|  |
| --- |
| ISE DI – Casus\_  COURSE DATABASE |

Nabben, Scholten

1 september 2021

INHOUDSOPGAVE

[1 Setup database 7](#_Toc81394552)

[1.1 Task 1 - Implement Foreign Keys and Cascading Rules 7](#_Toc81394553)

[2 Information needs 8](#_Toc81394554)

[2.1 Task 2.1 – List the valid trainers 8](#_Toc81394555)

[2.2 Task 2.2 – List exact same set of course registrations 8](#_Toc81394556)

[2.3 Task 2.3 – List total costs per salary grade per year 8](#_Toc81394557)

[2.4 Task 2.4 – List salary changes per employee 9](#_Toc81394558)

[3 Implement declarative constraints 10](#_Toc81394559)

[3.1 Task 3.1 – President earns more than $10.000 10](#_Toc81394560)

[3.2 Task 3.2 – Company hires adult personnel only 10](#_Toc81394561)

[4 Implement constraints & functionality using stored procedures 11](#_Toc81394562)

[4.1 Task 4.1 – Trainers cannot teach different courses simultaneously 11](#_Toc81394563)

[4.2 Task 4.2 – Promote employee as a manager 11](#_Toc81394564)

[5 Implement constraints using triggers 12](#_Toc81394565)

[5.1 Task 5.1 – Only valid trainers allowed 12](#_Toc81394566)

[5.2 Task 5.2 – Administrator for department 12](#_Toc81394567)

[5.3 Task 5.3 – Salary grade limitations 12](#_Toc81394568)

[5.4 Task 5.4 – Trainers can start only one course a time 13](#_Toc81394569)

[5.5 Task 5.5 – Home based course offerings 13](#_Toc81394570)

[6 Implement Unit tests 14](#_Toc81394571)

[6.1 Task 6.1 – Implement Unit tests for an information need 14](#_Toc81394572)

[6.2 Task 6.2 – Implement Unit tests for a declarative constraint 14](#_Toc81394573)

[6.3 Task 6.3 – Implement Unit tests for a stored procedure 14](#_Toc81394574)

[6.4 Task 6.4 – Implement Unit tests for constraint ‘Only valid trainers allowed’ 14](#_Toc81394575)

[6.5 Task 6.5 – Implement Unit tests for all other constraints 14](#_Toc81394576)

[7 Analyze Concurrency Control 15](#_Toc81394577)

[7.1 Task 7.1 – Analyze and solve concurrency problem stp\_RegisterForCourse 15](#_Toc81394578)

[7.2 Task 7.2 – Analyze and solve concurrency problem stp\_SetManager 16](#_Toc81394579)

[8 Implement indexes 17](#_Toc81394580)

[8.1 Task 8.1 – Optimize information need 1 17](#_Toc81394581)

[8.2 Task 8.2 – Optimize information need 2 17](#_Toc81394582)

[9 Implement history using code generation 18](#_Toc81394583)

[9.1 Task 9.1 – Generate history tables 18](#_Toc81394584)

[9.2 Task 9.2 – Generate history triggers 19](#_Toc81394585)

[9.3 Task 9.3 – Generate history tables and triggers for all tables 19](#_Toc81394586)

[10 Implement Security 20](#_Toc81394587)

[10.1 Task 10.1 – Implement Employee account 20](#_Toc81394588)

[10.2 Task 10.2 – Implement Reporting account 20](#_Toc81394589)

CASE DESCRIPTION

This case study concerns the implementation of a small company database system in which data about employees (including some historical data), their salaries, the departments where they work and company courses they take have to be managed.

In the spirit of the database modeling and design course DMDD the ten tables involved are documented in predicate style:  
  
Table **EMP**: The employee with employee number **EMPNO** has name **ENAME**, job **JOB,** was born on **BORN**, is hired on **HIRED,** has a monthly salary of **MSAL** dollars within **SGRADE** salary grade, is assigned to account **USERNAME**, and works for the department number **DEPTNO**.  
Table **SREP:** The sales representative with employee number **EMPNO** has an annual sales target of **TARGET** dollars and a yearly commission of **COMM** dollars.  
Table **MEMP**: The employee with employee number **EMPNO** is managed by the employee with employee number **MGR**.  
Table **TERM**: The employee with employee number **EMPNO** has resigned or was fired on date **LEFTCOMP** due to reasons **COMMENTS** (initially empty).  
Table **DEPT**: The department with department number **DEPTNO**, has name **DNAME**, is located at **LOC**, and is managed by the employee with employee number **MGR**.  
Table **GRD**: The salary grade with id **GRADE** has a lower monthly salary limit of **LLIMIT** dollars, an upper monthly   
limit of **ULIMIT** dollars, and a maximum yearly bonus of **BONUS** dollars.

