Materiały do przedmiotu *Introduction to databases* (Laboratory)
na kierunku Technologie Informatyczne w Logistyce w języku angielskim realizowanego w ramach projektu InfoGeolog

Author
Michał Bleja, Ph.D.

Łódź 2014
1 Entity-Relationship Modelling

Run Oracle SQL Developer Data Modeler (OSDDM).

1.1 Assignment 1

1. Creating entities

Create two entities in OSDDM: Person (pid, first_name, last_name, gender, birth_date) and Car(VIN, number_plate, mark, desc):
**Entity attributes** *(pid is an entity identifier)*

**Entity attributes** *(first_name, last_name, and gender have the varchar data type and are mandatory)*
Entity attributes (*birth_date* has the *date* data type and is mandatory)

Create the *Car* entity in the same way. All attributes of *Car* should have the *varchar* data type. The *VIN* attribute is the entity identifier.

The complete entities

2. Creating relationships

It is assumed that a relationship between *Person* and *Car* has the cardinality one-to-many. The relationship is mandatory on the *Car* side.
Relationship properties
3. Translating entity relationship diagrams into relations
Generating the SQL code – DDL statements
4. Sample ERD diagram.
Open the companyERD.dmd file in OSDDM. Analyse the entities and relationships between them.

5. Create an entity relationship diagram for the following situation:
A university needs to store information about faculties, employees, students and courses:

- Faculties have a faculty number, a name, an address, a dean (an employee).
- Employees and students have a nin (national insurance number – it is an identifier), a first name, a last name, an address, a date of birth, and a gender.
- Employees work in one or more faculties on a given position (each position has a position ID, a name). It is required to store information about salaries, starting dates, ending dates for employees.
- Students have student numbers and can study on several directions (each direction has a direction ID, name and it is assigned to a faculty). It is required to store information about students and their directions, starting dates, and ending dates.
- Students attend many courses on each direction and get final grades. A course is attended by many students.
- Courses have a course number, a name, a supervisor (an employee).
• An employee can give many courses. An employee can give the same course in several semesters.

6. Create an entity relationship diagram for the following situation:

An online shop needs to store information about products, customers, and orders:

• Customers have a cin (customer identification number), a first name, a last name, an email, a login, a password, a phone number, an address.

• Products have a pid (product identification number), a name, a unit price, a quantity, and a category (each category has a cid (category ID), a name, and a parent category).

• Products are delivered by suppliers. A supplier can provide many products. One product can be delivered by several suppliers. Supplier have sid (supplier ID), a name, an address.

• Customers can do many orders. An order has an oid (order ID), an order date, a price, and a delivery address. It is assigned to only one customer.

• Each order involves the list of ordered products. Each ordered product has a product id, a quantity, and the price.

2 Introduction to SQL

2.1 Prerequisites

Connect to SQL Server Express 2012 and create a database:

• Run SQL Server Management Studio Express 2012.

• Connect to the SQL Server instance (It is assumed that the instance is called SQLEXPRESS. The Windows authentication mode causes that the user identity is confirmed by Windows):

• Create a new database called CompanyDB
- Open a new query window:

- Copy and run the code from the *CompanyDB.sql* (see Appendix 1) script (it creates the tables and inserts some rows to them).
2.2 The basic SELECT statement

The SELECT statement is used to retrieve columns and rows of data from one or many tables (views). The basic syntax of this statement is as follows [BOL12]:

```
SELECT [DISTINCT] select_list -- it specifies the columns of the final result set
FROM list_of_tables -- it determines the target tables and/or views
```

The optional DISTINCT keyword eliminates duplicates from the result set.

**Run the following queries:**

**Query 2.1:** Retrieve data from all rows and all columns of the Department table.

```
SELECT *
FROM Department
```

The sign * denotes all columns.

The query 2.1 is equivalent to the previous one:

```
SELECT dept_id,name,address_id
FROM Department
```

**Query 2.2:** Display the names of all employees and their gender.

```
SELECT first_name,last_name,gender
FROM Employee
```

**Query 2.3:** Retrieve all different first names.

```
SELECT DISTINCT first_name
FROM Employee
```

**Assigning aliases (auxiliary names) to columns and tables.**

Run the following queries and analyse their results.

