behalve tabellen worden ook nog andere objecten in de datadictionary opgeslagen: constraints, views, indexen, stored procedures en triggers.

Elk object vereist een identificerende naam. Deze kan steeds expliciet gekozen worden in de create opdracht die het object definieert. De tabellen van de systeemcatalogus zijn gewone tabellen zoals gebruikerstabellen. Hun structuur wordt ook in de systeemcatalogus zelf beschreven.

De structuur van de systeemcatalogus is zeer RDBMS-specifiek. Toch zijn er veel overeenkomsten.
Zo zijn er altijd specifieke **metatabellen** voor elk type object:

- Metatabel met allerlei tabelkenmerken en een rij voor elke tabel
- Metatabel met kolomkenmerken en een rij voor elke kolom van elke tabel in de databank
- Tabellen met informatie voor objecten zoals sleutels, indexen en stored procedures

Veel RDBMS'en laten **metadata queries** toe: hierdoor is het voor ontwikkelaars mogelijk om met reguliere DML syntax veel informatie over de structuur van de tabellen te bekomen (zowel helpfunctie als controlefunctie). Het rechtstreeks wijzigen van de systeemcatalogus wordt nauwelijks toegelaten.

Een tabel aanmaken gebeurt met de **create table** opdracht. Deze opdracht bestaat uit een aantal regels, van elkaar gescheiden door komma's. Elke regel verwijst naar een kolomdefinitie of contstraintdefinitie.

```
CREATE TABLE Races (
   date smalldatetime NOT NULL
   rid int NOT NULL
   …
)
```

Eens de tabel bestaat, kunnen meerdere kolommen of constraints toegevoegd, gewijzigd of verwijderd worden met de **alter table** opdracht.

Een basis kolomdefinitie in de create table of alter table opdracht bestaat uit:

- Kolomnaam
- Datatype van de kolom
- Evt default specificatie
- Evt not-null constraint. Zonder een not-null constraint is de kolom optioneel.

Een default-specificatie (niet verplicht) zorgt ervoor dat een cel van een nieuwe rij automatisch kan ingevuld worden met een default waarde. (deze waarde hoeft geen constante te zijn, bewerkingen op constante uitdrukkingen zijn ook toegelaten, vb: random getal).

Het genereren van een random getal is ontoereikend om een surrogaatsleutel te creëren. Elke RDBMS heeft hiervoor alternatieven, zoals **sequencers** en **identity kolommen**, die gegarandeerd een unieke identifier kunnen produceren.

Wijzigen van een kolomdefinitie is zeer afhankelijk van de RDBMS. Procedure die overal toegepast kan worden:

1. Een kolom toevoegen met de nieuwe specificaties
2. Met een update opdracht de celinhouden van de oude kolom dupliceren in de nieuwe kolom
3. Eventuele constraints die naar de ouder kolom verwijzen verwijderen
4. De oude kolom zelf verwijderen
5. De eventuele constraints opnieuw definiëren op de nieuwe kolom

Indien je de kolomnaam wil blijven behouden (en de RDBMS ondersteunt geen procedure voor het veranderen van objectnamen) dan is een derde intermediaire kolom nodig.

Om een tabel te verwijderen moet men de **drop table** opdracht uitvoeren, met de tabelnaam als argument

```
DROP TABLE Races
```

Deze opdracht vernietigt niet alleen de inhoud van de tabel (zoals ook met de delete from opdracht kan) maar ook de structuurbeschrijvingen van de tabel in de systeemcatalogus, inclusief corresponderende constraints, views en indexen.
Het verwijderen kan mislukken als er rijen bestaan in een andere tabel, met verwijzingen naar een rij van de te verwijderen tabel.

6.2 Constraints

De mechanismen die ervoor zorgen dat de RDBMS voor de realisatie van beperkingsregels en gedragsregels kan instaan worden **constraints** of **integriteitsregels** genoemd. Bij elke poging tot aanpassing van de gegevens controleert de RDBMS of de nieuwe inhoud nog steeds aan alle constraints voldoet.

Constraints kunnen op 2 manieren in DDL gedefinieerd worden:

1. **Inline**: als onderdeel van een kolomdefinitie
2. **out-of-line**: via een aparte contraintdefinitie.

In de systeemcatalogus is van dit onderscheid niets meer te merken.

Een expliciete constraintdefinitie wordt optioneel voorafgegaan door het sleutelwoord **constraint** en een zelf gekozen constraintnaam.
Net als kolomdefinities kan men **out-of-line** constraintdefinities simultaan met een create table opdracht uitvoeren, of nadien via een alter table opdracht toevoegen of verwijderen. (dit is noodzakelijk omdat men sommige ingrepen in een databank enkel kan uitvoeren als men bepaald constraints tijdelijk buiten werking stelt)

Er zijn diverse contrainttypes:

1. een **not-null constraint** kan enkel inline (of via een check-constraint) gedefinieerd worden. Indien deze aanwezig is, wordt aangegeven dat de kolom in elke rij een waarde moet bevatten.

2. een **primaire sleutel constraint** zorgt ervoor dat elke rij enkel een unieke combinatie van waarden kan hebben voor de deelnemende attributen. Na het sleutelwoord PRIMARY KEY staan tussen haakjes de kolomnamen waaruit de primaire sleutel is opgebouwd. Deze constraint is niet verplicht (wel aangewezen, anders kunnen duplicaten optreden in een tabel zonder primaire sleutel.)

   ```
   ALTER TABLE Results
   ADD CONSTRAINT pk_Results
       PRIMARY KEY (rid,cid)
   ```

   Een primary key over 1 kolom kan ook inline gedefinieerd worden.
   In elke tabel kan maar 1 primaire sleutel constraint gelden.

3. Een **verwijzende sleutel constraint** dwingt af dat de specifieke combinatie van waarden, die in de (verplichte) attributen van de sleutel vermeld worden, ook werkelijk optreedt in de oudertabel.

