

Inhoudsopgave: Datum:

[1 Hoofdstuk 1: SQL Review 3](#_Toc186116541)

[1.1.1 SELECT: 3](#_Toc186116542)

[2 Hoofdstuk 2: SQL-Advanced 6](#_Toc186116543)

[2.1 Subqueries 6](#_Toc186116544)

[2.2 DML-tasks 7](#_Toc186116545)

[2.2.1 INSERT 7](#_Toc186116546)

[2.2.2 UPDATE 7](#_Toc186116547)

[2.2.3 DELETE 7](#_Toc186116548)

[2.3 VIEWS 8](#_Toc186116549)

[2.3.1 Common Table Expression (CTE) 8](#_Toc186116550)

[3 Hoofdstuk 3: Window Functions 10](#_Toc186116551)

[3.1 OVER-clause 10](#_Toc186116552)

[3.2 Window Functions 12](#_Toc186116553)

[4 Hoofdstuk 4: Programming in SQL, Cursors en Triggers 13](#_Toc186116554)

[4.1 Programming 13](#_Toc186116555)

[4.1.1 Stored Procedure 13](#_Toc186116556)

[4.1.2 Error afhandeling 13](#_Toc186116557)

[4.1.3 Oproepen van een stored procedure 14](#_Toc186116558)

[4.2 Functions 14](#_Toc186116559)

[4.3 Cursors 15](#_Toc186116560)

[4.4 Triggers 15](#_Toc186116561)

[5 Hoofdstuk 5: Indexen en performantie 17](#_Toc186116562)

[6 Hoofdstuk 6: Basics of Transaction Management 22](#_Toc186116563)

[6.1 Introduction 22](#_Toc186116564)

[6.2 Transactions and Transaction Management 22](#_Toc186116565)

[6.3 Recovery 23](#_Toc186116566)

[6.4 Concurrency Control 25](#_Toc186116567)

[6.5 Locking and Locking Protocols 26](#_Toc186116568)

[7 Hoofdstuk 7: Datawarehousing 30](#_Toc186116569)

[7.1 Design 30](#_Toc186116570)

[7.2 Kleine samenvatting 36](#_Toc186116571)

# Hoofdstuk 1: SQL Review

SQL bestaat uit 3 subtalen:

* **Data Definition Language (DDL)**
  + Aanmaken van een database, definiëren van databaseobjecten (tabellen, stored procedures, weergaven,...)
  + CREATE, ALTER, DROP
* **Data Manipulation Language (DML)**
  + Gegevens in een database opvragen en manipuleren
  + SELECT, INSERT, UPDATE, DELETE
* **Data Control Language (DCL)**
  + Gegevensbeveiliging en autorisatie, (rollen toekennen aan mensen bv)
  + GRANT, REVOKE, DENY

### SELECT:

Afbeelding met tekst, schermopname, Lettertype, lijn

Automatisch gegenereerde beschrijvingAlle mogelijke opties bij een SELECT:

* GROUP BY : groeperen van gegevens
* HAVING-clausule: filtervoorwaarde op groepen

Speciale karakters:

* % → willekeurige reeks van 0, 1 of meer tekens (maakt kortom niet uit wat er achter komt)
* \_ → 1 teken
* [ ] → 1 teken in een opgegeven bereik
* ^] → elk teken buiten het opgegeven bereik
* Voorbeeld: toon elk product waarvan de 2e letter a-k bevat en al de rest maakt niets uit:
  + WHERE productname LIKE '\_[a-k]%'

IN, NOT IN:

* Hier geef je een **array** aan voorwaarden mee:
  + WHERE prijs in (1.50, 3.00)

BETWEEN:

* Geeft resultaten weer **tussen** opgegeven waarden
  + WHERE prijs BETWEEN 0.50 AND 10.00

String functies:

Afbeelding met tekst, schermopname, Lettertype, nummer

Automatisch gegenereerde beschrijving

Datum functies:

Afbeelding met tekst, schermopname, Lettertype, nummer

Automatisch gegenereerde beschrijving

Wiskundige functies:

Afbeelding met tekst, schermopname, Lettertype, lijn

Automatisch gegenereerde beschrijving

CASE-functie (dit wordt Meestal niet gebruikt omdat dit server-specifiek is):

CASE

WHEN unitprice IS NULL THEN 'Not yet priced'

WHEN unitprice < 10 THEN 'Very Reasonable Price'

WHEN unitprice >= 10 and unitprice < 20 THEN 'Affordable'

ELSE 'Expensive!'

END AS 'Price Category' – **naam van de kolom**

Ingebouwde functies:

* SUM
* AVG
* MIN
* MAX
* COUNT
  + COUNT(\*) geeft alle SELECT-rijen mee
  + COUNT(kolomnaam) geeft alle NOT NULL rijen mee

TOP:

Geeft een beperkte hoeveelheid mee aan welke rijen je wil laten zien

GROUP BY:

* Maakt groepjes van de SELECT-statement.
* HAVING is filteren op die groepjes (zoals de WHERE)

Afbeelding met tekst, cirkel, diagram, schermopname

Automatisch gegenereerde beschrijvingJOINS:

* LEFT JOIN = zonder NULL-waarden
* RIGHT JOIN = enkel NULL-waarden

INTERSECT:

Welke records bevinden zich in de kruising?

SELECT City, Country FROM Customers

INTERSECT

SELECT City, Country FROM Suppliers

EXCEPT:

Toont de waarden die in de éne tabel staan, maar in de andere niet.

SELECT customerid

FROM Customers

EXCEPT

SELECT customerid

FROM Orders

# Hoofdstuk 2: SQL-Advanced

## Subqueries

Subqueries:

* Dit gebruik je vooral om functies te gebruiken, zoals

**Select** \*

**From** products p

**Where** p.unitprice = (select max(p.unitprice) from products p)

* De subqeuery moet niet per se in de WHERE-clause staan

JOIN:

Enkele subqueries kunnen worden vervangen met JOINS:

**SELECT** \*

**FROM** Customers

**WHERE** customerid NOT IN (SELECT DISTINCT customerid FROM Orders)

**SELECT** \*

**FROM** Customers c LEFT JOIN Orders o ON c.customerid = o.customerid

**WHERE** o.customerid is NULL

Nadeel subqueries:

* De subquery wordt uitgevoerd voor elke rij in de hoofdquery. => O(n²)
* De volgorde van uitvoering is van boven naar beneden, niet van beneden naar boven zoals in een eenvoudige subquery, die O(n) is.

Complexere subquerie (dat uiteindelijk het principe van een CTE wordt):

Example: Give the employees whose salary is larger than the average of the salary of the employees who report to the same boss.

**SELECT** firstname + ' ' + lastname As fullname, reportsto, Salary

**FROM** Employees As e

**WHERE** Salary > (SELECT AVG(Salary) FROM Employees WHERE reportsto = e.reportsto)

EXISTS operator:

**SELECT** \*

**FROM** Customers As c

**WHERE** EXISTS (SELECT \* FROM Orders WHERE customerid = c.customerid)

Performantie:

* Je kan dus enkele problemen op 3 verschillende manieren oplossen (gesorteerd op beste performantie tot slechtste):
  + JOIN
  + Subquery
  + Correlated subquery

## DML-tasks

### INSERT

Inserten via parameters:

INSERT INTO Products (productname, categoryid, Discontinued)

VALUES ('Toblerone', 3, 0)

Inserten zonder parameters:

Zonder parameters neemt het enkel de NOT NULL waarden op

INSERT INTO Products

VALUES ('Sultana', null, 3, null, null, null, null, null, 1)

Inserts via data van andere tabellen:

**INSERT INTO Customers** (customerid, contactname, contacttitle, companyname)

**SELECT** substring(firstname,1,2) + substring(lastname,1,3),

Firstname + ' ' + lastname,

Title,

'Employeecompany'

**FROM** Employees

### UPDATE

Algemene syntax:

**UPDATE** Products

**SET** unitprice = unitprice \* 1.1

Met subquery:

**UPDATE** Products

**SET** unitprice = (unitprice \* 1.1)

**WHERE** supplierid IN (SELECT supplierid FROM Suppliers WHERE Country = 'USA')

### DELETE

Algemene syntax:

**DELETE** FROM Products

**WHERE** productname LIKE '%Bröd%'

Verschil tussen delete en truncate:

* DELETE FROM Products
  + Dit houdt rekening met WHERES, TRIGGERS etc …
* TRUNCATE TABLE Products
  + Dit verwijderd alles in één klap en houdt met niks rekening, maar is wel sneller

Met subquery:

**DELETE** FROM orderdetails

**WHERE** orderid IN (SELECT orderid FROM Orders WHERE orderdate = (SELECT MAX(orderdate) from Orders))

Tijdelijke table en opvullen ervan:

**DROP** TABLE IF EXISTS shippersupdate;

**SELECT** \* INTO shippersupdate FROM Shippers

## VIEWS

Views:

* Een view is een opgeslagen SELECT statement
* Een view kan gezien worden als een virtuele tabel samengesteld uit andere tabellen & views
* Er worden geen gegevens opgeslagen in de view zelf, bij elke verwijzing wordt de onderliggende SELECT opnieuw uitgevoerd
* Views worden gemaakt om code makkelijker op te delen zodat de leesbaarheid hoger is en omdat het performanter is.