Table **CRS**: The course with code **CODE** has description **DESCR**, falls in category **CAT**, and has a duration of **DUR** days.  
Table **OFFR**: The course offering for the course with code **COURSE** that starts on date **STARTS**, has status **STATUS**, has a maximum capacity of **MAXCAP** attendees, is offered at location **LOC**, and (unless **TRAINER** equals NULL) the offering has the employee with employee number **TRAINER** assigned as the trainer.   
Table **REG**: The employee whose employee number is **STUD** has registered for a course with code **COURSE** that starts on **STARTS**, and (unless **EVAL** equals -1) has rated the course with an evaluation score of **EVAL**.  
Table **HIST**: At date **UNITIL**, for the employee whose employee number is **EMPNO**, either department or the monthly salary (or both) have changed. Prior to the date **UNTIL**, the department for that employee was **DEPTNO** and the monthly salary was **MSAL** (table initially empty).

Note: whether you like it or not, this is the database you have to deal with (you have to deal with some design flaws in your code).   
  
*Relational schema*

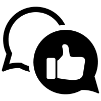
This schema is created with SAP PowerDesigner, shows the main structural- and integrity aspects of the database.



Scripts

At OnderwijsOnline (https://onderwijsonline.han.nl) in the course folder ISE DI chapter 2.2, you’ll find three sql scripts, a basic create table script (COURSE\_cretab.sql), an insert script (COURSE\_database\_state.sql) and a constraints creation script (COURSE\_constraints.sql). Run these scripts in the order as listed above and study these scripts.

Assignment tasks

Below you’ll find tasks to implement the Course Offering Database, but before you start, read the next section meticulously.   
  
The case study consists of two sets of assignments. One set of questions consists of a number of **feedback assignments** (in Dutch: formatieve feedback/beoordeling) organized per week. You hand in your answers to these feedback questions at the end of the week and receive feedback in the next week. The procedure of how to hand in/upload your work and where to look for feedback will be explained by your lecturer at startup of the course.  
  
The second set of case study assignments consists of **assignments which will actually be graded** (in Dutch: summatieve feedback/beoordeling). Ideally you hand in answers to these assignments after you have received feedback on feedback assignments concerning the same theme.

At the begin of the first week a detail planning will be given when each task should be finished and uploaded for feedback. For example (the first week):

|  |  |  |
| --- | --- | --- |
| Lesson | Topics | Case study (bold=FEEDBACK)  (underline=REPORT) |
| 1.1 | Herhaling SQL  Advanced SQL till EXISTS/NOT EXISTS (slide 24) | Task **1** + **2.1** |
| **1.2** | Advanced SQL tm SET operatoren (slide 36) | Task 2.2 |
| **1.3** | Advanced SQL tm Logical query processing (slide 55). | **Upload Task 1 + 2.1** |

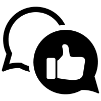
This way we try to spread the case study work load and force you to practice from the early start of the course.

In the end you deliver a report that should meet all these requirements:

* Deliver your report in Word format, **not in PDF format nor just as a .sql script file.**   
  It is easier for us to add feedback to Word files.
* Structure your document per task corresponding this document. Add all relevant T-SQL code in this document. Add also the definition of the task.
* Add all tests of one constraint in this report (see chapter 6). All other tests should be delivered as SQL script(s).
* Don’t forget to index the document and add page numbers.
* The total file size should not exceed the limitations of iSAS.

# Setup database

Run the case study database scripts provided on OnderwijsOnline



## Task 1 - Implement Foreign Keys and Cascading Rules

Check the constraint script, add the correct foreign key and cascading rule declarations as depicted in the PowerDesigner PDM.

The PDM schema uses some abbrevations:

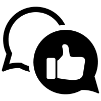
* Upd() means no cascade
* Upd(C) means cascade
* Del(N) means set null
* cpa means change parent allowed. You can ignore this for this task.

Make sure you explicitly define the cascading rules. Don’t use default implementations.

# Information needs

Some of these information needs are also usefull for implementing constraints during the rest of this case. Give an SQL implementation for each of the information needs as mentioned below. Add some testdata if needed.

Create one View per information need that produces the correct result.



## Task 2.1 – List the valid trainers

Give a list of all offered courses that are given by a valid trainer. A valid trainer is:

1. someone whose job is trainer and who is at least one year an employee in this organization,
2. or someone who has attended this course (as a participant).