   ```
   ALTER TABLE Results
   ADD CONSTRAINT fk_Results_Competitors
       FOREIGN KEY (cid, fis)
         REFERENCES Competitors (cid, fis)
           ON UPDATE CASCADE
           ON DELETE CASCADE
   ```

   De sleutelwoorden **foreign key** en **references** worden respectievelijk gevolgd door de attributen die de verwijzende sleutel vormen en de gerefereerde tabel met zijn kandidaatsleutel. De kandidaatsleutel moet in de gerefereerde tabel opgenomen zijn in een primaire sleutel of unique-constraint.
   Optionele refererende-actieregels geven aan wat er moet gebeuren als de referentiële integriteit wordt overtreden. Deze regels kan men opnemen door middel van **on delete**

en **on update** clausules. Voor elke van de clausules kan men 4 mogelijke opties kiezen:

1. **no-action** regel (soms ook met synoniem **restrict** aangeduid) wordt standaard toegepast. Het verwijderen of aanpassen van een ouderrij wordt onmogelijk gemaakt indien er nog verwijzingen naar bestaan vanuit tabellen met dergelijk referentie-actieregel. Dit heeft tot gevolg dat men kindtabellen altijd eerst moet aanpassen.

2. **cascade** optie: bij verwijdering of aanpassing van de ouderrij ondergaan rijen met referenties ernaar trapsgewijs precies dezelfde actie.

3. **set null**: verwijzingen in eventuele kindrijen worden op NULL ingesteld

4. **set default:** verwijzingen in eventuele kindrijen worden op de default waarde ingesteld.

Recursieve verwijzende sleutels kunnen pas gedefinieerd worden nadat de tabeldefinitie uitgevoerd is (wordt toegevoegd via alter table opdracht).

4. Een **unique-constraint** kan voor uniciteit zorgen van willekeurige kolomcombinaties

5. **Check-constraints** hebben als argument een predicaat. Enkel rijen toelaten waarvoor een specifiek predicaat true of unknown is.
   Alhoewel de SQL standaard predicaten ondersteunt met gecorreleerde subqueries die naar andere tabellen refereren, beperken de meeste RDBMS'en zich toch predicaten die een willekeurige bewerking uitvoeren op de kolommen van de rij zelf.

```
ALTER TABLE Regios
ADD CONSTRAINT ck_elevation CHECK ( elevation >=-10 AND elevation <=8850 )
```

6.3 Views

Views of afgeleide tabellen zijn select opdrachten die men als objecten in de datadictionary bewaart.

Gelijkenissen met CTE's

Views bieden nagenoeg dezelfde functionaliteit als niet-recursieve CTE's. Views kunnen net als CTE's opgevat worden als virtuele tabellen die overal in SQL opdrachten gebruikt kunnen worden waar tabellen verwacht worden. Bij het uitvoeren van een dergelijke opdracht wordt de select opdracht van de view in de instructie opgenomen, net als een inline view of CTE.

De query optimizer beslist (afhankelijk van wat de hoofdquery doet) tot:

- **view materialisation**: de view wordt voorafgaand globaal uitgevoerd en de volledige resultaattabel ervan wordt in het geheugen of tijdelijk op disk opgeslagen.

- **view resolution**: de view wordt selectief en simultaan met de hoofdquery verwerkt.

Net als CTE's kunnen views gebruikt worden om complexe problemen stapsgewijs op te lossen, om allerhande syntaxbeperkingen van SQL te omzeilen en om gefaseerd te reageren op een gereorganiseerde databankstructuur.

Verschillen met CTE's

Diverse opdrachten kunnen dezelfde view raadplegen, zonder de view telkens opnieuw te definiëren. Bij CTE's blijft het bereik beperkt tot 1 SQL-opdracht.

Creatie van een view vereist dikwijls DBA privileges. Om CTE's te kunnen gebruiken, zijn geen privileges nodig. Views zijn daarom een belangrijk **autorisatiemiddel:** de DBA kan aan verschillende gebruikers via views een specifiek beeld op maat geven van de databank,

beperkt tot datgene wat ze moeten kunnen raadplegen en wijzigen om hun taak uit te voeren. Dit beeld kan beperkt worden tot op een fractie van een tabel:

- **verticale reductie:** reductie op kolomniveau, door de select-clausule van de view in te stellen
- **horizontale reductie:** reductie op rijniveau, door in te spelen op de where-clausule.

Views worden gecreëerd met de **create view** opdracht, met een volledige select opdracht als parameter.

```
CREATE VIEW victories AS (
     SELECT …
     FROM …
)
```

De kolomnamen van een view kunnen afgeleid worden uit die van de select-clausule, maar kunnen ook expliciet na de viewnaam opgegeven worden.
Men mag views **nesten.**

Om een view te wijzigen moet hij eerst verwijderd worden met de **drop view** opdracht, en daarna opnieuw aangemaakt worden. De **drop view** opdracht verwijdert ook alle andere views die refereren naar de te verwijderen view. Het verwijderen van 1 van de basistabellen van de view, zorgt ervoor dat de view verwijderd wordt.

Sommige RDBMS'en laten in create table of alter table -opdrachten kolomdefinities toe die bereken worden op basis van andere kolommen (**computed columns**). Bij implementaties die dit niet ondersteunen kunnen views diezelfde functionaliteit simuleren.

Het beeld dat de gebruikers hebben van de systeemcatalogus komt meestal niet overeen met de werkelijke structuur van de systeemcatalogus, maar wordt door views gesimuleerd.
Een aantal van deze views bevatten een where-clausule die het resultaat beperken tot objecten waarvan de gebruiker zelf eigenaar is.

Men kan de inhoud van een tabel via een view wijzigen, hiervoor worden de sleutelwoorden **with check option** gebruikt.
De where-clausule fungeert als input/output filter: bij elke toevoeging van rijen of wijzigingen aan kolomwaarden wordt gecontroleerd of de rij nog steeds aan de where-clausule van de select-opdracht voldoet. Indien niet, wordt de wijziging geweigerd.

View met de with check option bieden een eenvoudig alternatief voor complexe check-constraints met gecorreleerde subqueries.

6.4 Indexering

Indexen zijn interne, gesorteerde gegevensstructuren met verwijzingen naar rijen van een tabel. Met indexen kan men vlug rijen in een bepaalde sorteervolgorde aflopen. Hierdoor kunnen bepaalde queries (vb: order by) sneller uitgevoerd worden. Ook rijen, waarvan de attributen aan bepaalde voorwaarden moeten voldoen, kunnen zo sneller gevonden worden. Dit is o.m. van belang bij joins, bij verwijzingen naar sleutelwaarden van een andere tabel.