View aanmaken:

Dit **blijft beschikbaar** in je hele database

**CREATE VIEW** V\_productscustomer(productcode, company, quantity)

AS (

**SELECT** od.productid, c.companyname, sum(od.Quantity)

**FROM** Customers c

**JOIN** Orders o ON o.customerid = c.customerid

**JOIN** orderdetails od ON o.orderid = od.orderid

**GROUP BY** od.productid, c.companyname;

)

Gebruik maken van een view:

**SELECT** \* FROM V\_productscustomer

Of via een JOIN

Veranderen van een view:

Verander simpelweg de structuur van de code en het is zo simpel als dat

**CREATE VIEW** V\_productscustomer(productcode, company, quantity)

AS (

**SELECT** od.productid, c.companyname

**FROM** Customers c

**JOIN** Orders o ON o.customerid = c.customerid

**JOIN** orderdetails od ON o.orderid = od.orderid

**GROUP BY** od.productid, c.companyname;

)

Verwijderen van een view:

**DROP** VIEW V\_productscustomer;

### Common Table Expression (CTE)

CTE:

* Dit is enkel bedoelt als ‘functie’ binnenin je sql-statement en wordt niet in de database opgeslagen.
* Het heeft als doel om code overzichtelijker en makkelijker te maken.

Recursie:

* Dit is een heel simpel voorbeeld maar het illustreert perfect hoe recursie hier werkt.
* Je start met een basiswaarde (select 1) en verhoogt de waarde via de WHERE-clause en combineert dat met een UNION ALL

**WITH** numbers(number)

AS

(**SELECT** 1

**UNION** all

**SELECT** number + 1

**FROM** numbers

**WHERE** number < 5)

Hiërarchische structuur:

* Een hierarchische structuur is **beginnen vanaf de** **top** en zo langzaam aan alles weergeven tot als je aan de **bodem** bent
* Dit kan in sql ook met de volgende stappen:

**-- 1. Maken van de CTE**

WITH recursieveTitels AS (

**-- 2. Selecteer alle informatie die je nodig hebt**

SELECT e1.EmployeeID, e1.FirstName + ' ' + e1.LastName as EmployeeName, e1.Reportsto,

1 as Level, **-- 3. Toon op welk level je zit**

cast(e1.Title AS VARCHAR(max)) as Path **-- 4. Maak de basis van je pad**

FROM Employees e1

WHERE Reportsto IS NULL **-- 5. Selecteer iedereen op de bovenste rang**

UNION ALL **-- 6. Combineer de bovenste en onderste select**

SELECT e2.EmployeeID, e2.FirstName + ' ' + e2.LastName as EmployeeName, e2.Reportsto,

cte.Level + 1 as Level, **-- 8 Verhoog het pad met +1**

CAST(cte.Path + ' <-- ' + e2.Title as varchar(max)) as Path **-- 9 Pak het Path van het vorige deel van de CTE en zet er cte.Path + ' <-- ' + voor**

FROM Employees e2

JOIN recursieveTitels cte on e2.ReportsTo = cte.EmployeeID **-- 7 Join het vorige deel van de CTE op de nieuwe query. De on-voorwaarde is het recursieve gedeelte dat in je pad terechtkomt**

)

SELECT rt.EmployeeID, rt.EmployeeName, rt.Level, rt.PATH

FROM recursieveTitels rt

ORDER by rt.EmployeeID

# Hoofdstuk 3: Window Functions

## OVER-clause

Window functions: business case:

* Vaak willen bedrijfsmanagers de huidige verkoop vergelijken met eerdere verkopen.
* Eerdere verkopen kunnen zijn
  + Verkopen in de vorige maand
  + Gemiddelde omzet in de afgelopen drie maanden -.
  + Verkopen van vorig jaar tot de huidige datum (year-to-date)
* Ze bieden een oplossing voor dit soort problemen in een enkele, efficiënte SQL-query

OVER clause:

* Resultaten van een SELECT worden gepartitioneerd
* Nummering, ordening en aggregaat functies per partitie
* De OVER-clausule creëert partities en ordening
* De partitie gedraagt zich als een venster dat over de gegevens schuift
* De OVER-clausule kan worden gebruikt met standaard aggregaat functies (som, avg, ...) Of specifieke vensterfuncties (rang, lag, ...)

Voorbeeld totaal, geven via OVER:

* Voeg een extra kolom toe om het doorlopende totaal van unitsinstock per categorie te berekenen.
* Oplossing 2 → OVER-clausule
  + Eenvoudiger + efficiënter
  + De som wordt berekend voor elke partitie

**SELECT** categoryid, productid, unitsinstock,

**SUM**(unitsinstock) OVER (PARTITION BY categoryid ORDER BY productid) AS totalunitsinstockpercategory

**FROM** Products

Afbeelding met tekst, schermopname, nummer, Lettertype

Automatisch gegenereerde beschrijvingVerschil tussen PARTITION BY en zonder PARITITON BY:

* In de screenshot zie je dat er meerdere CategorieIds hetzelfde id 1 bevatten.
* Als je dan de overtijdse sommering wilt schrijf je PARTITION BY categoryid
* Als elk ID anders is zoals onderstaande query, dan hoef je geen PARTITION BY te gebruiken

**SELECT** e.EmployeeID, e.FirstName + ' ' + e.LastName as FullName,

SUM (e.EmployeeID) over (order by e.EmployeeID) as overtijdseSommering

**FROM** Employees e

Wat met twee ORDER BY’s?

* De normale ORDER BY zal voorrang krijgen op de ORDER BY in de window function
* Maar de manier hoe de window function telt zal nogsteeds hetzelfde blijven

**SELECT** e.EmployeeID, e.FirstName + ' ' + e.LastName as FullName,

SUM (e.EmployeeID) over (order by e.EmployeeID) as overtijdseSommering

**FROM** Employees e

**ORDER BY** e.EmployeeID

RANGE:

* Betekenis van windowfunction: toepassen op een window dat over de resultatenset verschuift
* De vorige query werkt met het standaardvenster: begin van de resultatenset tot de huidige rij
* Je hebt 3 opties:
  + RANGE **BETWEEN UNBOUNDED PRECEDING** AND CURRENT ROW
  + RANGE **BETWEEN CURRENT ROW** AND UNBOUNDED FOLLOWING
  + RANGE **BETWEEN UNBOUNDED PRECEDING** **AND UNBOUNDED FOLLOWING**
* **PARTITION is optioneel, ORDER BY is verplicht**

UNBOUDED PRECEDING:

Geeft de som van iets over tijd heen: 39, 56, 76 … 559

CURRENT ROW:

Geeft eerst het totaal weer en trekt een bepaalde hoeveelheid steeds af 559, 502, 377 … 39

UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING:

Geeft meteen het totaal weer

ROWS:

* When you use RANGE, the current row is compared to other rows and grouped based on the ORDER BY predicate.
* This is not always desirable. You might actually want a physical offset.
* In this scenario, you would specify ROWS instead of RANGE. Thisgives you three options in addition to the three options enumerated previously:
  + ROWS BETWEEN N PRECEDING AND CURRENT ROW
  + ROWS BETWEEN CURRENT ROW AND N FOLLOWING
  + ROWS BETWEEN N PRECEDING AND N FOLLOWING

Voorbeelden:

Maak een overzicht van het salaris per werknemer en het gemiddelde salaris van deze werknemer en de 2 werknemers voor hem.

**SELECT** employeeid, firstname + ' ' + lastname As fullname, Salary,

AVG(Salary) OVER (ORDER BY Salary DESC ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) As avgsalary2preceding

**FROM** Employees

Maak een overzicht van het salaris per werknemer en het gemiddelde salaris van deze werknemer en de 2 werknemers die hem volgen.

**SELECT** employeeid, firstname + ' ' + lastname As fullname, Salary,

AVG(Salary) OVER (ORDER BY Salary DESC ROWS BETWEEN CURRENT ROW AND 2 FOLLOWING) As avgsalary2following

**FROM** Employees

Voorbeelden vervolg:

Maak een overzicht van het salaris per werknemer en het gemiddelde salaris van deze werknemer en de werknemer voor en na hem.