Return the empno and ename of the trainer and the course code and course description of the applicable courses.

Use one or more set operators or exists operators.

The header of this view should be:

**CREATE VIEW VW\_ListTheValidTrainers**



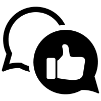
## Task 2.2 – List exact same set of course registrations

Give a list of all employees that are registered for exact the same set of courses started on the same date. Make sure to filter out the duplicates (pair A and B and B and A are duplicate pairs).

Use one or more set operators or exists operators.

The header of this view should be:

**CREATE VIEW VW\_ListExactSameSetOfCourseRegistrations**



## Task 2.3 – List total costs per salary grade per year

The cost per salary grade is calculated by the yearly salary of all employees of this grade. The yearly salary is based on the monthly salary plus the yearly bonus as defined per grade.

Give a list of all the salary grades, the total salary costs per salary grade (the bonus included) per year and also give a running total.

Use one or more windowing functions or CTE operators.

The header of this view should be:

**CREATE VIEW VW\_ListTotalCostsPerSalaryGradePerYear**



## Task 2.4 – List salary changes per employee

The table hist stores the history of modifications in salary. If the salary of an employee changes the previous salary is stored in the history table together with the actual date (column until). The salary in the employee table is always to most recent salary.

Give a list of salary modifications per employee (so sorted on employee) that gives an overview of the salary someone had in a certain periode.

The output should by in this format:

|  |  |  |  |
| --- | --- | --- | --- |
| **empno** | **msal** | **from** | **until** |
| 1033 | 4000.00 | 2000-01-01 | NULL |
| 1099 | 4000.00 | 2019-01-01 | 2019-02-01 |
| 1099 | 4100.00 | 2019-02-01 | 2019-07-01 |
| 1099 | 4300.00 | 2019-07-01 | NULL |

In this example empno 1033 has had no salary modification. The periode is from hire date until now (displayed as null). Empno 1099 has had 2 modifications. He started at 2019-01-01 (hired date), had his 1st salary increase of 100 at 2019-02-01 and his 2nd salary increase of 200 at 2019-07-01. He now earns 4300 per month.

Use one or more windowing functions or CTE operators.

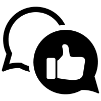
The header of this view should be:

**CREATE VIEW VW\_ListSalaryChangesPerEmployee**

*Note: You might need to add data to the history table for testing your solution.*

# Implement declarative constraints

This chapter involves rather simple constraints which can be implemented using declarative constructs.



## Task 3.1 – President earns more than $10.000

Implement and test the constraint ‘*the president of the company earns more than* *$10.000 monthly’*.

## Task 3.2 – Company hires adult personnel only

Implement and test the constraint ‘*the company hires adult personnel only’.*

# Implement constraints & functionality using stored procedures

## Social Media Positive Feedback Icon Icons - Download Free Vector Icons | Noun ProjectTask 4.1 – Trainers cannot teach different courses simultaneously

Implement and test a stored procedure enforcing the constraint ‘*trainers cannot teach different courses simultaneously’*. The constraint should be validated as soon as a new course offering is inserted.



## Task 4.2 – Promote employee as a manager

In case an employee is promoted to MANAGER several database actions need to be executed;

* The employee role needs to be updated in ‘MANAGER’
* The employees of the same department as their new manager will be managed by this employee. Therefore, the data in table MEMP need to be updated likewise.
* The salary grade of this new manager will be set to 9 and his salary amount will be set to the minimum of this grade plus an extra amount of 20.00 per employee that is managed by him/her. The salary should always be limited to the max of this grade.

For example: if employee 1015 (Antoinet) will become the new manager of department 15, she will manage 21 employees and have a salary of 8420 (8000 + 20.00 \* 21).

Some extra checks need to be done on forehand;

-         The new manager is already working for this department.

-         There is no other employee in this department with the role ‘MANAGER’

The header of this stored procedure should be:

**CREATE PROCEDURE stp\_SetManager (@empno numeric (4.0), @deptno numeric(2,0))**

*Concurrency control is not an issue yet. In Task 7.2 you will optimize this for avoiding concurrency problems.*

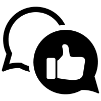
# Implement constraints using triggers

The following constraints need to be implemented using one or more triggers.

When you develop the constraint:

* Include in your report the original definition of the constraint,
* Describe which tables and columns are involved in this constraint
* Describe for which actions (I/U/D) this constraint should be validated

It is not allowed to implement a constraint with a combination of check constraints and user defined functions.



