|  |
| --- |
| HAN Arnhem |
| DI Casus |
| Course offering database |

|  |
| --- |
| Erik Knaake (598368) & Jannick Joosten (598696)  [Date] |

Inhoudsopgave

[1 Introductie 2](#_Toc4615468)

[1.1 Mappen structuur 2](#_Toc4615469)

[2 Opdracht A 3](#_Toc4615470)

[3 Opdracht B 4](#_Toc4615471)

[4 Opdracht C 7](#_Toc4615472)

[4.1 Constraint 5: Geen concurrency problemen 7](#_Toc4615473)

[4.2 Constraint 11: Non Repeatable Read 9](#_Toc4615474)

[5 Opdracht D 11](#_Toc4615475)

[6 Opdracht E 12](#_Toc4615476)

[7 Opdracht F 13](#_Toc4615477)

[8 Literatuurlijst 14](#_Toc4615478)

# Introductie

Voor het vak DI …

## Mappen structuur

In de hoofdfolder is een bestand genaamd: Di Casus.ssmssln aanwezig. Wanneer dit bestand geopend wordt door middel van Microsoft SQL Server Management Studio (ssms) zal de folder genaamd: DI Casus geopend worden binnen de Solution Explorer van ssms. Zo hoeft deze folder zelf niet benaderd te worden echter is in deze folder wel het Word bestand te vinden voor opdracht C.

Verder is er een folder aangemaakt voor alle execution plans. De execution plans die hier in staan zijn als aanvulling op opdracht D.

# Opdracht A

Opdracht A luidt: “*Implement the database according to the PowerDesigner PDM you find on the last page of this document. Once again you’ll find the scripts to get you started on OnderwijsOnline. Add to the constraint script the foreign key and the cascading rule declarations as depicted in the PowerDesigner PDM*.” (HAN, z.d.).

In het PDM staan de foreign keys vermeldt, met telkens de acties die worden genomen bij een update of delete. Hierbij staat N voor SET NULL, C voor CASCADE en niets voor NO ACTION.

Het volgende stuk code is representatief voor hoe wij de code hiervoor hebben geïmplementeerd:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FK hist -> emp (ON DELETE CASCADE)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ALTER TABLE hist

ADD CONSTRAINT FK\_hist\_emp

FOREIGN KEY (empno) REFERENCES emp(empno)

ON UPDATE NO ACTION

ON DELETE CASCADE

go

# Opdracht B

Opdracht B luidt: “*Implement constraints 1 through 11. In case a declarative implementation is possible, provide it.*

*Motivate your choice for stored procedures or triggers. Build at least 3 stored procedures and 3 triggers. If your choices result in less than 3 of each, create alternative solutions to in the end deliver at minimum 3 of each. Analyze what actions on which tables may cause the constraint to be violated (describe all possible scenarios). Implement the in your opinion most logical of the possibly many scenarios that may cause the constraint to be violated.*

***FIRST*** *create the test cases you’ll need test your solution. Add tests with* ***multiple*** *rows if your solution is a trigger (this means at least tests with at minimum two allowed, two dis-allowed and a combination of rows). Only than start implementing the constraint.*

*So, every trigger should correctly handle multiple row SQL statements.  
Minimize the use of variables. Use only one SQL-statement for a condition where ever possible.*

*Use correct error handling and transaction management in both Triggers and Stored Procedures like taught in the course.”* (HAN, z.d.).

Om deze opdracht te realiseren hebben wij telkens eerst een analyse gedaan waaruit blijkt onder welke omstandigheden de constraint fout kan gaan. Hierna hebben we een keuze gemaakt of we een stored procedure, trigger of check constraint gingen gebruiken.

Vervolgens hebben we met tSQLt testcases gebouwd, zodat er eenvoudig te valideren is of de constraint goed werkt. Als laatste hebben we de constraint geïmplementeerd.

Een voorbeeld hiervan is:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

4. A salary grade overlaps with at most one lower salary grade.

The llimit of a salary grade must be higher than the llimit of the next lower salary grade.

The ulimit of the salary grade must be higher than the ulimit of the next lower salary grade.