De query optimizer bepaalt voor elke query of een beschikbare index al dan niet gebruikt kan worden en bepaalt zo de optimale strategie.

Indexen worden als object in de systeemcatalogus gecreëerd met behulp van de **create index** opdracht. Hierbij kunnen verschillende opties meegegeven worden.

```
CREATE NONCLUSTERED INDEX gender_discipline_date
   ON Races (gender ASC, discipline ASC, date ASC)
WITH (….)
```

Men kan indexen verwijderen met de **drop index** opdracht.

Views kan men doorgaan niet indexeren. Wel kan men meestal indexeren (virtuele-kolom indexen) op bewerkingen van waarden van dezelfde rij.

Bij primaire sleutel constraints en unique-constraints worden doorgaans automatisch indexen aangemaakt. Voor de RDBMS is dit blijkbaar de eenvoudigste manier om het toevoegen van duplicaten te vermijden.

6.4.1 Bestandsorganisatie van gegevenstabellen

1. in **heap-bestanden**:
   rijen worden aan bestanden toegevoegd, in volgorde van creatie en onafhankelijk van de kolomwaarden. Nieuwe records worden in het laatste blok van het bestand weggeschreven (een nieuw blok wordt aangemaakt als het vol is).
   Insert opdrachten worden optimaal uitgevoerd: er gaat geen tijd verloren om de locatie van de records te berekenen.
   Om een rij met specifieke kolomwaarden te vinden, moet men het bestand volledig lineair doorzoeken ➔ weinig efficiënt.

   Bij het verwijderen van een rij, wordt de vrijgekomen ruimte gemarkeerd, maar niet onmiddellijk opnieuw gebruikt ➔ periodieke reorganisatie nodig voor compactie van de vrije ruimte

   Goed wanneer tabellen slechts enkele blokken opslag vereisen of om beschikbare opslagcapacitiet optimaal te benutten. Ook goed wanneer men op een tabel frequent queries uitvoert waarbij men alle rijen moet verwerken

2. in **sequentiële bestanden**:
   rijen worden gesorteerd opgeslagen, volgende de waarde van de sorteersleutel (kan uit 1 of meer attributen bestaan).
   Het opzoeken van kolomwaarden die onderdeel zijn van de sorteersleutel, kan nu met veel snellere **binaire** zoekmethoden

   Toevoegen/verwijderen van rijen is problematisch: indien er bij het toevoegen van een rij nog voldoende plaats is in het blok waar het record terecht moet komen, dan moet men enkel dat blok reorganiseren. Indien niet, dan moet men alle (in sorteervolgorde) volgende blokken ook aanpassen.
   Als alternatief kan men werken met tijdelijke **overflow** of **transactiebestanden**, waarin men rijen niet-sequentiëel toevoegt. Een dergelijk bestand wordt slechts periodiek **gemerged** met het sequentiële basisbestand.
   Deze werkwijze heeft een positief effect op insert-opdrachten. Nadeel is dat bij opzoekingsopdrachten het transactiebestand lineair doorzocht moet worden indien de binaire zoekopdracht in het basisbestand geen resultaat oplevert.

3. in **hash-bestanden**
   het adres van het blok waarin een rij opgeslagen wordt, wordt berekend met een hash-functie, op basis van 1 of meerdere attributen.
   Met elke hashwaarde komen een beperkt aantal **slots** overeen, die elk 1 rij kunnen bevatten.

   De hashfunctie moet de rijen gelijkmatig verdelen over de beschikbare blokken.
   Indien er toch geen slot vrij is om een rij op te slaan, dan moet men alternatieve technieken gebruiken, zoals gelinkte lijsten van overloopgebieden, hiërarchieen van overloopgebieden (dynamische hashing) of gemeenschappelijke overloopgebieden die men met een 2e hashfunctie adresseert.

   Hashing is globaal efficiënt voor zoekopdrachten op een specifieke waarde van attributen, maar ontoereikend voor zoekopdrachten met like en between ... and

predikaten.
Ook wanneer er wordt gezocht op slechts een aantal attributen, waarop de hashfunctie gebaseerd is, dan moet de tabel lineair doorzocht worden.

Hash bestanden ondervinden veel overhead indien men 1 van de attributen, waarop men hasht, frequent aanpast ➔ de corresponderende rij moet dan telkens naar een ander blok verplaatst worden.

6.4.2 Implementatie van indexen

De gegevensstructuur van een index bestaat zelf uit records, die voor een specifieke combinatie van waarden voor een aantal attributen, het adres (de **record identifier**) van de overeenkomstige rij in het gegevensbestand aangeven. NULL waarden worden niet in de index opgeslagen. De gegevensstructuren van een index wordt zelf in een bestand opgeslagen, het **indexbestand**.

1. In **index-sequetiële bestanden** zijn zowel het gegevensbestand als het indexbestand georganiseerd als sequentiële bestanden, gesorteerd op dezelfde combinatie van attributen, meestal de primaire sleutel van de tabel. Zo'n index noemt men een primaire index.
   Indien de combinatie van attributen waarop men sorteert niet noodzakelijk uniek is, spreekt men van een **geclusterde index**. Een bestand kan slechts 1 primaire of geclusterde index hebben. Omdat het gegevensbestand gesorteerd verondersteld mag worden, moet men niet voor elke rij een indexrecord voorzien. Een indexrecord naar de eerst rij in elk blok is meer dan voldoende. De indexering is dan niet **dicht** (dense) maar **ijl** (sparse). Dit biedt meer mogelijkheden om de volledige index in het hoofdgeheugen op te slaan.
   Wanneer men de tabel echter frequent wijzigt, blijkt het gebruikt van sequentiële bestandsstructuur niet aangewezen.

2. Bij **secundaire indexen** worden de indexbestanden opnieuw sequentiëel opgeslagen. De gegevensbestanden zijn echter ofwel niet sequentieel, ofwel worden ze in een andere sorteervolgorde als de attributen van de index opgeslagen (andere sleutel). Het aantal secundaire indexen is onbeperkt. Secundaire indexen vereisen geen unieke combinaties van attributen.