**SELECT** employeeid, firstname + ' ' + lastname As fullname, Salary,

AVG(Salary) OVER (ORDER BY Salary DESC ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) As avgsalary1preceding1following

**FROM** Employees

## Window Functions

Window functions:

* **ROW\_NUMBER()** nummert de uitvoer van een resultatenset. Geeft meer specifiek het volgnummer van een rij binnen een partitie van een resultatenset terug, beginnend bij 1 voor de eerste rij in elke partitie.
* **RANK()** geeft de rang van elke rij binnen de partitie van een resultatenset terug. De rang van een rij is één plus het aantal rijen dat voor de rij in kwestie komt.
* **ROW\_NUMBER** en RANK zijn vergelijkbaar. ROW\_NUMBER nummert alle rijen opeenvolgend (bijvoorbeeld 1, 2, 3, 4, 5). RANK geeft dezelfde numerieke waarde voor gelijke rijen (bijvoorbeeld 1, 2, 2, 4, 5).
* **DENSE\_RANK()** geeft de rang van elke rij binnen de partitie van een verzameling resultaten, zonder gaten in de rangschikkingswaarden (bijvoorbeeld 1, 2, 2, 3, 4).
* **PCT\_RANK(**) geeft de rangorde weer op een schaal van 0 – 1
* **Samenvatting** eigenlijk is dit niets meer dan een simpele rijnummering op verschillende manieren.

LAG en LEAD theorie:

* **LAG** verwijst naar de vorige regel. Dit is een afkorting voor LAG(..., 1)
* **LAG** (..., 2) verwijst naar de regel voor de vorige regel, ...
* **LEAD** verwijst naar de volgende regel. Dit is een afkorting van LEAD(..., 1)
* **LEAD** (..., 2) verwijst naar de regel na de volgende regel, ...

LAG voorbeeld:

**WITH** cte

AS (

**SELECT** employeeid, firstname + ' ' + lastname As fullname, Salary, LAG(Salary) OVER (ORDER BY Salary DESC) As precedingsalary

**FROM** Employees )

**SELECT** \*, precedingsalary - Salary As earnslessthanpreceding

**FROM** cte

LEAD voorbeeld:

**WITH** cte

AS (

**SELECT** employeeid, firstname + ' ' + lastname As fullname, Salary, LEAD(Salary) OVER (ORDER BY Salary DESC) As forwardinggsalary

**FROM** Employees )

**SELECT** \*, forwardinggsalary - Salary As earnsmorethanforwardhing

**FROM** cte

# Hoofdstuk 4: Programming in SQL, Cursors en Triggers

## Programming

### Stored Procedure

Stored procedure:

* Een opgeslagen procedure is een benoemde verzameling SQL- en control-of-flow-opdrachten (programma) die is opgeslagen als een databaseobject.
* Het werkt als een functie zoals Javascript maar niet helemaal, je kan het bv. Gebruiken om insert-, delete- of updatefuncties te maken.

Variabelen:

* Hiervoor heb je variabelen nodig, in javascript kan de declaratie en initialisatie op één lijn gebeuren, hier niet.
* Hier moet je explicitie:
  + **DECLARE** @name varchar(50), @secondname money …
  + **SET** @name = ‘Lente’, @secondname = select naamseizoen from x where seizoendatum = ‘dddd/mm/yy’
* In de output (bv. Via een PRINT-statement) moet je ints converteren naar een STR()

Parameters:

* Met een invoerparameter wordt een parameter doorgegeven aan de SP
* Met een output kun je mogelijk een waarde doorgeven aan de SP en een waarde terugkrijgen

### Error afhandeling

Throw:

CREATE OR ALTER PROCEDURE deleteid

@id INT

AS

BEGIN

DECLARE @msg NVARCHAR(100);

IF @id IS NULL

BEGIN

SET @msg = N'The ID cannot be NULL.';

THROW 50001, @msg, 1;

END

DELETE FROM Tabel

WHERE tabelrij = @id;

IF @@ROWCOUNT = 0

BEGIN

SET @msg = 'No record found with the specified ID.';

THROW 50002, @msg, 1;

END

END;

### Oproepen van een stored procedure

Begin transaction

Declare @companynamee NVARCHAR(40), @Num int;

Begin try

Exec contactcustomers @companynamee, str(@Num)

Set @companynamee = 'Jan Claes'

Print 'Aantal klanten: ' + str(@Num)

End try begin catch end catch

End catch

Dit is hoe een algemene stored procedure er uit ziet, als je dit snapt heb je al veel mee:

Afbeelding met tekst, software, Lettertype, Webpagina

Automatisch gegenereerde beschrijving

## Functions

Functie:

* Zowel functies als stored procedures worden opgeslagen in de database maar er zijn een paar verschillen
* Hier moet je bv. Specificeren wat de functie retourneert.
* Je kan een functie aanroepen binnenin een select-statement
  + Dbo.naamfunctie(naamkolom) as berekennetto
* Je kan hem ook oproepen via een join, maar dit moet niet

Structuur:

CREATE OR ALTER FUNCTION calculatenettopermonth (@salary Money)

RETURNS Money

AS BEGIN

RETURN

CASE -- kan ook met IF ... ELSE ... THEN maar dit is mooier

WHEN @salary <= 40000 THEN @salary \* 0.7 / 12

WHEN @salary <= 55000 THEN @salary \* 0.65 / 12 ELSE @salary \* 0.6 / 12

END

END

## Cursors

Werken met cursor:

* **DECLARE** naam\_cursor CURSOR **FOR SELECT** …
  + Je maakt enkele variabelen aan en selecteert wat data via een select en stopt ze in die variabelen en zo komen ze in de cursor
* Afbeelding met schermopname, diagram, ontwerp

  Automatisch gegenereerde beschrijving**OPEN** naam\_cursor
* **FETCH** NEXT FROM naam\_cursor INTO @variabele
* While @@FETCH\_STATUS = 0
  + Print de variabelen en doe opnieuw een FETCH NEXT
* **CLOSE** naam\_cursor
* **DEALOCATE** naam\_cursor
* **Als je deze informatie al weet, kan je ook een dubbele cursor maken**

Voorbeeld van een cursor:

**DECLARE** @employeeNaam NVARCHAR(30), @employeeSalaris DECIMAL(10, 2);

**DECLARE** employeeCursor CURSOR FOR

**SELECT** e.FirstName + ' ' + e.LastName, e.Salary

**FROM** Employees e;

**OPEN** employeeCursor;

**FETCH NEXT** FROM employeeCursor INTO @employeeNaam, @employeeSalaris;

**WHILE @@FETCH\_STATUS = 0**

BEGIN

PRINT 'Employee: ' + @employeeNaam + ', Salaris: ' + CAST(@employeeSalaris AS NVARCHAR(100));

**FETCH NEXT** FROM employeeCursor INTO @employeeNaam, @employeeSalaris;

END;

**CLOSE** employeeCursor;

**DEALLOCATE** employeeCursor;

## Triggers

Trigger:

Een trigger: een databaseprogramma, bestaande uit procedurele en declaratieve instructies, opgeslagen in de catalogus en geactiveerd door het DBMS als een bepaalde bewerking op de database wordt uitgevoerd en als aan een bepaalde voorwaarde is vervuld

Schema dat je vanbuiten moet leren:

Afbeelding met tekst, schermopname, Lettertype, nummer

Automatisch gegenereerde beschrijving

Waarom triggers?

* Dankzij triggers kan je validatie stellen op bepaalde opdrachten als insert, update of delete.
* Stel dat je bv. Iets verwijderd dat niet bestaat kan je via een trigger een foutmelding geven.
* Een ander voorbeeld is dat als je iets update in één tabel dat het ook in andere tabellen overgaat.

Voordelen:

* Eenmaal gemaakt, zal het altijd worden uitgevoerd
* Geen redundante code

Nadelen:

* Moeilijk om te debuggen
* Database ontwerpen wordt nog ingewikkelder

Schema om vanbuiten te leren:

Afbeelding met tekst, schermopname, nummer, Lettertype

Automatisch gegenereerde beschrijving

2 Geheime tabellen:

* Wanneer je een trigger uitvoert gebeurt dit via 2 geheime tabellen:
  + Deleted
    - Houdt alle verwijderde en geüpdatet records bij
  + Inserted
    - Houdt alle toegevoegde records bij

Afbeelding met tekst, Lettertype, schermopname, lijn

Automatisch gegenereerde beschrijvingMaken van een trigger:

* CREATE TRIGGER trigger\_naam
* ON table\_naam
* FOR (INSERT, UPDATE or DELETE)
* Na dit kan je beginnen met foutafhandeling via if’s etc. …

Extra informatie triggers:

* Je kan een trigger maken voor meerdere operaties
* Dus één trigger voor zowel insert, update or delete

Extra toevoeging updatetrigger:

* Je kan je afvragen of een specifieke kolom wordt geüpdatete
* Dit kan met code IF UPDATE(unitprice) gevolgd door de rest van de body van de trigger

Voorbeeld van een dubbele cursor:

**DECLARE** @employeeNaam NVARCHAR(30), @employeeSalaris DECIMAL(10, 2);

**DECLARE** employeeCursor CURSOR FOR

**SELECT** e.FirstName + ' ' + e.LastName, e.Salary

**FROM** Employees e;

**OPEN** employeeCursor;

**FETCH NEXT FROM** employeeCursor INTO @employeeNaam, @employeeSalaris;

WHILE @@FETCH\_STATUS = 0

**BEGIN**

**PRINT** 'Employee: ' + @employeeNaam + ', Salaris: ' + CAST(@employeeSalaris AS NVARCHAR(100));

**FETCH** **NEXT** **FROM** employeeCursor INTO @employeeNaam, @employeeSalaris;

declare @orderId int, @orderEmployeeId int, @orderFreight money, @orderEmployeeName varchar(30)

declare orderCursor cursor for

**select** o.OrderID, o.EmployeeID, o.Freight, e.FirstName + ' ' + e.LastName as fullname

**from** Orders o join Employees e on o.EmployeeID = e.EmployeeID

open orderCursor

**fetch next from** orderCursor into @orderId , @orderEmployeeId , @orderFreight , @orderEmployeeName

while @@FETCH\_STATUS = 0

begin

**print** 'Order: ' + cast(@orderId as varchar(10)) + ', EmployeeId: ' + cast(@orderEmployeeId as varchar(10))

+ ', Freight: ' + cast(@orderFreight as varchar(50)) + ', EmployeeName: ' + @orderEmployeeName

**fetch next from orderCursor** into @orderId , @orderEmployeeId , @orderFreight , @orderEmployeeName

end

**open** orderCursor;

**close** orderCursor;

**deallocate** orderCursor;

END;

**CLOSE** employeeCursor;

**DEALLOCATE** employeeCursor;

# Hoofdstuk 5: Indexen en performantie

Is performantie nog steeds relevevant?

* Aan de ene kant: dwet van Moore
  + De wet van Moore is de observatie dat het aantal transistors in een dicht geïntegreerd circuit (IC) ongeveer elke twee jaar verdubbelt. De wet van Moore is een observatie en projectie van een historische trend. Het is geen natuurkundige wet, maar een empirische relatie die gekoppeld is aan winst uit ervaring in productie.
  + Niet langer geldig sinds 2016?
* Aan de andere kant: De wet van Wirth
  + Software wordt sneller trager dan hardware sneller wordt

Space allocation by sql-server:

* SQL Server gebruikt bestanden met willekeurige toegang
* Toewijzing van ruimte in extents en pagina's
* Pagina = blok van 8 kb aaneengesloten ruimte
* Extent = 8 logische opeenvolgende pagina's.
  + Uniforme extents: voor één db-object
  + Gemengde extents: kunnen worden gedeeld door 8 db-objecten (=tabellen, indexen)
* Nieuwe tabel of index: toewijzing in gemengde extent
* Uitbreiding > 8 pagina's: in uniforme omvang
* Https://techyaz.com/sql-server/understanding-sql-serverdata-files-pages-extents/

Creatie van indexes:

* Unieke index: alle waarden in de geïndexeerde kolom moeten uniek zijn
* Opmerking:
  + Bij het definiëren van een index kan de tabel leeg of gevuld zijn.
  + Kolommen in een unieke index mogen niet de not null beperking constraint

Verwijderen van indexes:

* Uniek alle waarden in de geïndexeerde kolom moeten uniek zijn
* Opmerking:
  + Bij het definiëren van een index kan de tabel leeg of gevuld zijn.
  + Kolommen in een unieke index moeten de beperking not null hebben

Table scan:

* Heap: een ongeordende verzameling van gegevenspagina's zonder geclusterd index = **standaardopslag van een tabel**
* Toegang via Index Alllocation Map (IAM)
* **Tabelscan: als een query alle pagina's van de tabel ophaalt**
* Vaak het beste om te vermijden!
* Andere prestatieproblemen met heap:
  + Fragmentatie: tabel is verspreid over meerdere, niet opeenvolgende pagina's
  + Afbeelding met tekst, schermopname, Lettertype, ontwerp

    Automatisch gegenereerde beschrijvingForward pointers: als een rij met variabele lengte (bv. Varchar velden) langer wordt bij een update, wordt een forward pointer naar een andere pagina toegevoegd. → tabel scannen nog langzamer

Table scan (vervolg):

* Eerste stap: ongeordende gegevens in tabellen zetten
* Zeer inefficiënt: als we een telefoonnummer zoeken, zouden we door alle telefoonnummers moeten gaan, zelfs wanneer we een overeenkomst hebben gevonden, want er zijn misschien meer overeenkomsten later
* Tabelscans zijn niet altijd slecht, bijvoorbeeld als je veel gegevens gaat ophalen uit de tabel
* In een telefoonboek: gegevens worden geordend op achternaam en als er duplicaten zijn, worden de gegevens ook op voornaam geordend
* Tweede stap: Je kunt proberen de gegevens in fysieke volgorde te zetten
* Dit lost het prestatieprobleem niet op. Het zou een soortgelijke type van tabel scannen, maar deze keer zou het weten waar te stoppen omdat de tabelgegevens geordend zijn.

Clustered index:

Afbeelding met tekst, Lettertype, schermopname

Automatisch gegenereerde beschrijvingNadat de tabelgegevens in fysieke volgorde zijn gezet, bouwt SQL Server een set indexpagina's op waarmee query’s **direct naar de gegevens kunnen navigeren waarin ze geïnteresseerd zijn**. Geïnteresseerd zijn. De hele structuur, inclusief de tabelgegevens, wordt een geclusterde index genoemd. Wanneer een query door de boom van de geclusterde index naar de basistabelgegevens, wordt dit een geclusterde **index seek** genoemd

Non clustered index:

* Aangezien een geclusterde index de gegevens van de basistabel, zelf bevat, kun je maar één geclusterde index maken.
* Je moet een aparte fysieke structuur maken die dezelfde indexstructuur heeft als de geclusterde index. Deze keer echter, in plaats van de basisgegevens op het onderste (of leaf) niveau van de boom, is er een set van pointers (of verwijzingen) terug naar de basisgegevens => we kunnen elke volgorde van indexsleutels gebruiken
* In dit voorbeeld zijn de basisgegevens een hoop en de verwijzingen ernaar zijn RID's (row identifiers). Dit zijn de fysieke locaties van de rijen in de tabel.
* Aangezien een niet-geclusterde index gescheiden is van de basisgegevens, kunnen de basisgegevens in plaats daarvan bestaan als een geclusterde index. Als dat zo is, zijn de verwijzingen ernaar geen RID's, maar de sleutelwaarden van de geclusterde index.
* Net als een geclusterde index scan en zoek, kunnen niet-geclusterde indexen dezelfde bewerkingen uitvoeren. Maar de direct beschikbare gegevens kunnen beperkt zijn omdat niet-geclusterde indexen meestal slechts een subset van kolommen uit de tabel bevatten. Als waarden worden opgevraagd van kolommen die niet in de index staan, kan de query terug navigeren naar de basisgegevens met behulp van de referenties.

Afbeelding met tekst, schermopname, Rechthoek, diagram

Automatisch gegenereerde beschrijving

Afbeelding met tekst, diagram, schermopname, Plan

Automatisch gegenereerde beschrijvingFiltered index:

* Filtered indexes only contain rows that meet a user defined predicate, and to create these, you have to adda WHERE clause to the index definition.
* A clustered index can't be filtered because it has to contain all the data in the table.

Include:

* INCLUDE-kolommen voegen kopieën van niet-sleutel kolomwaarden toe aan het bladniveau van de indexboom.
* Dit betekent dat query’s die de niet-geclusterde index gebruiken, niet terug hoeven te navigeren naar de basisgegevens om die niet-sleutel kolomwaarden te krijgen.
* Een geclusterde index heeft geen INCLUDE kolommen nodig, omdat alle kolommen beschikbaar zijn in de leafs.

Clustered vs non-clustered:

* De geclusterde index is een manier om de basisgegevens als geheel weer te geven.
* Een niet-geclusterde index is een fysiek afzonderlijke structuur die verwijst naar de basisgegevens en die een andere sorteervolgorde kan hebben.

Table scan:

Met dit icoontje  kan je een table scan uitvoeren hier kan je de cost zien van een sql-operatie.

Indexen maken het verschil:

* **Wat?**
  + Geordende structuur opgelegd aan records uit een tabel
  + Snelle toegang door boomstructuur (B-tree = gebalanceerde boom)
* **Waarom?**
  + Kan het ophalen van gegevens versnellen
  + Kan eenheid van rijen afdwingen
* **Waarom niet?** 
  + Indexen verbruiken opslagruimte (overhead)
  + Indexen kunnen vertragen

SQL-optimizer:

* SQL Optimizer: module in elke DBMS
* Analyseert en herformuleert elk SQL-commando dat naar de DB wordt gestuurd
* Bepaalt de optimale strategie voor bijvoorbeeld het gebruik van indexen op basis van statistieken over tabelgrootte, tabelgebruik en gegevensdistributie
* In SQL wordt zoeken gebruikt voor velden in where-, group by-, having- en order by-clausules en voor velden die worden samengevoegd.