Kan misgaan bij:

- Insert in grd waarbij de llimit lager is dan de llimit van een lagere grd

- Insert in grd waarbij de ulimit lager is dan de ulimit van een lagere grd

- Update van grd waarbij de llimit lager wordt dan de llimit van een lagere grd

- Update van grd waarbij de ulimit lager wordt dan de ulimit van een lagere grd

Gekozen voor een trigger omdat die met dezelfde code zowel voor inserts als updates kan werken

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

GO

CREATE OR ALTER TRIGGER utr\_OverlappingSalaryGrades

ON grd

AFTER UPDATE, INSERT

AS

BEGIN

BEGIN TRY

SET NOCOUNT ON

IF UPDATE(llimit) OR UPDATE(ulimit)

BEGIN

IF EXISTS (

SELECT 1

FROM inserted i

WHERE EXISTS (

SELECT 1

FROM grd g

WHERE g.grade < i.grade AND (

i.llimit < g.llimit OR

i.ulimit < g.ulimit

)

)

)

THROW 50040, 'Salary grades kunnen niet overlappen', 1

END

END TRY

BEGIN CATCH

THROW

END CATCH

END

GO

Een deel van de bijbehorende tests zien er als volgt uit:

EXEC tSQLt.NewTestClass 'testSalaryGradesCantOverlap'

GO

CREATE OR ALTER PROC testSalaryGradesCantOverlap.SetUp

AS

BEGIN

EXEC tSQLt.FakeTable 'dbo.grd'

EXEC tSQLt.ApplyTrigger 'dbo.grd', 'dbo.utr\_OverlappingSalaryGrades'

SELECT \*

INTO expected

FROM dbo.grd

END

GO

GO

CREATE OR ALTER PROC testSalaryGradesCantOverlap.testInsertWithWrongLowerLimit

AS

BEGIN

INSERT INTO grd VALUES (1, 10, 20, NULL)

INSERT INTO expected VALUES (1, 10, 20, NULL)

EXEC tSQLt.ExpectException @ExpectedErrorNumber = 50040

INSERT INTO grd VALUES (2, 9, 20, NULL)

EXEC tSQLt.AssertEqualsTable expected, grd

END

GO

GO

CREATE OR ALTER PROC testSalaryGradesCantOverlap.testInsertWithCorrectLowerLimit

AS

BEGIN

INSERT INTO grd VALUES (1, 10, 20, NULL)

INSERT INTO expected VALUES (1, 10, 20, NULL), (2, 11, 20, NULL)

EXEC tSQLt.ExpectNoException

INSERT INTO grd VALUES (2, 11, 20, NULL)

EXEC tSQLt.AssertEqualsTable expected, grd

END

GO

Elke test is opgebouwd uit een aantal delen, namelijk:

* Het aanmaken van een test klasse waaronder de test valt
* Het aanmaken van een SetUp procudure die de tabellen compleet vrij maakt van andere constraints, zodat de constraint in isolatie te testen is
* De test gevallen, deze bestaan elk uit drie delen (Microsoft, 2016), namelijk:
  + Arrange, hierin wordt er data voorbereidt die nodig is om het testgeval uit te kunnen voeren
  + Act, hierin wordt de actie die getest moet worden uitgevoerd
  + Assert, hierin wordt gecontroleerd of de actie daadwerkelijk heeft gedaan wat er verwacht word.

# Opdracht C

Opdracht C luidt: *“Choose* ***two*** *of your procedural* ***constraints****. One which has no problems with non-repeatable read or phantoms, in a multi-user environment under the default isolation level (READ COMMITTED) and one which does.*

*Explain your choices by giving scenarios with two transactions that illustrate why it can or can’t go wrong. Add a success scenario with an isolation level that solves the problematic one.*

*For every scenario describe what kind of locks are acquired (e.g. s-locks and x-locks), when, why and for how long.”*

Voor deze opdracht hebben wij gekeken naar twee van onze constraints. Met gebruik van tabellen, waarin twee connecties / transacties zich bevinden, tonen wij aan hoe het zit met de concurrency en of dit dus fout kan gaan.