3. De efficiëntie van binaire opzoekingen via een indexbestand daalt logaritmisch met het aantal blokken dat vereist is om het indexbestand op te slaan. Deze situatie is vergelijkbaar met het beheer van de paginatabellen van virtueel geheugen. Oplossing hiervoor is paginatabellen zelf in het virtueel geheugen opslaan. Dit kan men hier ook uitwerken: het indexbestand wordt beschouwd als een gegevensbestand, en de lineaire indexstructuur wordt vervangen door een **hiërarchie van indexen**.

   Deze wordt door een boomstructuur voorgesteld. Meestal doet men een beroep op **dynamisch gebalanceerde bomen** (B^+-tree) waarbij met elk knooppunt een blok van het indexbestand overeenkomt. Men zorgt ervoor dat elk blok minstens half gevuld blijft. Het blok overeenstemmend met de wortel van de structuur wordt permanent in het hoofdgeheugen opgeslagen.

   Elk blok bevat pointers naar de knooppunten op lager niveau van de hiërarchie. De blokken onderaan de hiërarchie verwijzen uiteindelijk naar de rijen in de gegevensbestanden. Omdat de boom gebalanceerd is, duurt het opzoeken van om het even welke gegevensrij telkens ongeveer even lang. De performantie daalt ook niet na herhaaldelijk wijzigen van de tabelgegevens.

4. **Hash index**. Hash gegevensbestanden maken het gebruik van extra indexstructuren vaak overbodig. Toch spreekt men vaak van hash indexen, omdat hash bestanden net als indexen gericht zijn op tabeltoegang via een specifieke combinatie van attributen. Hash indexen bieden doorgaans de snelste toegang bij het ophalen van een beperkt

aantal rijen.

5. **Bitmap indexen**. Biedt vooral voordelen in situaties waar het aantal mogelijke waarden van een attribuut relatief klein is. Voor elke rij houdt de index een bitmapveld bij, even lang als het aantal mogelijke waarden van het geïndexeerde attribuut. Elke bit in deze bitmap komt overeen met een specifieke waarde van het attribuut. Bitmapindexes kunnen beter gecomprimeerd worden, daardoor kunnen ze meestal compacter opgeslagen worden.
   Ideaal voor selecties
   - op kolommen met kleine **selectiviteit** (<1%)
   - met samengestelde predikaten op diverse kolommen

6. **join index (multi tabel index)**. Wordt vooral in datawarehouses gebruikt.
   Heeft een indexrecord voor elke combinatie van rijen uit 2 tabellen met dezelfde sleutelwaarde. Elke indexrecord bevat de sleutelwaarde en pointers naar beide corresponderende rijen. Ideaal voor queries die vaak uitgevoerd worden en die steeds dezelfde joins uitvoeren.

7. **Bitmap join index:** combinatie van bitmap index en join index.

6.4.3 Gebruik van indexen

Indexen kunnen de performantie van bepaalde queries aanzienlijk verbeteren.
Men moet echter een balans vinden tussen die potentiële performantieverbetering en de overhead die men ondervindt om de indexen te onderhouden bij wijziging van de gegevens (bij elke toevoeging/verwijdering van rijen moet men de tabel zelf en alle indexen bijwerken)
Elke index kost ook opslagruimte en de query optimizer moet meer alternatieve uitvoeringsstrategieën overwegen als er meer indexen zijn ➔ onzin om op elke kolom en op elke combinatie van kolommen een index te creëren.

Richtlijnen:

1. Indexen op kolomcombinaties moeten enkel overwogen worden
   - indien de kolommen frequent gebruikt worden in de **where-, group by- en order by-clausules**
   - bij gebruik van het **distinct** sleutelwoord
   - bij join predikaten.
     De meeste joins gebruiken de sleutels van beide tabellen, daarom is het dikwijls zinvol om niet alleen op primaire sleutels, maar ook op de verwijzende sleutels indexen te creëren.

   Indexen kunnen niet gebruikt worden voor selecties in de having-clausule.

2. Andere kolommen, vooral kolommen die men vaak wijzigt, worden best niet geïndexeerd, aangezien indexen daar globaal vertragend werken. In transactionele omgevingen, waar snelheid van updaten van gegevens belangrijk is, indexeert men best zo weinig mogelijk.

3. Niet alleen indexen die nauwelijks nut hebben voor queries, maar ook indexen die de query optimizer systematisch blijkt te negeren, moeten verwijderd worden.

4. De attribuutcombinatie waarop men het meest selecteert, of die het meest voorkomt in joins, kom in aanmerking als primaire of geclusterde index.

   In sommige gevallen kan men de primaire sleutel beter niet door een primaire index ondersteunen:
   - als de primaire sleutel een sequentiële surrogaatsleutel is zal daarop zelden geselecteerd worden in een select-clausule.

Dit leidt ook tot performantieproblemen bij insert-opdrachten

5. een index wordt **selectief** genoemd indien er relatief veel verschillende waarden zijn voor de geïndexeerde kolommen. De **selectiviteit** zou men kunnen uitdrukken als de verhouding van het aantal unieke indexwaarden tot het aantal rijen van de tabel.

   Niet geclusterde indexen met kleine selectiviteit hebben minder nut omdat er dan teveel rijen met dezelfde indexwaarde corresponderen. Het is dan efficiënter om de ganse tabel sequentieel te doorlopen. In grote tabellen (waar dit te lang zou duren) kan men beroep doen op bitmap indexen.

6. In een samengestelde index fungeert de meest selectieve kolom het best als eerste component, tenzij frequent gebruikte group by- en order by-clausules iets anders insinueren.

7. Indexering op kleine tabellen is doorgaans overbodig aangezien die dikwijls volledig in het geheugen doorzocht kunnen worden.

8. Indexen op combinaties van kolommen kunnen voordelig zijn:
   - Ze kunnen ook gebruikt worden voor selecties waarbij alleen de eerste kolom of kolommen van de index betrokken zijn.
   - Sommige queries kunnen genoeg informatie halen uit het indexbestand, zonder het gegevensbestand te moeten raadplegen.
     Van dergelijke **covered indexen** (ook **index-only plans** genoemd) maakt men dikwijls gebruik bij het opvragen van statistische informatie over kindtabellen.