Afbeelding met diagram, lijn, Plan, Technische tekening

Automatisch gegenereerde beschrijvingClustered index:

* The physical order of the rows in a table corresponds to the order in the clustered index.
* As a consequence, each table can have only one clustered index.
* The clustered index imposes unique values and the primary key constraint
* Advantages as opposed to table scan:
  + Double linked list ensures order when reading sequential records + no forward pointers necessary

Afbeelding met diagram, tekst, lijn, schets

Automatisch gegenereerde beschrijvingNon Clustered index:

* Standaard index
* Langzamer dan geclusterde index
* > 1 per tabel toegestaan
* Voorwaartse en achterwaartse verwijzingen tussen bladknooppunten
* Elke bladzijde bevat sleutelwaarde en rij locator
  + Naar positie in geclusterde index als die bestaat
  + Anders naar hoop
* Als een query meer velden nodig heeft dan in de index aanwezig zijn, moeten deze velden worden opgehaald uit gegevenspagina's.
* Bij lezen via niet-geclusterde index
  + Ofwel: RID lookup = bookmark lookups naar de heap met behulp van RID's (= row identifiers)
  + Ofwel: key lookup = bookmark lookups naar een geclusterde index, indien aanwezig

Covering indexes:

* Als een niet-geclusterde index een query niet volledig afdekt, voert SQL Server voor elke rij een **lookup** uit om de gegevens op te halen.
* Covering index = niet-geclusterde index die alle kolommen bevat die nodig zijn voor een bepaalde query
* Met SQL Server kunt u extra kolommen aan de index toevoegen (hoewel deze kolommen niet geïndexeerd zijn!)
* Voorbeeld, er staat een index op alleen achternaam:
  + Index **seek** via niet-geclusterde index voor zoeken achternaam = 'Duffy'
  + **Key lookup** in geclusterde index (= data) voor ophalen titel (niet in index)

Één index met meerdere kolommen of meerdere indexen met één kolom?

* Bij het bevragen (bijv. In de WHERE-clausule) van alleen het 2e en 3e, ... Veld van de index, wordt de index niet gebruikt. Dit volgt direct uit de B-tree tabelstructuur van de samengestelde index.
* Maak je indexen volgens de meest gebruikte queries.

Index seek vs index scan:

* Index Seek: de boomstructuur van de index wordt gebruikt, wat resulteert in het zeer snel ophalen van gegevens.
* Index Scan: index wordt gebruikt, maar deze wordt vanaf het begin gescand totdat de gezochte records zijn gevonden.

Wanneer een index te gebruiken?

* Welke kolommen moeten worden geïndexeerd?
  + Primaire en unieke kolommen worden automatisch geïndexeerd
  + Foreign keys vaak gebruikt in joins
  + Kolommen die vaak worden gebruikt in zoekvoorwaarden (WHERE, HAVING, GROUP BY ) of in joins
  + Kolommen die vaak worden gebruikt in de ORDER BY clausule
* Welke kolommen moeten niet geïndexeerd worden?
  + Kolommen die zelden worden gebruikt in queries
  + Kolommen met een klein aantal mogelijke waarden (bijv. Geslacht)
  + Kolommen in kleine tabellen
  + Kolommen van het type bit, tekst of afbeelding

Samenvatting van indexen:

Afbeelding met tekst, schermopname, Lettertype, nummer

Automatisch gegenereerde beschrijving

Maken van indexen in sql:

* Als je dit uitvoert zonder include en je voert select p.ProductName uit is ProductName automatische gesorteerd
* Voer je p.UnitPrice aan de select-statement toe is het niet meer gesorteerd tenzij je het include in de nonclustered index

**CREATE NONCLUSTERED INDEX** IX\_Products\_ProductName

ON Products (ProductName) include (UnitPrice);

**DROP INDEX** IX\_Products\_ProductName ON Products;

# Hoofdstuk 6: Basics of Transaction Management

## Introduction

Transactions, Recovery and Concurrency Control:

* Majority of databases are multi user databases
* Concurrent access to the same data may induce different types of anomalies
* Errors may occur in the DBMS or its environment
* DBMS must support ACID (Atomicity, Consistency, Isolation, Durability) properties
* Transaction: set of database operations induced by a single user or application, that should be considered as one undividable unit of work
  + E.g., transfer between two bank accounts of the same customer
* Transaction always 'succeeds' or 'fails' in its entirety
* Transaction renders database from one consistent state into another consistent state
* Examples of problems: hard disk failure, application/DBMS crash, division by 0, …
* **Recovery**: activity of ensuring that, whichever of the problems occurred, the database is returned to a consistent state without any data loss afterwards
* **Concurrency control**: coordination of transactions that execute simultaneously on the same data so that they do not cause inconsistencies in the data because of mutual interference

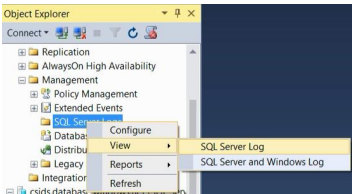
ACID Properties of Transactions:

* ACID stands for Atomicity, Consistency, Isolation and Durability
* Atomicity guarantees that multiple database operations that alter the database state can be treated as one indivisible unit of work
  + Recovery manager can induce rollbacks where necessary, by means of UNDO operations
* Consistency refers to the fact that a transaction, if executed in isolation, renders the database from one consistent state into another consistent state
  + Developer is primary responsible
  + Also an overarching responsibility of the DBMS’s transaction management system
* Isolation denotes that, in situations where multiple transactions are executed concurrently, the outcome should be the same as if every transaction were executed in isolation
  + Responsibility of the concurrency control mechanisms of the DBMS, as coordinated by the scheduler
* Durability refers to the fact that the effects of a committed transaction should always be persisted into the database
  + Responsibility of recovery manager (e.g. By REDO operations or data redundancy)

## Transactions and Transaction Management

Delineating Transactions and the Transaction Lifecycle:

* Transactions boundaries can be specified implicitly or explicitly
  + Explicitly
    - Developer determines when transaction starts and stops or how steps are rolled back to handle faulty situations
    - Begin transaction and rollback transaction / commit transaction
  + Implicitly: first executable SQL statement
* Once the first operation is executed, the transaction is active
* If transaction completed successfully, it can be **committed**. If not, it needs to be **rolled back**.

Logfile registers:

* A unique log sequence number
* A unique transaction identifier
* A marking to denote the start of a transaction, along with the transaction’s start time and indication whether the transaction is read only or read/write
* Identifiers of the database records involved in the transaction, as well as the operation(s) they were subjected to
* **Before images** of all records that participated in the transaction
* **After images** of all records that were changed by the transaction
* The current state of the transaction (active, committed or aborted)
* Afbeelding met tekst, schermopname

  Automatisch gegenereerde beschrijvingLogfile may also contain checkpoints
  + Moments when buffered updates by active transactions, as present in the database buffer, are written to disk at once
* Write ahead log strategy
  + All updates are registered on the logfile before written to disk
  + Afbeelding met tekst, schermopname, nummer, Lettertype

    Automatisch gegenereerde beschrijvingBefore images are always recorded on the logfile prior to the actual values being overwritten in the physical database files

## Recovery

Types of failure:

* **Transaction failure** results from an error in the logic that drives the transaction’s operations and/or in the application logic
* **System failure** occurs if the operating system or the database system crashes
* **Media failure** occurs if the secondary storage is damaged or inaccessible T

Afbeelding met tekst, diagram, lijn, schermopname

Automatisch gegenereerde beschrijvingSystem recovery:

* In case of system failure, 2 types of transactions
  + Already reached the committed state before failure
  + Still in an active state
* Logfile is essential to take account of which updates were made by which transactions (and when) and to keep track of before images and after images needed for the UNDO and REDO
* Database buffer flushing strategy has impact on UNDO and REDO
* Note 1: checkpoint denotes moment the buffer manager last ‘flushed’ the database buffer to disk!
* Note 2: similar reasoning can be applied in case of transaction failure (e.g. T3 , T5 )

Media recovery:

* Media recovery is invariably based on some type of data redundancy
  + Stored on offline (e.g., a tape vault) or online media (e.g., online backup hard disk drive)
* Tradeoff between cost to maintain the redundant data and time needed to restore the system
* Two types: disk mirroring and archiving

Afbeelding met tekst, cilinder

Automatisch gegenereerde beschrijvingDisk mirroring:

* A (near) real time approach that writes the same data simultaneously to 2 or more physical disks
* Limited failover time but often costlier than archiving
* (limited) negative impact on write performance but opportunities for parallel read access

Afbeelding met elektronica, compactdisk, cirkel, Apparaat voor gegevensopslag

Automatisch gegenereerde beschrijvingArchiving:

* Database files are periodically copied to other storage media (e.g. Tape, hard disk)
* Trade-off between cost of more frequent backups and cost of lost data
* Full versus incremental backup

Mixed approach: rollfoward recovery

* Archive database files and mirror logfile such that the backup data can be complemented with (a redo of) the more recent transactions as recorded in the logfile
* Note: nosql databases allow for temporary inconsistency, in return for increased performance **(eventual consistency)**