## Task 5.1 – Only valid trainers allowed

Implement using a trigger the constraint *‘You are allowed to teach a course only if: 1) your job type is trainer and you have been employed for at least one year or 2) you have attended the course yourself (as participant).’*

## Task 5.2 – Administrator for department

Implement the following constraints using triggers *‘A department that employs the president or a manager should also employ at least one administrator’.*

## Task 5.3 – Salary grade limitations

Implement the following constraints using triggers ‘*The llimit of a salary grade must be higher than the llimit of the next lower salary grade and the ulimit of the salary grade must be higher than the ulimit of the next lower’.*



## Task 5.4 – Trainers can start only one course a time

Implement the following constraints using triggers ‘T*he start date and known trainer uniquely identify course offerings.’*

*Note: the constraint ‘ofr\_unq’ is too strict, this does not allow multiple unknown trainers on the same start date, this unique constraint should therefore be dropped. Create a new constraint to implement the constraint, the use of a filtered index is not allowed;*

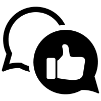
## Task 5.5 – Home based course offerings

Implement the following constraints using triggers *‘At least half of the course offerings (measured by duration) taught by a trainer must be ‘home based’.*

*Note: ‘Home based’ means the course is offered at the same location where the employee is employed.*

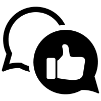
# Implement Unit tests

The constraints and functionality that you developed need to be tested. The tasks below describe what you nee to unit test as a minimum. Tests should be developed using the tSQLt framework.



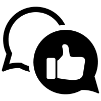
## Task 6.1 – Implement Unit tests for an information need

Implement unit tests for one of the information needs as mentioned in chapter 2. Choose one yourself.



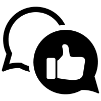
## Task 6.2 – Implement Unit tests for a declarative constraint

Implement unit tests for one of the implemented declarative constraints as mentioned in chapter 3. Choose one yourself.



## Task 6.3 – Implement Unit tests for a stored procedure

Implement unit tests for one of the developed stored procedures as mentioned in chapter 4. Choose one yourself.



## Task 6.4 – Implement Unit tests for constraint ‘Only valid trainers allowed’

Implement unit tests for the constraint as mentioned in 5.1.

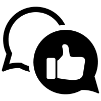
## Task 6.5 – Implement Unit tests for all other constraints

Implement unit tests for the constraint as mentioned in 5.2 till 5.5.

Add the tests of one of these constraints in your report but deliver all tests in a sql script file.

# Analyze Concurrency Control

In this part of the assignment you are going to analsye whether or the code is suited for mult-user aspects.



## Task 7.1 – Analyze and solve concurrency problem stp\_RegisterForCourse

Analyze the stored procedure printed hereafter. It implements the constraint *a trainer cannot register for a course offering taught by him- or herself*.  
  
This implementation potentially suffers from concurrency problems when run in default isolation level mode READ COMMITTED. Illustrate where and why problems may arise by describing what kind of locks are acquired (e.g. s-locks and x-locks), when they are acquired, why they are acquired and for how long they lock a resource.   
  
The code of the stored procedure is listed below:  
  
create or alter procedure stp\_RegisterForCourse

@studentCode numeric(4),

@courseCode varchar(6),

@startDatum date,

@eval numeric(1) = -1

as

BEGIN

DECLARE @savepoint varchar(128) = CAST(OBJECT\_NAME(@@PROCID) as varchar(125)) +

CAST(@@NESTLEVEL AS varchar(3))

DECLARE @startTrancount int = @@TRANCOUNT;

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

BEGIN TRY

BEGIN TRANSACTION

SAVE TRANSACTION @savepoint;

-- Student of the course cannot be the same employee as the trainer of the course

IF (select trainer from offr where course = @courseCode and starts = @startDatum) = @studentCode

RAISERROR('Person registering for this course is already provisioned to be the trainer of the course', 16, 1)

-- In case of no conflict insert new registration

insert into reg values (@studentCode, @courseCode, @startDatum, @eval)

COMMIT TRANSACTION

END TRY

BEGIN CATCH

--a rollback is possible if a transaction is "doomed", but only if @startTrancount = 0

--If @startTrancount is higher, the calling sproc/system is responsible for rolling back.

IF XACT\_STATE() = -1 AND @startTrancount = 0

BEGIN

ROLLBACK TRANSACTION

END

ELSE

BEGIN

--if the transactions is not doomed, the work of this sproc can be undone.

--Other work in the whole transaction, might still be able to be committed.

--so a rollback to savepoint is necessary and transactions counter substracted by 1 ;

IF XACT\_STATE() = 1

--transaction not doomed, but work needs to be undone.