9. Men moet de componenten van indexen met samengestelde sleutels steeds in de juiste volgorde aanspreken. Meerdere enkelvoudige indexen kunnen een beter alternatief zijn, indien men kan rekenen op de medewerking van de query optimizer om de indexen apart te raadplegen en toch efficiënt de doorsnede van de resultaattabellen te bekomen. Sommige query optimizers voeren hiervoor achter de schermen eque-joins van indexen uit, met de **record identifiers** in het join-predicaat. Deze **index joins** bieden dezelfde functionaliteit als **covered indexen**.

10. Men moet niet zozeer het aantal componenten, maar eerder de totale lengte van de indexsleutel zo klein mogelijk houden.

7. 3GL aspecten

7.1 Procedurele extensies

SQL-instructies worden meestal niet interactief uitgevoerd, maar opgenomen in een programma, geschreven in 1 of andere *host-programmeertaal.*
Vb:

- CLI: Call Level Interfaces zoals OLE DB en JDBC
- Embedded SQL

Voor veel combinaties van specifieke RDBMS'en en host-talen bestaat een **precompiler**. Deze converteert een programma dat uit een mix van host-taal en SQL instructies bestaat, naar een programma met enkel host-taal instructies, maar dat de SQL-instructies wel op een of andere manier kan uitvoeren. Sommige producten compileren de SQL opdrachten, sommige interpreteren ze.

Bijna elke RDBMS biedt een eigen hybride programmeeromgeving aan, waarin de declaratieve SQL syntax aangevuld kan worden met procedurele extensies.

vb:

- Oracle: PL/SQL
- SQL Server: Transact-SQL

Deze 3GL-omgevingen zijn in eerste instantie interessant om allerlei SQL opdrachten met elkaar te combineren tot een script. (in Oracle *anonymous block* genoemd). De scripts kunnen beschouwd worden als een verzameling 3GL opdrachten waarin met DML en DDL instructies kan inbedden.

Alle 3GL-omgevingen bieden faciliteiten om:

1. **host-variabelen** en **constanten** de definiëren en deze overal in de SQL opdrachten te gebruiken waar men uitdrukkingen mag specifiëren

2. het **resultaat** van select opdrachten in host-variabelen **op te slaan**.

3. Via ingebouwde systeemvariabelen **informatie** te bekomen over de **uitvoering** van ingebedde SQL instructies

4. **foutopvang** via *exception handlers* uit te voeren

5. de **code** aanwezig **in andere bestanden** met behulp van 1 enkele instructie over te nemen

6. de **code modulair** te construeren, o.a. door gebruik te maken van geneste blokken, of door *stored procedures* als subroutines te gebruiken

7. allerlei conditionele en iteratieve **controlestructuren** te gebruiken

Dankzij recente SQL toevoegingen, zoals case-expressies en recursieve CTE's, die men in 1 enkele SQL opdracht kan integreren, is er veel minder nood aan procedurele extensies dan voorheen.

Voordelen van relationele bewerkingen tov procedurele extensies

1. case-expressies en recursieve CTE's leiden tot een significant kleiner aantal SQL opdrachten, wat in de client-server concept tot een aanzienlijke beperking van de RTT leidt.
2. de query optimizer kan helpen om tot een globaal meer efficiëntere oplossingen te komen ➔ in een procedureel script kan de query optimizer enkel een bijdrage leveren op het beperkte niveau van de individuele SQL opdrachten
3. enkelvoudige SQL opdrachten zijn een orde efficiënter dan 3GL technieken, o.m. omdat ze gegevens zonder omwegen kunnen benaderen: gegevens moeten niet eerst in intermediaire datastructuren geladen worden om er bewerkingen op te verrichten.
4. Zoekmethodes en sorteeropdrachten worden optimaal uitgevoerd, zo veel mogelijk rekening houdend met de bestandsorganisatie van de gegevenstabellen, indices, geheugencaches,…

3GL-extensies zijn wel nuttig om:

1. enkele SQL opdrachten te verpakken tot 1 geheel in een script voor bv routinematige DBA-taken
2. eenvoudige transacties te ontwikkelen (zonder externe host-programmeertaal)
3. procedurele databankobjecten zoals stored procedures, triggers en events te creëren
4. cursors te gebruiken

7.2 Transacties

Een transactie is een atomaire, gebundelde reeks opeenvolgende SQL queries en wijzigingen, waarbij aan het einde met de opdrachten **commit** of **rollback** kan opgegeven worden of alle

wijzigingen permanent of ongedaan gemaakt moeten worden. Het begin van een transactie kan expliciet aangegeven worden met de **begin transaction** opdracht.

Transacties zijn vooral zinvol bij wijzigingen, waarbij men diverse tabellen moet aanpassen. Tijdens de uitvoering van een transactie zijn ongewenste inconsistenties best mogelijk. Aan het begin en na afloop van de transactie worden alle regels wel geacht in orde te zijn. Indien er tijdens de transactie iets fout gaat, dan kan de oorspronkelijke toestand hersteld worden door de commit opdracht te vervangen door rollback. Transacties zorgen voor de overgang van de ene consistente toestand van de databank naar een volgende consistente toestand. Een rollback onmiddellijk na een commit opdracht heeft geen enkele zin.

Men kan een transactie ook deels ongedaan maken, door **savepoints** te definiëren met de **save transaction** opdracht. In sommige RDBMS'en is het mogelijk om transacties te nesten. Door aan commit of rollback opdrachten een argument mee te geven, kan duidelijk gemaakt worden tot op welke savepoint of tot op welk niveau men de toestand wil bevestigen of herstellen.

7.2.1 Transactielogbestanden

Worden gebruikt om transacties eventueel ongedaan te kunnen maken. Transactieresultaten worden eerst naar het logbestand weggeschreven, vooraleer ze werkelijk in de databank verwerkt worden. De wijzigingen worden hierbij in chronologische volgorde opgeslagen. Het logbestand bevat ook **before images**, een kopie van de waarden van elke rij voor elke wijziging. Een transactie kan eenvoudig ongedaan gemaakt worden door de before images van alle wijzigingen opnieuw in de databank te laden.