## Constraint 5: Geen concurrency problemen

Hieronder wordt er gekeken naar een constraint waarbij geen concurrency problemen op kunnen treden. Deze constraint voert een enkele update uit, en het is niet mogelijk dat er tijdens deze update een tweede connectie iets aan kan passen. Ondanks dat er dus twee keer dezelfde select statement wordt uitgevoerd, zal een andere connectie geen aanpassingen kunnen aanbrengen.

|  |  |
| --- | --- |
| Connection 1 | Connection 2 |
| SET TRANSACTION ISOLATION LEVEL  READ COMMITED |  |
| BEGIN TRANSACTION |  |
| (initialy)  insert into reg values (1111, 'RGDEV', '2004-09-03', 4) |  |
| (trigger)  UPDATE offr  SET status = 'CONF'  WHERE course IN (  SELECT O.course  FROM offr O JOIN reg R ON O.course = R.course AND O.starts = R.starts  WHERE O.status <> 'CONF'  GROUP BY O.course, O.starts  HAVING COUNT(\*) >= 6  ) AND starts IN (  SELECT O.starts  FROM offr O JOIN reg R ON O.course = R.course AND O.starts = R.starts  WHERE O.status <> 'CONF'  GROUP BY O.course, O.starts  HAVING COUNT(\*) >= 6  )  X-Lock requested and granted until end of transaction |  |
| COMMIT TRANSACTION |  |

Nu blijkt dat volgens verschillende bronnen, nieuwere versies van SQL Server queries kunnen onderscheppen door het principe van: interleaving. Dit kan mogelijk tot problemen zorgen binnen de uitvoering van de select statements in deze query, maar aangezien dit buiten de course valt moet dit voor de huidige situatie geen probleem zijn. Er is meer over te lezen in een artikel van Joseph Sack (2017), of van Dmitry Piliugin (2018).

## Constraint 11: Non Repeatable Read

Hieronder wordt een situatie geschetst waarin het voor kan komen dat een employee een course gaat geven terwijl deze geen trainer is, ondanks dat er gecontroleerd wordt of dat deze employee wel een trainer is. Voor deze situatie bestaat er een employee met employee nummer 1111, en job TRAINER.

|  |  |  |
| --- | --- | --- |
| Connection 1 | Connection 2 | Employee job value |
| SET TRANSACTION ISOLATION LEVEL  READ COMMITED |  |  |
| BEGIN TRANSACTION |  |  |
| (initialy)  exec usp\_InsertOffering  'J2EE', '2050-10-10', 'CONF', 6, 1111, 'Amsterdam' |  |  |
| (stored procedure)  SELECT \* FROM emp WHERE job <> 'TRAINER'  AND empno = @trainer  S-Lock requested and granted until end of select |  | Employee 1111  Is een TRAINER |
|  | **update emp set job = 'ADMIN' where empno = 1111**  X-lock request and granted until auto commit | Employee 1111  Is een ADMIN |
| SELECT 1 FROM emp e WHERE e.empno = @trainer AND ( DATEDIFF( YEAR, hired, @starts ) < 1 )  S-Lock requested and granted until end of select |  | Employee 1111  Is een ADMIN |
| SELECT 1 FROM emp e INNER JOIN reg r ON e.empno = r.stud WHERE e.empno = @trainer AND r.course = @course  S-Lock requested and granted end of select |  | Employee 1111  Is een ADMIN |
| INSERT INTO offr VALUES ( @course, @starts, @status, @maxcap, @trainer, @loc ) |  | Employee 1111  Is een ADMIN |
| COMMIT TRANSACTION |  |  |

Nadat deze transactie is uitgevoerd, is er een course offering waarbij de employee een ADMIN is, en geen TRAINER terwijl dit wel de bedoeling is volgens de constraint. Dit kan dus gebeuren wanneer er na de select controle een update gedaan wordt op dezelfde employee.