**Query 2.4:** Assigning aliases to columns with the AS keyword.

```
SELECT emp_id AS "Employee identification number", last_name AS Surname
FROM Employee
```

**Query 2.5:** Assigning aliases to columns without the AS keyword.

```
SELECT emp_id "Employee identification number", last_name Surname
FROM Employee
```

2.3 Restricting query result

To restrict the number of rows in the query result set the WHERE clause (the selection operator) is required. The general form of a select statement with the where clause is as follows:

```
SELECT [DISTINCT] select_list
FROM list_of_tables
WHERE selection_predicates
```
Run the following queries:

**Query 2.6:** Get employees whose last name equals Smith.

```sql
SELECT * 
FROM Employee 
WHERE last_name = 'Smith'
```

**Query 2.7:** Retrieve identifiers and names of females born after January first 1980.

```sql
SELECT emp_id, first_name, last_name 
FROM Employee 
WHERE birth_date > '01.01.1980' AND gender='F'
```

### 2.4 Transact-SQL operators

Transact-SQL provides the following operators [BOL12] (Run the example queries):

#### 2.4.1 Arithmetic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
</table>
| +        | Addition (also used for string concatenation) | SELECT 2 + 8  
SELECT 'foot' + 'ball' | 10  
'football' |
| -        | Subtraction | SELECT 20 - 5 | 15 |
| *        | Multiplication | SELECT 20 * 5 | 100 |
| /        | Division | SELECT 20 / 3  
SELECT 20 / 3.0 | 6  
6.666666 |
| %        | Remainder from the division | SELECT 20 % 3  
SELECT 20 % 4 | 2  
0 |

#### 2.4.2 Comparison Operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
</table>
| =        | Equal to | IF(10 = 0) SELECT 'true'  
ELSE SELECT 'false'  
IF('ok' = 10) SELECT 'true'  
ELSE SELECT 'false' | false  
Conversion failed when converting the varchar value 'ok' to data type int. |
| <> (!=)  | Not equal to | IF('sql' <> 'sql') SELECT 'true'  
ELSE SELECT 'false' | false |
| >        | Greater than | IF('b' > 'aa') SELECT 'true'  
ELSE SELECT 'false' | true |
<p>| &lt;        | Less than | IF(10 &lt; 0) SELECT 'true' | true |</p>
<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt;=</code></td>
<td>Greater then or equal to</td>
<td><code>IF('bcd' &gt;= 'bcdaa') SELECT 'true' ELSE SELECT 'false'</code></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less then or equal to</td>
<td><code>IF(10 &lt;= 15) SELECT 'true' ELSE SELECT 'false'</code></td>
</tr>
<tr>
<td><code>IN</code></td>
<td>TRUE if a value equals one of a list of values</td>
<td><code>SELECT first_name, last_name FROM Employee WHERE emp_id IN (1, 3, 5)</code></td>
</tr>
<tr>
<td><code>EXISTS</code></td>
<td>TRUE if a subquery returns some rows</td>
<td><code>SELECT * FROM Department WHERE EXISTS (SELECT 1) //Get departments with at least one employee</code></td>
</tr>
<tr>
<td><code>ALL</code></td>
<td>TRUE if all comparisons of the scalar value to the values of the list are true. It must be proceeded by <code>=</code>, <code>&lt;&gt;</code>, <code>&gt;</code>, <code>&gt;=</code>, <code>&lt;</code>, <code>&lt;=</code> and followed by a subquery.</td>
<td><code>SELECT * FROM Employee WHERE emp_id &gt;= ALL (SELECT emp_id FROM Employee)</code></td>
</tr>
<tr>
<td><code>SOME</code></td>
<td>TRUE if some comparison of the scalar value to the values of the list are true. It must be preceded by <code>=</code>, <code>&lt;&gt;</code>, <code>&gt;</code>, <code>&gt;=</code>, <code>&lt;</code>, <code>&lt;=</code> and followed by a subquery.</td>
<td><code>SELECT * FROM Employee WHERE birth_date &gt; ANY (SELECT birth_date FROM Employee)</code></td>
</tr>
<tr>
<td><code>BETWEEN</code></td>
<td>TRUE if the value of expression is within a range</td>
<td><code>SELECT first_name, last_name FROM Employee WHERE birth_date BETWEEN '01.01.1980' and '31.12.1981'</code></td>
</tr>
</tbody>
</table>
| `LIKE`    | TRUE if the value of expression matches a pattern. The pattern can involve special characters:  
  - underscore sign (_) - any single character  
  - percent sign (%) - zero or more characters  
  - `[]` - any single character inside the given range (`[f-k]`) or set (`[fgijk]`)  
  - `[^]` - any single character outside the given range (`[^f-k]`) or set (`[^fgijk]`) | `Get employees with the word 'gmail' anywhere in the email address`  
  `SELECT * FROM Employee WHERE email LIKE '%gmail%` |
| `IS (NOT) NULL` | Tests for null | `Get employees without email addresses`  
  `SELECT * FROM Employee WHERE email IS NULL` |
2.4.3 Logical operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AND</strong></td>
<td>TRUE if both arguments are true. It requires two boolean expression.</td>
<td><code>SELECT emp_id,first_name,last_name FROM Employee WHERE emp_id&gt;10 AND birth_date&gt;'01.01.1980'</code></td>
<td></td>
</tr>
<tr>
<td><strong>OR</strong></td>
<td>TRUE if at least one argument is true. It requires two boolean expression.</td>
<td><code>SELECT emp_id,first_name,last_name FROM Employee WHERE gender='F' AND last_name LIKE 'B%'</code></td>
<td></td>
</tr>
<tr>
<td><strong>NOT</strong></td>
<td>TRUE if its argument is false (otherwise FALSE)</td>
<td><code>SELECT * FROM Department WHERE name NOT LIKE '[A-K]%'</code></td>
<td></td>
</tr>
</tbody>
</table>