Transactielogbestanden spelen ook een belangrijke rol bij het herstel van niet meer goed functionerende databanksystemen. Cruciaal hierbij is het periodiek maken van een volledige kopie (backup) van de databank en het bijhouden van de logbestanden sinds de laatste backup.

Bij **recovery via rollback** worden wijzigingen van gelogde transacties in omgekeerd chronologische volgorde ongedaan gemaakt.
Bij **recovery via rollforward** wordt de databank hersteld op basis van een backup en worden alle in de log verwerkte wijzigingen opnieuw uitgevoerd. De transactielog moet daarom ook **after images** bevatten, een kopie van alle rijen na elke wijziging.

Een backup wordt meestal maar eenmaal daags genomen. Een recovery via rollforward kan daarom ook heel wat tijd vergen. Om deze tijd te minimaliseren, synchroniseren veel RDBMS'en de databank en transactielog meer regelmatig: **checkpoints**. Bij dergelijke chechpoints worden nieuwe aanvragen tijdelijke tegengehouden, openstaande transacties verwerkt, en alle geheugenbuffers naar schijf weggeschreven.

Sommige RDBMS'en bieden transacties ook de mogelijkheid om bij elke uitvoering van een specifieke transactie een markering in het logbestand op te nemen. Het logbestand kan dan selectief hersteld worden tot een tijdstip net voor of net na een dergelijke markering.

7.2.2 Locking

Indien meerdere gebruikers of toepassingsprogramma's (X en Y) dezelfde gegevens gelijktijdig gebruiken, kan er heel wat mis gaan.

1. **dirty-write** probleem: X wijzigt gegevens, Y wijzigt diezelfde gegevens, één van beiden voert een rollback uit ➔ het is niet duidelijk op welke waarde men moet terugschakelen

2. **dirty-read** probleem: X wijzigt gegevens, Y vraagt deze op. X voert een rollback uit. Y gebruikt daarna gegevens die in feite nooit bestaan hebben

3. **non-repeatable reads**: X vraagt gegevens op, Y wijzigt deze gegevens. Als X deze gegevens opnieuw zou inlezen, zou worden vastgesteld dat de gegevens aangepast of

deels verwijderd zijn.

4. **Phantom reads**: X vraagt gegevens op, Y voegt rijen toe. Wanneer X de gegevens opnieuw opvraagt, stelt hij vast dat er rijen bijgekomen zijn.

5. **Lost update**: X vraagt gegevens op, Y vraagt diezelfde gegevens op. X baseert zich op zijn beeld om gegevens te wijzigen, Y ook.

Transacties zijn de bouwstenen om dergelijke problemen bij multi-user gebruik te voorkomen. Indien men transacties volledig serieel verwerkt, kunnen dergelijke problemen niet voorkomen. Het niveau van gelijktijdig gebruik ligt dan echter zeer laag. Daarom moeten transacties zoveel mogelijk parallel verwerkt worden.

Het mechanisme waarop men beroep doet om dit te bereiken is **selectie vergrendeling** of **locking**: als een transactie gegevens opvraagt of wijzigt, dan worden enkel deze gegevens voor andere transacties geblokkeerd. Lock-implementaties zouden ervoor moeten zorgen dat de inhoud van een databank na een parallelle uitvoering van een verzameling transacties precies dezelfde is als na seriële uitvoering van dezelfde transacties. Deze serialiseerbaarheid kan ondermeer gerealiseerd worden mbhv het tweefasen-lock-protocol (in de groeifase mogen transacties zoveel locks plaatsen als ze willen, maar van zodra de eerste lock vrijgegeven wordt, treedt de krimpfase in en kan de transactie geen andere lock meer verkrijgen)

De gebruiker of het toepassingsprogramma kan meestal geen locks expliciet plaatsen: binnen transacties stelt het DBMS automatisch impliciete locks in (bij Oracle wel: for update-clausule van de select opracht). Toepassingsprogramma's hebben hierop een indirecte invloed, door de transactiegrenzen af te bakenen. Transacties worden daarom best zo kort mogelijk gehouden, vooral in omgevingen met veel concurrente toepassingsprogramma's.

De hoeveelheid gegevens die geblokkeerd wordt, noemt men de **granulariteit** van de lock. Een kleinere granulariteit verhoogt de graad van parallelle uitvoering en zorgt voor minder congestieproblemen, maar maakt het beheer van locks voor het RDBMS aanzienlijk moeilijker.

Een andere DBMS instelling die bepalend is voor het functioneren van locks is het gewenste **isolation level**. Het isolation level bepaalt in hoeverre transacties beïnvloedt mogen worden door gelijktijdige transacties die dezelfde gegevens manipuleren. Het isolation level is instelbaar per sessie, per transactie of per SQL opdracht ingesteld worden op 1 van de volgende niveaus:

1. **serializable**: transacties worden seriëel verwerkt ➔ maximale afscherming van gebruikers/toepassingsprogramma's. Meest pessimistische lockingstrategie: er wordt van uit gegaan dat er waarschijnlijk conflicten optreden.

2. **repeatable read**: bij het opvragen van gegevens wordt een **share** of **read lock** ingesteld. Andere transacties, die enkele een select opdracht uitvoeren op deze gegevens worden niet geblokkeerd. Bij het wijzigen van de gegevens wordt een **write lock** of **exclusive lock** ingesteld. De locks worden opgeheven wanneer de transactie eindigt (commit)
   Phantom reads mogelijk.

3. **Read commited** of **cursor stability**: gelijkaardig aan repeatable reads, maar de share locks worden reeds vrijgegeven na verwerking van de select opdracht. Dit is het standaard isolatieniveau van vee RDBMS'en
   Nonrepeatable reads mogelijk.

4. **Read uncommited**: ook exclusive locks worden onmiddellijk na de wijziging opgegeheven. Meest optimistische lockingstrategie: de kans op conflicten wordt klein verondersteld. Op dit niveau is de parallelliteit van uitvoering optimaal.
   Dirty reads mogelijk.

Het RDBMS moet ervoor zorgen dat **deadlocks**, die kunnen optreden van zodra men exclusive locks gebruikt, gedetecteerd en opgeheven worden door in dergelijke situatie voor 1 of meerdere transacties een rollback te forceren.