## Concurrency Control

Typical concurrency problems:

* Scheduler is responsible for planning the execution of transactions and their operations
* Simple serial execution would be very inefficient
* Scheduler will ensure that operations of the transactions can be executed in an interleaved way
* Interference problems could occur
* Lost update problem
* Uncommitted dependency problem
* Inconsistent analysis problem

Afbeelding met tekst, schermopname, Lettertype, lijn

Automatisch gegenereerde beschrijving

Typical concurrency problems (next):

* Het probleem van **lost update** treedt op als een anders succesvolle update van een data-item door een transactie wordt overschreven door een andere transactie die niet 'op de hoogte' was van de eerste update.
* Afbeelding met tekst, schermopname, Lettertype, nummer

  Automatisch gegenereerde beschrijving Als een transactie een of meer gegevensitems leest die worden bijgewerkt door een andere, nog niet vastgelegde, transactie, kunnen we tegen het probleem van de niet-vastgelegde afhankelijkheid (ook wel **Dirty read** genoemd) aanlopen
* Afbeelding met tekst, schermopname, nummer, Lettertype

  Automatisch gegenereerde beschrijvingThe **inconsistent analysis** problem denotes a situation where a transaction reads partial results of another transaction that simultaneously interacts with (and updates) the same data items.
* **Non-repeatable read** (unrepeatable read) treedt op wanneer een transactie T1 dezelfde rij meerdere keren leest, maar verschillende opvolgende waarden verkrijgt, omdat een andere transactie T2 deze rij in de tussentijd heeft bijgewerkt.
* **Phantom Read** kan voorkomen wanneer een transactie T2 invoeg- of verwijderbewerkingen uitvoert op een set rijen die door een transactie T1 worden gelezen.

Afbeelding met tekst, schermopname, Lettertype, ontwerp

Automatisch gegenereerde beschrijving

Schedules and Serial Schedules:

* A schedule S is a set of n transactions, and a sequential ordering over the statements of these transactions, for which the following property holds: “For each transaction T that participates in a schedule S and for all statements si and sj that belong to the same transaction T: if statement si precedes statement sj in T, then si is scheduled to be executed before sj in S.”
* Schedule preserves the ordering of the individual statements within each transaction but allows an arbitrary ordering of statements between transactions
* Schedule S is serial if all statements si of the same transaction T are scheduled consecutively, without any interleave with statements from a different transaction
* Serial schedules prevent parallel transaction execution
* We need a non-serial, correct schedule!

Optimistic and Pessimistic Schedulers:

* It is likely that transactions will interfere and cause conflicts
* Execution of transaction’s operations delayed until scheduler can schedule them in such a way that chance of conflicts is minimized
* Will reduce the throughput to some extent
* E.g., a serial scheduler.

## Locking and Locking Protocols

Purposes of Locking:

* Purpose of locking is to ensure that, in situations where different concurrent transactions attempt to access the same database object, access is only granted in such a way that no conflicts can occur
* A lock is a variable that is associated with a database object, where the variable’s value constrains the types of operations that are allowed to be executed on the object at that time
* Lock manager is responsible for granting locks (locking) and releasing locks (unlocking) by applying a locking protocol
* An exclusive lock (x-lock or write lock) means that a single transaction acquires the sole privilege to interact with that specific database object at that time
  + No other transactions are allowed to read or write it
* A shared lock (s-lock or read lock) guarantees that no other transactions will update that same object for as long as the lock is held
  + Other transactions may hold a shared lock on that same object as well, however they are only allowed to read it
* If a transaction wants to update an object, an exclusive lock is required
  + Afbeelding met tekst, schermopname, Lettertype, nummer

    Automatisch gegenereerde beschrijvingOnly acquired if no other transactions hold any lock on the object
* Compatibility matrix
* Lock manager implements locking protocol
  + Set of rules to determine what locks can be granted in what situation (based on e.g. Compatibility matrix )
* Lock manager also uses a lock table
  + Which locks are currently held by which transaction, which transactions are waiting to acquire certain locks, etc.
* Lock manager needs to ensure ‘fairness’ of transaction scheduling to, e.g., avoid starvation

Isolation levels:

* Level of transaction isolation offered by 2PL may be too stringent
* Limited amount of interference may be acceptable for better throughput
* Long-term lock is granted and released according to a protocol, and is held for a longer time, until the transaction is committed
* A short-term lock is only held during the time interval needed to complete the associated operation
  + Use of short-term locks violates rule 3 of the 2PL protocol
  + Can be used to improve throughput!
* Reader: statement that reads data using a shared lock (SELECT)
* Writer: statement that writes data, using an exclusive lock (INSERT, UPDATE, DELETE)
* Writers can't be influenced in SQL Server with respect to the locks they claim and the duration of these locks. They always claim an exclusive lock.
  + Readers can be influenced explicitly
  + Using isolation levels
* This might have an implicit influence on the behaviour of writers
* Isolation level = setting at session or query level

Afbeelding met tekst, schermopname, software, nummer

Automatisch gegenereerde beschrijvingIsolation Levels in SQL Server:

* Based on a pessimistic concurrency control (locking)
* Afbeelding met Lettertype, Elektrisch blauw, tekst, lijn

  Automatisch gegenereerde beschrijving1. READ UNCOMMITTED
* 2. READ COMMITTED (default)
* 3. REPEATABLE READ
* 4. SERIALIZABLE

Isolations levels (next):

* **Read uncommitted** is het laagste isolatieniveau. Er wordt geen rekening gehouden met lange-termijn locks; er wordt aangenomen dat concurrency conflicten niet voorkomen of simpelweg dat hun impact op de transacties met dit isolatieniveau niet problematisch zijn. Dit isolatieniveau is meestal alleen toegestaan voor alleen-lezen transacties, die sowieso geen updates uitvoeren.
* **Read committed** gebruikt lange termijn schrijfsloten, maar korte termijn lees-sloten. Op deze manier kan een transactie gegarandeerd geen gegevens lezen die nog worden bijgewerkt door een nog niet vastgelegde transactie. Dit lost zowel het verloren update als het uncommitted afhankelijkheidsprobleem op. Echter, het inconsistente analyse probleem kan nog steeds voorkomen met dit isolatieniveau, evenals niet-herhaalbare lezingen en fantoomlezingen.
* Afbeelding met tekst, schermopname, nummer, Lettertype

  Automatisch gegenereerde beschrijving**Repeatable read** gebruikt zowel lange termijn lees- als schrijfvergrendelingen. Een transactie kan dus herhaaldelijk dezelfde rij lezen, zonder storing door invoeg-, update- of verwijderbewerkingen door andere transacties. Toch blijft het probleem van spooklezingen onopgelost met dit isolatieniveau.
* **Serializable** is het sterkste isolatieniveau en komt ruwweg overeen met een implementatie van 2PL. Phantom reads worden nu ook vermeden. Merk op dat in de praktijk de definitie van serialiseerbaarheid in de context van isolatieniveaus slechts neerkomt op de afwezigheid van concurrency problemen, zoals niet-herhaalbare reads en phantom reads.

**Read uncommitted:**

* Laagste isolatieniveau
* Reader vraagt niet om gedeeld slot
* Lezer nooit in conflict met schrijver (die exclusief slot heeft)
* Lezer leest ongecommitteerde data (= dirty read)

**Read committed:**

* Standaard isolatieniveau
* Laagste niveau dat dirty read voorkomt
* Reader leest alleen vastgelegde data
* Lezer eist gedeeld slot op
* Als een schrijver op dat moment een exclusief slot heeft, moet de lezer wachten op een gedeeld slot
* Lezer houdt gedeeld slot totdat data is verkregen (einde SELECT), niet tot einde transactie (= kortdurend slot)
  + Opnieuw lezen van data in dezelfde transactie kan een ander resultaat geven
  + = niet-herhaalbaar lezen of inconsistente analyse
  + Aanvaardbaar voor veel, maar niet alle toepassingen

Afbeelding met tekst, schermopname, Lettertype, lijn

Automatisch gegenereerde beschrijving**Repeatable read:**

* Lezer claimt gedeeld slot en houdt dit vast tot einde transactie (= lange termijn slot)
* Andere transactie kan geen exclusief slot krijgen tot het einde van de transactie van de lezer
* Herhaalbaar lezen = consistente analyse
* Vermijdt ook verloren updates (mogelijk in 1 & 2) door aan het begin van de transactie een gedeeld slot op te eisen (met SELECT omdat alleen lezers beïnvloed kunnen worden, niet schrijvers)

**Serializable:**

* Herhaalbaar lezen vergrendelt alleen rijen gevonden met eerste SELECT
* Dezelfde SELECT in dezelfde transactie kan nieuwe rijen opleveren (toegevoegd door andere transacties) = fantomen
* Seriëerbaar vermijdt fantomen
* Blokkeert alle sleutels (huidige en toekomstige) die overeenkomen met de whereclause

Dealing with deadlocks:

* A deadlock occurs if 2 or more transactions are waiting for one another’s’ locks to be released
* Example
* Deadlock prevention can be achieved by static 2PL
  + Transaction must acquire all its locks upon the start
* Detection and resolution
  + Wait for graph consisting of nodes representing active transactions and directed edges Ti → Tj for each transaction Ti that is waiting to acquire a lock currently held by transaction Tj
  + Deadlock exists if the wait for graph contains a cycle
  + Victim selection

Lock granularity:

* Database object for locking can be a tuple, a column, a table, a tablespace, a disk block, etc.
* Trade-off between locking overhead and transaction throughput
* Many dbmss provide the option to have the optimal granularity level determined by the database system

Back-up mechanism:

* NEVER use OS backup for a database because
* Only single or very few database files → incremental backup impossible
* Data and logfiles are always open
* Due to running transactions data is inconsistent after restore
* Use backup tools provided by database vendor: e.g. Microsoft BACKUP command
* On a regular basis the data and logfiles are automatically copied to a safe location
  + Without stopping the system
  + Copies are kept on offline storage
* 2 approaches
  + Complete back-up or Incremental back-up
  + Possible backup strategy: full back up on Sunday night, incremental backup on other nights;
  + Restore: last full backup + subsequent incremental backups, can be very time-consuming => All work of current day is lost!