String concatenation operator

The operator + allows to concatenate two string expressions.

**Query 2.8:** Using concatenation operator.

```
SELECT first_name + ' ' + last_name AS Name, birth_date AS 'Date of birth'
FROM Employee
```

### 2.5 Sorting query result

T-SQL provides the ORDER BY clause to sort the rows retrieved by a query. It has the following form [BOL12]:

- **ORDER BY expression(s) [ASC | DESC]** - it arranges the rows based on the values of expression(s) which usually specifies a column name. The ASC keyword arranges the rows in ascending order. DESC arranges the rows in descending order. ASC is the default.

**Query 2.9:** Select all information about all women. Arrange the result based on last name in descending alphabetical order.

```
SELECT *
FROM Employee
WHERE gender='F'
ORDER BY last_name DESC
```

### 2.6 Assignment 2

Write the following queries in T-SQL.

1. Find the names of all the men born after May first 1985.

2. Find all departments whose name starts with the letter 'I'. Display the result in the following way:

<table>
<thead>
<tr>
<th>Department number</th>
<th>Department name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IT</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
3. Find all different street names.

4. Find employees born between January 1, 1970 and December 31, 1980 who have an email.
   Display the result in the following way:

<table>
<thead>
<tr>
<th>Name</th>
<th>Date of birth</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Smith</td>
<td>8.08.1980</td>
<td><a href="mailto:JSMITH@COMPEDU.EU">JSMITH@COMPEDU.EU</a></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

5. Find the names of all the men and their date of birth. Arrange the result in alphabetical order: ascending order based on last name and descending order based on date of birth.

6. Find identification numbers and salaries of currently employed workers earning more than 3000 (Hint: Use the Employment table, currently employed workers have the null value in the end_date column). Display the result in the following form:

<table>
<thead>
<tr>
<th>Employee identification number</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3800</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

7. Find employee identification numbers and task names if a given employee misses the deadline for his/her task (Hint: Use the Task table, compare the values stored in the end_date and deadline columns).