De rol die transacties spelen bij multi-user gebruik worden vaak samengevat met de acronycm ACID: Atomair, Consistent, wederzijdse Isolatie, Duurzaam (duidt erop dat het DMBS alle wijzigingen van afgewerkte transacties als permanent moet kunnen beschouwen)

Stored procedures

Een stored procedure bestaat uit SQL opdrachten, aangevuld met de procedurele extenties van de specifieke RDBMS.

- Worden in tegenstelling tot scripts permanent in de databank zelf opgeslagen, als objecten in de systeemcatalogus
- zijn in de SQL standaard opgenomen, maar de syntax en mogelijkheden verschillen sterk bij verschillende implementaties
- invoer/uitvoer-parameters zijn mogelijk ➔ ook lokale variabelen kunnen gedefinieerd worden
  → deze parameters en lokale variabelen kunnen overal voorkomen waar uitdrukkingen verwacht worden
- Bij de uitvoering van een **create procedure** opdracht wordt de syntax van de code en het bestaan van de objecten (tabellen, kolommen,...) gecontroleerd en wordt de code in de systeemcatalogus opgeslagen, maar nog niet uitgevoerd
- Om de performantie te verhogen, ondergaat de code meestal 1 of andere vorm van binding of compilatie (die o.m. bepaalt met welke verwerkingsstrategie de procedure zal uitgevoerd worden.)
- stored procedures worden expliciet opgeroepen, vanuit een toepassingsprogramma, vanuit een andere of (recursief) vanuit dezelfde stored procedure, of vanuit een trigger ➔ hiervoor wordt een execute-opdracht of iets analoogs gebruikt

Voordelen van stored procedures

1. mogelijkheid om gedeelten van het toepassingsprogramma centraal in de databank op te slaan. Hierdoor zijn ze meteen aanroepbaar vanuit andere toepassingsprogramma's. ➔ centrale opslag dwingt niet alleen modulariteit van de toepassingsprogramma's af, maar vereenvoudigt ook het onderhoud ervan
2. periodiek uitvoeren van bundels opdrachten door bv de task sheduler van het OS
3. ze kunnen opgeroepen worden vanuit om het even welke host-programmeertaal of host-omgeving
4. met elke RDBMS worden standaard een hele reeks system **stored procedures** meegeleverd. Hierdoor kunnen een aantal beheerstaken aanzienlijk vereenvoudigd worden (beveiliging, replicatie, performantiemonitoring)
5. de DBA kan vastleggen welke gebruikers welke stored procedures kunnen uitvoeren
6. tijdens het uitvoeren van een stored procedure is er geen enkele communicatie tussen het toepassingsprogramma en de RDBMS
7. door voorafgaande compilatie is de uitvoering van individuele SQL-opdrachten dikwijls sneller als men ze via een stored procedure uitvoert. De query optimizer hoeft bij

uitvoering de optimale strategie niet meer te bepalen.

Sommige RDBMS'en maken het mogelijk om een verzameling stored procedures modulair te groeperen in **stored packages**.
Packages maken ook **overloading** mogelijk: een package kan diverse stored procedures met dezelfde naam, maar met een andere parameterlijst bevatten.
Door de SQL-standaard worden packages PSM's genoemd (**persistent stored procedures**)

De meeste RDBMS'en ondersteunen een bijzondere vorm van stored procedure die zich laat gebruiken als een door de ontwikkelaar gedefinieerde **functie**: dergelijke procedures kunnen alleen invoerparameters hebben en fungeren zelf als uitvoerparameter.

7.4 Triggers

Een trigger is net als een stored procedure een procedureel object dat men in de systeemcatalogus opslaat. Stored procedures worden echter passief uitgevoerd, wanneer een gebruikersprogramma daar expliciet om vraagt. Triggers kunnen automatisch uitgevoerd worden bij bepaalde bewerkingen.

Er zijn verschillende soorten triggers:
1. **DML-triggers**: worden uitgevoerd als reactie op insert, delete of update opdrachten
2. **DDL-triggers**: worden uitgevoerd als reactie op create, alter en drop opdrachten

Triggers worden gecreëerd en verwijderd met **create trigger** en **drop trigger** opdrachten.
Triggers zijn gekoppeld aan een specifieke tabel: wordt de tabel verwijderd, dan verdwijnen ook de triggers.

Triggerdefinities bestaan volgens het **ECA-model** (Event-Condition-Action) uit 3 componenten:

1. **triggerevent:** bepaalt bij welke specifieke SQL-opdracht (**de triggering instructie**) op welke tabel (**triggering tabel**) de trigger geactiveerd wordt. Ook moet opgegeven worden of de trigger voor, na, of ter vervanging van de SQL instructie uitgevoerd zal worden ➔ **before triggers**, **after triggers** en **instead of triggers**

2. **triggeractie**: bepaalt wat de trigger precies doet. De triggeractie is het equivalent van een stored procedure en beschikt over dezelfde mogelijkheden. De triggeractie hoeft niet beperkt te blijven tot de triggering tabel, maar mag ook stored procedures aanroepen. De triggeractie zelf kan, eventueel recursief, het activeren van triggers tot gevolg hebben.

3. **triggerconditie** (optioneel) beperkt de triggeractie tot rijen die aan specifieke voorwaarden voldoen. Deze voorwaarden moeten niet beperkt blijven tot de kolommen van de triggering tabel.

Triggers worden op niveau van SQL afgevuurd: de triggeractie wordt slecht éénmaal uitgevoerd, ook al passen de insert, delete of update opdrachten meerdere rijen aan. Men moet in de triggercode zelf zien te achterhalen welke aanpassingen allemaal aan de orde zijn. Hiervoor bieden de RDBMS'en allerlei hulptabellen, variabelen en functies aan.

Vb: in SQL server:
- **pseudotabellen** inserted en deleted, die kopieën bevatten van de toegevoegde en verwijderde rijen.
- **Functies** update en columns_updated die kunnen aangeven welke specifieke kolommen door een update opdracht zijn beïnvloed.