# Hoofdstuk 7: Datawarehousing

## Design

Afbeelding met tekst, diagram, Plan, Lettertype

Automatisch gegenereerde beschrijvingDWH components:

* Het is zodanig toegankelijk dat zelf de poetsvrouw een rapport zou kunnen trekken.
* Datawarehouse metadata mag je vergeten

Afbeelding met tekst, schermopname, Lettertype, diagram

Automatisch gegenereerde beschrijvingDWH & Data marts:

Bepaalde datamarkts die klaarstaan voor bepaalde acties zoals ‘storesalesdat’ of ‘history data’

Why a data mart?

* Gebruikers toegang geven tot de gegevens die ze het vaakst analyseren
* Gegevens aanbieden op een manier die overeenkomt met het collectieve beeld van een groep gebruikers op een afdeling of een groep gebruikers in hetzelfde bedrijfsproces
* Om de responstijd te verbeteren door lagere datavolumes aan te bieden
* Gegevens aanbieden in een formaat dat past bij de tools die eindgebruikers gebruiken (OLAP (online , datamining tools)
* Reductie van complexiteit in het ETL (Extraction, Transaction, Load) proces
* Vermindering van kosten in tegenstelling tot het opzetten van een compleet DWH
* Je kan bepaalde users bepaalde rechten geven en ook interne databanken aanmaken dat enkel bepaalde leden/het hele bedrijf kan gebruiken en niet meerdere bedrijven.

Design of a DWH:

* + 2 ontwikkelingsmethoden
  + Inmon
  + Kimball
  + Inmon en Kimball zijn de 'godfathers' van het DWH, maar ze zitten niet op dezelfde lijn als het gaat om hoe een datawarehouse eruit moet zien.
  + In deze cursus volgen we de 'Kimball' - benadering.

Inmon:

* An Inmon DWH is a normalised database
* Creation of a data model based on all **data** of the organisation;
* **An Inmon DWH is modeled based on relationships between data that are stable over time rather than based on variable processes (cfr. Kimball: see further).** 
  + E.g. An address consists of a street name, a house number possibly with an addition, a postcode and a place name. No matter how you work or what business you are in, the above is always true. An address is an address.

Kimball:

* + Normalizing leads to many different tables.
  + This makes query writing complex, and it often causes poor query performance.
  + Dimensional modeling is a way of database design that attempts to eliminate these shortcomings.
  + The starting point is that the database is given a primarily read-only (OLTP) workload. For a database with an OLTP workload, dimensional modeling is not appropriate.
  + **Dimensional modeling** leads to a database with a so-called **star model**.
  + A database consisting of only one star model contains information about one process within an organization. A database that sticks to one star is called a datamart.
  + **In Kimball's theory, the data warehouse is the collection of all star models, which in turn describe all business processes and thus describe the entire organization.**

Afbeelding met tekst, schermopname, lijn, Lettertype

Automatisch gegenereerde beschrijvingStar schema:

* A Star schema is a logical structure that has **a fact table** (containing factual data) in the center, **surrounded by** **denormalized** **dimension tables** (containing reference data).
* Natural keys from the operational system are available but not as a key in the star schema
* Surrogate integer keys are used instead because they are simpler and faster
* This way independence between OLTP and DWH is ensured
* Efficiëntie is hier heel erg belangrijk, daarom automatische nummering

Fact table:

* The fact table is the table in which quantities that make a process measurable are stored.
* The fact table has multiple foreign keys referring to each of the dimension tables, implementing a 1:N relationship type.
* The primary key of the fact table consists of the composition of all these foreign keys.
* The fact table typically contains a tuple per transaction or event (i.e., a fact) and also contains measurement data (e.g., Sales in Units and Sales in Euros).
* Important to treat fact data as read-only reference data that will not change.

Dimensions:

* Dimension tables are tables that contain the context that gives meaning to the facts.
* A dimension table stores further information about each of the facts in the fact table (e.g., Time, Store, Customer, …).
* The dimension tables contain the criteria for aggregating the measurement data and will thus be used as constraints to answer queries such as: What are the maximum sales during a particular quarter across all products, stores, and customers? What is the minimum number of units sold in store XYZ during Quarter 2 across all products? …
* To speed up report generation and avoid time-consuming joins, the dimension tables often contain denormalized data.

Measures:

* **Additive measures** can be summarized along all dimensions, using addition operators such as sum, average, etc. These are the most common type of measures encountered. E.g. Sales in Units and Sales in Euros can be meaningfully added across the Time, Store, Product, and Customer dimensions (e.g., sales per month, average sales per customer).
* **Semi-additive measures** can only be summarized using addition along some of the dimensions. E.g., the measure inventory\_quantity cannot be added across two different time periods since the quantities may be overlapping.
* **Non-additive measures** cannot be added along any of the dimensions. Examples are product price or cost.

Afbeelding met tekst, schermopname, nummer, lijn

Automatisch gegenereerde beschrijving

Example (star schema):

It can be seen that Bart (with customerkey = 20006008 and customernr = 20) bought 12 bottles of Jacques Selosse, Brut Initial, 2012 (with productkey = 30 and prodnr = 0199) in Vinos del Mundo (with storekey = 150 and storenr = 69).

Snowflake schema:

* A snowflake schema is a variant of the star schema that has a fact table in the centre, surrounded by normalised dimension tables => more tables and primary–foreign key relationships, which may have a negative impact on report generation due to the many joins that need to be evaluated.
* This approach might be considered if the dimension tables grow too large and a more efficient usage of storage capacity is required.
* It may also be beneficial if it turns out that most queries don’t make use of the outer-level dimension tables (e.g., Category dimension, Segment dimension) and only need access to the dimension tables directly connected to the fact table (e.g., Product dimension, Customer dimension).
* Since the latter dimension tables are now smaller compared to the corresponding (unnormalized) star schema, they can be more easily stored in internal memory.
* Een aantal dingen die je gaat herhalen stoppen in een extra tabel, 2 tot 3 niveaus dieper
* Het optimaliseert dus je opslagruimte maar kant performantie iets verminderen.

Afbeelding met tekst, schermopname, lijn, diagram

Automatisch gegenereerde beschrijvingFact constellation:

* A fact constellation schema has more than one fact table
* The two fact tables Sales Fact Table and Shipping Fact Table share the tables Time dimension, Product dimension, and Store dimension.

Remark:

* The three schemas (star schema, snowflake schema, fact constellation) are simply reference schemas.
* An organization may choose to adopt a mixture of these approaches by, for example, normalizing some dimensions and keeping the others denormalized (e.g., a starflake schema is a combination of a denormalized star schema with a normalized snowflake schema).

Surrogate keys:

* Dimensie tabellen gebruiken surrogaat sleutels, bijv. Storekey, productkey, shipperkey, ...
* Dit zijn betekenisloze gehele getallen die worden gebruikt om het feit te verbinden met de dimensietabellen.
* Veranderingen bij tabel-x moeten beperkte aanpassingen hebben bij andere tabellen

Granularity of the fact table:

* Level of detail of one row of the fact table (hoeveel keer per dag je wil raporten trekken)
  + Higher (lower) granularity implies more (fewer) rows and more (fewer) questions that can be answered
  + Trade-off between level of detailed analysis and storage requirements
  + It is always possible to obtain lower granularity from data stored with higher granularity (e.g. Going from days to months) by using aggregation.
  + Examples: 1 tuple of the fact table corresponds to 1 line on a purchase order
  + One tuple of the fact table corresponds to one purchase order
  + One tuple of the fact able corresponds to all purchase orders made by a customer

Factless fact tables:

* A factless fact table is a fact table that contains only foreign keys and no measurement data.
* Afbeelding met tekst, schermopname, Lettertype, lijn

  Automatisch gegenereerde beschrijvingThe fact table records the attendance of a student for a course taught by a professor at a specific moment in time. This data warehouse design allows you to answer questions such as:
  + Which professor teaches the highest number of courses?
  + What is the average number of students that attend a course?
  + Which course has the maximum number0 of students?