8. Find all distinct postal codes which the first two digits are outside the range [5-9].

9. Find identification numbers and names of all departments. Sort the result in ascending order based on department names.

10. Find the names of all employees who do not have the manager.

### 3 SQL Data Definition Language

#### 3.1 Prerequisites

- Connect to an instance of SQL Server using SQL Server Management Studio Express 2012.
- Create a new database called CompanyDB.
- Launch a new query window for the CompanyDB database.

#### 3.2 Creating tables

The CREATE TABLE statement is applied to create tables in a given database.

**Example 3.1**: Create a table called Employee which involves six columns: emp_id, first_name, last_name, birth_date, gender, and email.
CREATE TABLE Employee(
  emp_id INT IDENTITY(1,1) CONSTRAINT PK_Employee_Emp_id PRIMARY KEY,
  first_name VARCHAR(25) CONSTRAINT NN_Employee_fn NOT NULL,
  last_name VARCHAR(30) NOT NULL,
  birth_date DATE NOT NULL,
  gender CHAR(1) CONSTRAINT CH_Employee_gen CHECK(gender='M' OR gender='F'),
  email VARCHAR(40) CONSTRAINT U_Employee_Email UNIQUE,
  manager_id INT,
  CONSTRAINT FK_Employee_Mgr_id FOREIGN KEY(manager_id) REFERENCES Employee(emp_id))

3.2.1 Column properties

- **IDENTITY**(1,1) – it automatically generates unique sequential numbers starting at 1 and increasing by 1

  emp_id INT IDENTITY(1,1)

- **NULL** – it allows to NULL values in the column. NULL is the default.

  manager_id INT NULL

  manager_id INT

3.2.2 Column constraints

- **PRIMARY KEY** – it enforces uniqueness for the column and prohibits NULL values in it:

  emp_id INT IDENTITY(1,1) CONSTRAINT PK_Employee_Emp_id PRIMARY KEY

- **FOREIGN KEY** – it requires only the values which occur in the corresponding referenced column:

  CONSTRACT FK_Employee_Mgr_id FOREIGN KEY(manager_id) REFERENCES Employee(emp_id)

- **UNIQUE** – guarantees that values in the column are distinct:

  email VARCHAR(40) CONSTRAINT U_Employee_Email UNIQUE

- **CHECK** – restricts the inserted values to the values which comply with the specified condition:

  gender CHAR(1) CONSTRAINT CH_Employee_gen CHECK(gender='M' OR gender='F')

- **NOT NULL** – prohibits NULL values in the column:

  first_name VARCHAR(25) NOT NULL CONSTRAINT NN_Employee_fn NOT NULL

- **DEFAULT** – allows to define a default value for the column:

  Example 3.2: Creating other tables.

  CREATE TABLE Position(
    p_id INT IDENTITY(1,1) CONSTRAINT PK_Position_P_id PRIMARY KEY,
    name VARCHAR(30) NOT NULL)

  /* constraints can be defined also after column definitions */

  CREATE TABLE Department(
    dept_id INT IDENTITY(1,1),
    name VARCHAR(50) NULL,
    CONSTRAINT PK_Department_Dept_id PRIMARY KEY(dept_id))
CREATE TABLE Employment(
    e_id int IDENTITY(1,1),
    emp_id int NOT NULL,
    dept_id int NOT NULL,
    p_id int NOT NULL,
    salary money NOT NULL,
    start_date date NOT NULL,
    end_date date NULL,
    CONSTRAINT PK_Employment_E_id PRIMARY KEY(e_id),
    CONSTRAINT FK_Employment_Department_Dept_id FOREIGN KEY(dept_id) REFERENCES Department(dept_id),
    CONSTRAINT FK_Employment_Employee_Emp_id FOREIGN KEY(emp_id) REFERENCES Employee(emp_id),
    CONSTRAINT FK_Employment_Position_P_id FOREIGN KEY(p_id) REFERENCES Position(p_id))

3.3 Altering tables

The ALTER TABLE statement is mainly applied to add, alter, or drop columns in the given database table.