--your not in this CATCH block without a serious cause.

BEGIN

ROLLBACK TRANSACTION @savepoint; --undo work of this sproc

COMMIT TRANSACTION; --trancount - 1

END;

END;

THROW

END CATCH;

END

Explain your analysis by giving interleaved 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 problem.

## Task 7.2 – Analyze and solve concurrency problem stp\_SetManager

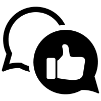
Analyze the stored procedure of task 4.2

This implementation potentially suffers from concurrency problems when run in default isolation level mode READ COMMITTED. Illustrate where and why problems may arise by describing what kind of locks are acquired (e.g. s-locks and x-locks), when they are acquired, why they are acquired and for how long they lock a resource.

Explain your analysis by giving interleaved 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 problem.

# Implement indexes

In this part of the assignment you are going to implement indexen in order to get a better queryplan and thus a better performance for your query.



## Task 8.1 – Optimize information need 1

Find a query of Task 2 which may be optimize by using indexes.

Analyze the currenct queryplan and think of an index on one or more columns, either clustered or non-clustered, that may optimize the query performance.

Describe your solution 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*

****

## Task 8.2 – Optimize information need 2

In this second part of the task you will be given a query of Task 2 by your lecturer which may be optimize by using indexes.

Analyze the currenct queryplan and think of an index on one or more columns, either clustered or non-clustered, that may optimize the query performance.

Describe your solution 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.

Add extra test data if the improvement is hard to measure given the small dataset!

# Implement history using code generation

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 developed with name HIST\_<table name> so HIST\_EMP, HIS\_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 column(s) combined with a timestamp column (type timestamp). The primary key is the only constraint of these history tables.

* a trigger needs to be developed with name TRG\_<table\_name>\_FILLHISTORY so TRG\_EMP\_FILLHISTORY, TRG\_DEPT\_FILLHISTORY etc. These trig-gers will add the inserted, updated and delete records to the history tables.

## Social Media Positive Feedback Icon Icons - Download Free Vector Icons | Noun ProjectTask 9.1 – Generate history tables

The first part of this assignment is to create the history tales that will store the history data.

The following steps need te bo taken;

* Create a working instance for one history table based on the Course database.
* Define the template for the code that need to be generated
* Write a stored procedure that generates the ‘create history table’ script for a specific table (given a table name as the input parameter). Determine where you can find (using the Information Schema Views) the metadata and retrieve the parameter values you have to use in this template to produce the specific history table.
* Test the correct generation of one history table (similar as your working instance)

The header of this stored procedure should be:

**CREATE PROCEDURE stp\_CreateHistoryTable (@table\_name sysname)**

Be sure you meet the naming convention requirement as mentioned above.

****

## Task 9.2 – Generate history triggers

The second part of this assignment is to create the history triggers that store the data modifications in the history tables.

* Create a working instance for a trigger that populates a history table. Define the template for the code that need to be generated
* Write a stored procedure that generates the ‘create trigger’ script for a specific table (given a table name as the input parameter). Determine where you can find (using the Information Schema Views) the metadata and retrieve the parameter values you have to use in this template to produce the specific history table.
* Test the correct generation of one generated trigger (similar as your working instance)
* Test the automatically record of data changes in history tables.

The header of this stored procedure should be:

**CREATE PROCEDURE stp\_CreateHistoryTrigger (@table\_name sysname)**

****

## Task 9.3 – Generate history tables and triggers for all tables

Write a stored procedure that generates all history tables and all triggers for all tables in the Course database.

Test the automatically record of data changes in history tables.

The header of this stored procedure should be:

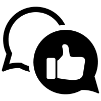
**CREATE PROCEDURE stp\_CreateHistoryTablesAndTriggersForAllTables**

# Implement Security

In this part you are going to implement some security aspects to implement these business requirements secribed in the tasks below.

Implement a suitable (minimal and maintainable) security policy in de database, no more no less. The way you do this may depend on the way you implemented the constraints listed in task.

Test your security regime. Create test users representing the user types employee and reporting service testing via “impersonation” the implementation of the security policy.



## Task 10.1 – Implement Employee account

Employees can register themselves for courses on offer using an app provided by the Human Resources department. For this purpose you should create an **employee** account. The only data the employee have full access to are the data in the REG table. Of course read access to the EMP and OFFR tables are also needed (foreign key checks require the user to have access to the referenced data).

****

## Task 10.2 – Implement Reporting account

For reporting purpose a specific reporting service account needs to be created allowing reporting tools full read access to all data. Be aware that this account also needs to see the output of the information needs of task 2.