Vb: in Oracle:
- men kan bij definitie van het triggerevent bepalen dat de triggeractie voor elke individuele gewijzigde rij moet uitgevoerd worden mbhv de **for each row** clausule.

Dergelijke triggers worden **rijtriggers** genoemd.
- De **pseudorecords** old en new zijn dan de gegevensstructuren die de inhouden van de rij net voor en onmiddellijk na de triggering instructie bevatten.

7.4.1 Before triggers

Constraints zijn de aangewezen mechanismen om beperkingsregels en gedragsregels door het RDBMS te laten controleren. Constraints zijn echter vrij beperkt in functionaliteit. De meeste RDBMS'en laten bijvoorbeeld in check-constraints geen joins toe, of gecorreleerde subqueries. Constraints zijn o.m. niet in staat om **cardinaliteitsregels** af te dwingen.

Dankzij before triggers kan men om het even welke integriteitsregel in de databank opslaan en bij elke poging tot wijziging van de gegevens toetsen.

Triggers zijn minder efficiënt dan constraints
Constraints hebben voorrang op triggers: de triggeracties van wijzigingen die met constraints conflicten geven, worden nooit uitgevoerd.

7.4.2 After triggers

Voldoen uitstekend om de redundante informatie in gedenormaliseerde tabellen automatisch te corrigeren. De triggercode kan immers om het even welke SQL opdrachten uitvoeren, en bijvoorbeeld ook opnieuw een update opdracht toepassen op de rijen, die voor het activeren van de trigger gezorgd hebben.

Kunnen gebruikt worden voor het aanpassen en dupliceren van meerdere tabellen.

Triggers kunnen in sommige RDBMS implementaties recursief zichzelf activeren. Hiervan kan ondermeer gebruik gemaakt worden om in een hiërarchie volgens het **adjacency model**, het wijzigen van bepaalde waarden ook een effect te laten hebben op de corresponderend waarden van de ouderknooppunten.

Ook het automatisch archiveren van te verwijderen rijen kan men met 1 enkele regel triggercode realiseren. Het loggen van wie en wanneer specifieke bewerkingen uitgevoerd heeft (**audits**) behoort ook tot de mogelijkheden.

7.4.3 Instead of triggers

vooral gebruikt om de restricties te versoepelen waaraan het wijzigen van views en CTE's normaal onderworpen is. (omzeilen beperkingen van wijzigbare views)
Dikwijls gebruikt om invoer van bepaalde tabellen om te leiden naar andere tabellen, soms zelfs naar andere databankservers toe.

7.5 Cursors

Om het resultaat van select-opdrachten die meerdere rijen kan opleveren op te slaan, gebruikt met **cursors**.
Een cursor wordt gedefinieerd met de **declare** opdracht, door er een willekeurige select opdracht aan te koppelen met een zo restrictief mogelijke where-clausule. De opdracht wordt op dat moment nog niet uitgevoerd, maar wel bij de **open** instructie.
Een gebruikersprogramma mag diverse cursors tegelijkertijd openen.
Het gebruikersprogramma kan de **fetch** opdracht gebruiken om de rijen 1 voor 1 te doorlopen en de kolomwaarden ervan aan host-variabelen toe te kennen. Telkens is er 1 enkele rij ter

beschikking: de cursor gedraagt zicht als een pointer naar een specifieke rij van de resultaattabel.

Na de **close** opdracht verwijdert het RDBMS de resultaattabel en eventuele locks. Dit sluiten gebeurt best zo snel mogelijk, om het geheugen op serverniveau niet onnodig vast te houden. Dezelfde cursor kan vervolgens opnieuw geopend worden in hetzelfde programma. Met de **deallocate** opdracht worden ook de buffers op cliënt niveau verwijderd.

Ter vervanging van het (default) **next** sleutelwoord, aanvaardt de **fetch** opdracht ook parameters zoals **prior**, **first** en **last** waarmee men resp. de resultaattabel in omgekeerde zijn, opnieuw vanaf het begin of vanaf het einde kan doorlopen. Ook een willekeurige rij kan opgehaald worden door het vermelden van een absolute positie of relatieve positie tov de laatst behandelde rij.

Ook het wijzigen van kolomwaarden en het verwijderen van rijen is soms mogelijk, met een aangepaste versie van de **update** en **delete** opdrachten. Dit wordt **positioneel updaten** genoemd.

De manier waarop het RDBMS de resultaattabel buffert, is afhankelijk van de implementatie:

1. Slechts één enkele rij uit de databank halen, telkens men daar specifiek om vraagt.
   + Efficiënt als maar weinig rijen opgehaald worden.
   - Verhoogt de belasting op het netwerk?

2. De volledige resultaattabel bufferen.

3. Tussenoplossingen bufferen een beperkt aantal rijen, en laten ook de cliënt cachen, om het verkeer tussen gebruikersprogramma en RDBMS zo veel mogelijk te beperken.

Afhankelijk van de methode ziet de toepassing de gegevens in hun actuele toestand (**dynamische cursors**) of zoals ze eruit zagen toen de cursor geopend werd (**statische cursors**).

Nadelen

Het gebruik van cursors heeft een negatieve impact op de verwerkingstijd en de geheugenruimte, en kan tot globale performantieproblemen van het RDBMS leiden. Cursors laten ontwikkelaars toe om de databanktabellen als sequentiële bestanden te misbruiken.

Toch gebruiken om

Het gebruik van cursors is in een beperkt aantal omstandigheden redelijk te verantwoorden:

1. Doorgeven van gegevens tussen stored procedures en een gebruikersprogramma. Standaard ondersteunen stored procedures enkel scalaire waarden als uitvoerparameters. In de praktijk blijkt dat het doorgeven van een cursor als uitvoerparameter een geschikte oplossing is hiervoor.

2. In de context van wat men soms **dynamisch SQL** noemt. Het gebruikersprogramma maakt dan pas tijdens de uitvoering de code van SQL opdrachten aan.
   vb: bij het **pivoteren**: de kolommen, naamgeving en aantal werden statisch gecodeerd in de SQL opdracht. Voor een groot of variabel aantal kolommen biedt dit echter geen oplossing. De **fetch** opdrachten zorgen precies voor de meest eenvoudige manier om de code voor elk van de kolommen te genereren.