Optimizing the dimension tables:

* A dimension table usually has a smaller number of rows compared to the fact table.
  + The number of columns can get quite large, with many of them containing descriptive text.
  + Dimension tables should be heavily indexed to improve query execution time
  + On average the number of dimension tables is between 5 and 10
  + E.g. Datedimension timekey, Date, dayofweek, dayofmonth, dayofyear, Month, monthname, Year, lastdayinweekflag, lastdayinmonthflag, fiscalweek, holidayflag, …

Afbeelding met tekst, schermopname, nummer, Lettertype

Automatisch gegenereerde beschrijvingJunk dimensions:

* How to deal with low cardinality attribute types such as flags or indicators?
* Example: On-line Purchase (Yes or No), Payment (cash or credit-card), Discount (Yes or No)
* Either add these attribute types directly to the fact table or model them as three separate dimensions
* Alternative: **Junk dimension** is a dimension that simply enumerates all feasible combinations of values of the low cardinality attribute types
  + Advantage? Contributes to the maintainability and query performance of the data warehouse
* Maakt een tabel met vaste waarde je dan steeds een bepaald veld kan oproepen met een vaste (betaal)combinatie

Outrigger tables:

* An outrigger table can be defined to store a set of attribute types of a dimension table which are highly correlated, low in cardinality, and updated simultaneously.
* Drawback: an extra join is needed.
  + Example: a Customer dimension table including demographic data (such as average income, average household size, unemployment rate, …)
  + If this information kept in Customer dimension table => a lot of ducplication
  + => put this information in an outrigger table and link it through a foreign key with the Customer table

Slowly changing dimensions:

* De term Slowly Changing Dimension verwijst naar het feit dat attributen van dimensies in de loop van de tijd kunnen veranderen en biedt in zijn oorspronkelijke theorie standaardoplossingen voor hoe hiermee om te gaan in het datawarehouse.
* Kimball beschrijft 3 manieren om om te gaan met veranderingen in dimensie attributen
  + SCD type 1
  + SCD type 2
  + SCD type 3
* Inmiddels zijn er al 8 SCD-typen + je kunt de typen combineren
  + Voorbeeld: klantsegment variërend van AAA, AA, A, BBB, ... tot C, bepaald op jaarbasis
  + We willen een klant opwaarderen van AA naar AAA.

**SCD Type 1 (geen historiek):**

* De oude waarde van een attribuut wordt **overschreven** door de nieuwe, huidige waarde.
* De waarde van het attribuut wordt niet historisch bijgehouden. .
* Bv. Foute geboortedatum, de foute datum moet niet meer worden bijgehouden.

**SCD Type 2 (nieuwe rij maken):**

* Een volledig nieuw record wordt aangemaakt telkens een attribuut gewijzigd wordt. Er zijn 'huidige' records en 'gesloten' records.
* Dupliceren van het record en toevoegen **van Start\_Date, End\_Date, en Current\_Flag**. Een nieuwe surrogaat sleutelwaarde + dezelfde waarde voor bedrijfssleutel.

**SCD Type 3 (oud veld behouden, maar nieuw veld maken met correcte waarde):**

* Zowel de huidige als de vorige waarde van een attribuut worden in twee verschillende kolommen bijgehouden. Op die manier kunnen huidige en vorige waarden gemakkelijk worden vergeleken.
* Slechts gedeeltelijke historische informatie wordt opgeslagen
* De aanpak kan eenvoudig worden uitgebreid door ook een Datum attribuuttype toe te voegen dat aangeeft wanneer de meest recente wijziging heeft plaatsgevonden.

**SCD Type 4:**

* Maak twee dimensietabellen: Klant en Klant\_Historie. Beide zijn gekoppeld aan de feitentabel met behulp van hun surrogaatsleutels, maar de eerste bevat de meest recente informatie, terwijl de laatste de volledige geschiedenis van updates bevat.
* Afhankelijk van het type informatie dat nodig is (meest recent of historisch), wordt de juiste dimensietabel geselecteerd.

**Rapidly Changing Dimensions:**

* Dimensies die snel en regelmatig veranderen gedurende een bepaalde periode
* Voorbeeld: klantsegment variërend van AAA, AA, A, BBB, ... tot C, bijgewerkt op wekelijkse basis in plaats van jaarlijks
* Benaderingen 2 en 4 besproken in de vorige sectie zullen resulteren in veel rijen
* Splits klantinformatie op in stabiele (bijv. geslacht, burgerlijke staat, ...) en snel veranderende informatie die in een minidimensionale tabel wordt gezet met een nieuwe surrogaatsleutel.
  + 2 opties om beide tabellen te verbinden Extra vreemde sleutel in de feitentabel die naar deze minidimensie verwijst
  + Een vreemde sleutel opnemen in de klantdimensie die verwijst naar het huidige segment in de minidimensie Klantsegment.

**Voordelen van dimensioneel model:**

* Voorspelbare en standaard vorm van het onderliggende dimensionale model biedt belangrijke voordelen:
* Efficiëntie
* Een consistente DB-structuur geeft tools efficiënte toegang tot de gegevens
* Vermogen om te gaan met veranderende eisen
* Het model kan gemakkelijk worden aangepast aan veranderende behoeften omdat elke dimensie equivalent is aan de feitentabel
* Ideaal voor ad hoc queries
* Uitbreidbaarheid
* Nieuwe facts toevoegen
* Nieuwe dimensies toevoegen
* Attributen toevoegen aan dimensies

## Kleine samenvatting

Verschil tussen clustered en non-clustered index:

* Clustered
  + Geeft alle rijen mee
* Non-clustered
  + Geef enkel de rij mee waarop de index is gezet, tenzij je een include doe op een bepaald veld tijdens het creëren van de index

Scan vs Seek:

* Scan:
  + Je loopt alles af
* Seek:
  + Je zoekt iets specifiek via de indexstructuur die je hebt al hebt opgebouwd
    - Like ‘Lie%’

SCD (Slowly Changing Data):

* Fixed
  + Dit blijft altijd hetzelfde
* Changing
  + Dit kan veranderen en oude waarde wordt overschreven
* Historical
  + Hier wordt een historiek van bijgehouden als de waarde veranderd

Verschil tussen OLTP en OLAP en wat ETL is:

* **OLTP (Online Transaction Processing)**
  + Relationele databasestructuur.
  + Veel tabellen met meestal **weinig kolommen**.
  + Geoptimaliseerd voor **snelle transacties** en **invoeg-/updatebewerkingen** (CRUD: Create, Read, Update, Delete).
  + Ondersteunt een groot aantal gebruikers die tegelijkertijd transacties uitvoeren.
  + Data is zeer gedetailleerd en actueel.
  + Voorbeeld: Banksystemen, reserveringssysteme
* **OLAP (Online Analytical Processing)**
  + Datawarehouse-structuur.
  + **Weinig tabellen** met meestal **veel kolommen** (denormalisatie).
  + Geoptimaliseerd voor **complexe query's** en analyses (zoals trends, samenvattingen en historische gegevens).
  + Ondersteunt beslissingsondersteunende processen (business intelligence, rapportage).
  + Data is vaak historisch en niet realtime.
  + Voorbeeld: Rapportagesystemen, dashboards**ETL (Extract, Transform, Load)**
  + Dit is het proces dat gebeurt van een OLTP-DB naar de datawarehouse

ETL:

* **ETL (Extract, Transform, Load)**
  + **Extract**:
    - Het ophalen van data uit diverse bronnen, zoals OLTP-databases, CSV-bestanden, API's, enz.
  + **Transform**:
    - Het opschonen, combineren, en aanpassen van de data zodat het geschikt is voor opslag in een datawarehouse.
    - Voorbeelden: berekeningen uitvoeren, datatypes aanpassen, duplicaten verwijderen.
  + Load:
    - Het opslaan van de getransformeerde data in een datawarehouse (OLAP).

Afbeelding met tekst, schermopname, Lettertype, nummer

Automatisch gegenereerde beschrijvingSQL-task vs Dataflow task (wordt gemaakt via een SSIS-tool):

* **SSIS (SQL Server Integration Services)**
  + Wordt gebruikt voor ETL-tasks
* **SQL-task.**
  + UPDATE, INSERT, DELETE, of een opgeslagen procedure uitvoeren in een database.
  + Een SQL-task voert SQL-scripts of opdrachten uit op een database (OLTP of OLAP).
  + Kan worden gebruikt om specifieke taken uit te voeren, zoals
* **Dataflow-task**
  + een Dataflow-task wordt gebruikt om data te verplaatsen en transformeren tussen verschillende bronnen en bestemmingen