Deze situatie kan opgelost worden door de transaction isolation level te zetten naar Serializable. Wanneer dit gedaan wordt zal connectie 2 wachten tot connectie 1 zijn transactie heeft voltooid. Dit is te zien in de volgende tabel.

|  |  |  |
| --- | --- | --- |
| Connection 1 | Connection 2 | Employee job value |
| SET TRANSACTION ISOLATION LEVEL  SERIALIZABLE |  |  |
| BEGIN TRANSACTION |  |  |
| (initialy)  exec usp\_InsertOffering  'J2EE', '2050-10-10', 'CONF', 6, 1111, 'Amsterdam' |  |  |
| (stored procedure)  SELECT \* FROM emp WHERE job <> 'TRAINER'  AND empno = @trainer  S-Lock requested and granted on empno 1111 until end of transaction |  | Employee 1111  Is een TRAINER |
|  | **update emp set job = 'ADMIN' where empno = 1111**  X-lock request and not granted due to serializable isolation level in connection 1. X-lock will be granted when connection 1 has completed it’s transaction. | Employee 1111  Is een TRAINER |
| … |  |  |

Deze situatie is niet op te lossen wanneer de isolation level op repeatable read wordt gezet, aangezien de employee (1111) door de eerste select niet gevonden wordt en dus ook niet vergrendeld zal worden door de isolation level. Dit zal dus betekenen dat connectie 2 nog steeds de mogelijkheid heeft om de job van deze employee aan te passen.

# Opdracht D

Opdracht D luidt: *Find* ***two queries*** *to possibly optimize by adding indexes (discuss your ideas concerning the two queries to optimize with your lecturer). Use queries you wrote in task B Now think of an index (per query) on one or more columns, either clustered or non-clustered, that may optimize the query performance. Describe your solution using plain text (do not just throw a bunch of code “over the wall”) and motivate your choice of columns and clustering well. Give the code for the indexes and the execution plans before and after adding the index.*

*(Note: improvement might be hard to actually measure given this small dataset)*

# Opdracht E

Opdracht E luidt: *The Course Database implements a bit of history awareness like in for instance the HIST table (changes in employee’s department and/or the salary amount are recorded by stacking the historical state in the HIST table). We want the history of all data changes to be automatically recorded in history tables belonging to the original tables using database triggers. Note: assume primary keys are immutable!*

*For every table in the Course Database a history version table needs to be available with name HIST\_<table name> so HIST\_EMP, DEPT etc.*

*These history tables versions need to have the same structure as the original tables, but with a different primary key consisting of the original primary key col-umn(s) combined with a timestamp column (type timestamp). The primary key is the only constraint of these history tables.*

*Assignment:*

*- isolate the fixed boilerplate template T-SQL trigger code and determine where you can find (using the Information Schema Views) and retrieve the parameter values you have to generate into the fixed boilerplate template T-SQL code to produce the specific trigger;*

*- write at minimum two stored procedures that do the job, one generating the trigger code per table (on the basis of an input parameter), one calling this stored procedure as many times as there are non-history tables.*

# Opdracht F

# Literatuurlijst

Microsoft (1 juli 2016), *Unit test basics*. Geraadpleegd op 27 maart 2019, van <https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2017>

HAN (z.d.), *CASE DI*. Geraadpleegd op 27 maart 2019, van <https://onderwijsonline.han.nl/elearning/lessonfile/XyrGK5RD/eyJpdiI6IllmU0phWWRqQW16V0tTWEhWUk9DbEE9PSIsInZhbHVlIjoiRnZia09mZkpVMytRSTB6NmZNZmRraU1qMWp6SWtCY296N3o0ZVVDOXU4VjErTzlHcFpsTm1tS1Q3RFRVXC9UbXFMTDhBV3JWSDVrdHF5cmZwdFRxU3B3PT0iLCJtYWMiOiI4ODg0YTc0ZWRiMGYwZTRmNmJkNzllZjllMGY5OGE2NTVjMjk3MDMzNTc5NWZmMDZiYjhkZDRmNGViYzcyNmJlIn0=>

Sack, J. (2017, 19 April). Introducing Interleaved Execution for Multi-Statement Table Valued Functions. Geraadpleegd van: <https://blogs.msdn.microsoft.com/sqlserverstorageengine/2017/04/19/introducing-interleaved-execution-for-multi-statement-table-valued-functions/>

Piliugin, D. (2018, 27 April). SQL Server 2017: Interleaved Execution for mTVF. Geraadpleegd van: <https://www.sqlshack.com/sql-server-2017-interleaved-execution-for-mtvf/>