Run the following statements to create two tables, one named Address and the other named Country:

CREATE TABLE Country(
    country_id int CONSTRAINT PK_Country_C_id PRIMARY KEY,
    name varchar(50) NOT NULL)

CREATE TABLE Address(
    address_id int IDENTITY(1,1) NOT NULL,
    street varchar(40) NOT NULL,
    city varchar(50) NOT NULL)

Adding columns

Example 3.3: Add the three new columns to the Address table.

ALTER TABLE Address
ADD postal_code varchar(8)

ALTER TABLE Address
ADD country_id int

ALTER TABLE Address
ADD description varchar(50)

Altering columns

Example 3.4: Alter the column Address.postal_code (change the data type of postal_code from varchar(8) to varchar(12) and set NOT NULL to it).

ALTER TABLE Address
ALTER COLUMN postal_code varchar(12) NOT NULL

Example 3.5: Set NOT NULL to Address.country_id.

ALTER TABLE Address
ALTER COLUMN country_id int NOT NULL

Example 3.6: Set the default value for Employment.start_date to the current date (The GETDATE()
function returns the current date).

```sql
ALTER TABLE EMPLOYMENT
ADD CONSTRAINT D_EMPLOYMENT_SD DEFAULT GETDATE() FOR start_date
```

Adding new constraints

**Example 3.7**: Create the primary key on the column `Address.address_id`.

```sql
ALTER TABLE Address
ADD CONSTRAINT PK_Address_A_id PRIMARY KEY(address_id)
```

**Example 3.8**: Create the foreign key on `Address.country_id` which refers to `Country.country_id`.

```sql
ALTER TABLE Address
ADD CONSTRAINT FK_Address_C_id FOREIGN KEY(country_id) REFERENCES Country(country_id)
```

**Example 3.9**: Create the `UNIQUE` constraint on `Address.street`.

```sql
ALTER TABLE Address
ADD CONSTRAINT U_Address_street UNIQUE(street)
```

Dropping columns and constraints

**Example 3.10**: Remove the column `Address.description`.

```sql
ALTER TABLE Address
DROP COLUMN description
```

**Example 3.11**: Remove the `UNIQUE` constraint `U_Address_street`.

```sql
ALTER TABLE Address
DROP CONSTRAINT U_Address_street
```

3.4 Dropping tables

The DROP TABLE statement is applied to drop database tables.

**Example 3.12**: Drop the table `Country`.

```sql
DROP TABLE Country
Could not drop object 'Country' because it is referenced by a FOREIGN KEY constraint.
```

The error results from the fact that the `country_id` column of `Country` is referenced by the foreign key of the `Address` table. In such the cases, the table `Address` or the foreign key constraint `FK_Address_C_id` must first be dropped.

**Example 3.13**: Drop the constraint `FK_Address_C_id` and the table `Country`.

```sql
ALTER TABLE ADDRESS
DROP CONSTRAINT FK_Address_C_id
```

```sql
DROP TABLE Country
```

**Example 3.14**: Drop the table `Address` (the statement below does not cause an error because the `Address` table is not referenced by any foreign key).

```sql
DROP TABLE Address
```

3.5 Assignment 3

Write the following T-SQL DDL statements:

1. Create three tables:
• *Book*(isbn, title, year_of_publication, price) with primary key on isbn and all attributes required (NOT NULL).

• *Author*(author_id, first_name, last_name, birth_date) with primary key on author_id and all attributes required.

• *BookAuthor*(isbn, author_id) with all attributes required.

2. Add the column quantity to *Book* and set the default value to 0.

3. Add the column phone to *Author* with check constraint to ensure that all phone numbers are in the format [1-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9][0-9].

4. Set the foreign key on *BookAuthor*.isbn which refers to *Book*.isbn.

5. Set the foreign key on *BookAuthor*.author_id which refers to *Author*.author_id.

6. Set the primary key on the two columns (isbn, author_id) of the *BookAuthor* table.

7. Add the column description to *BookAuthor* with the data type varchar(30).

8. Change the data type of *BookAuthor* from varchar(30) to varchar(50) and set the unique constraint to it.

9. Drop the column *BookAuthor*.description.

10. Drop the table *BookAuthor* without necessity of removing the tables *Book* and *Author*.

4 SQL Data Manipulation Language

4.1 Inserting rows

The INSERT INTO statement is applied to insert new rows into database tables. The basic syntax of this statements is as follows [BOL12]:

```sql
insert into table_name [(column_list)] values (value_list)
```

Prerequisites:

• Connect to an instance of SQL Server using SQL Server Management Studio Express 2012.

• Create a new database called *CompanyDB*.

• Launch a new query window for the *CompanyDB* database.

• Run the script *companyDB.sql*.

Example 4.1: Insert some rows to the tables *Employee*, *Department*, *Position*, *Employment*, *Country*, and *Address*.

```
--set date format to day-month-year
set dateformat dmy

--an insert statement without optional list of columns in which to insert data (the value for emp_id column is omitted because it is the identity column)
```
--insert statements with column lists
insert into Employee(first_name,last_name,birth_date,gender,email,manager_id)
values('Agnes','Braun','11.09.1982','F','ABRAUN',1)
insert into Employee(first_name,last_name,birth_date,gender,email,manager_id)
values('Emily','Lewis','15.10.1987','F','ELEWIS',2)
insert into Country(name) values('France')
insert into Country(name) values('Russia')
insert into Address values('67 Koszykowa','Warsaw','00-605',1)
insert into Address values('45 Wschodnia','Poznan','23-496',1)
insert into Address values('82 Piotrkowska','Lodz','91-388',1)
insert into Position values('human resources representative')
insert into Position values('marketing representative')
insert into Position values('marketing manager')
insert into Department(name) values('Sales')
insert into Employment values(3,2,5,8000,'05.10.2010',null)
insert into Employment(emp_id,dept_id,p_id,salary,start_date)
values(4,3,12,4000,'18.01.2012')
insert into Employment(emp_id,dept_id,p_id,salary,start_date)
values(5,4,13,15000,'05.05.2013')

The \texttt{INSERT INTO} statement can be based on the \texttt{SELECT} clause:
\texttt{INSERT INTO table\_name SELECT...}

\textbf{Example 4.2}: Create the table \textit{Emp}(id, first\_name, last\_name) and insert rows to it using \texttt{INSERT INTO} with \texttt{SELECT}.

\begin{verbatim}
CREATE TABLE Emp(
    id int CONSTRAINT PK_Emp_id PRIMARY KEY,
    first\_name varchar(25) NOT NULL,
    last\_name varchar(30) NOT NULL)

INSERT INTO EMP
SELECT e.emp\_id,e.first\_name,e.last\_name
FROM Employee e
\end{verbatim}

The insert statement can involve the output clause to return its result.

\textbf{Example 4.3}: Obtaining the result of an insert statement (\texttt{INSERTED} represents the row added by the insert or update statement).

\begin{verbatim}
--create the table Tab1 with the identity column
CREATE TABLE Tab1(
    id int identity,
    name varchar(30))
--declare the local temporary table Tab2 to store the result of insert statements
DECLARE @Tab2 TABLE(id int,name varchar(30), operation\_date datetime)
--insert some rows to Tab1 and store the result in Tab2.
INSERT Tab1
    OUTPUT INSERTED.id, INSERTED.name, GETDATE() INTO @Tab2
VALUES('test1')
INSERT Tab1
\end{verbatim}
OUTPUT INSERTED.id, INSERTED.name, GETDATE() INTO @Tab2 VALUES('test2')

--select data from both tables
SELECT * FROM Tab1
SELECT * FROM @Tab2

### 4.2 Updating rows

The UPDATE statement allows to update existing rows in a table [BOL12]:

```
UPDATE table_name
SET column_name1=value1, column_name2=value2,...
[WHERE selection_predicates]
```

If no WHERE clause is defined all rows in the table are updated.

**Example 4.4:** Increase the salary by 20% for employees earning less than 3000 and working in the first department.

```
UPDATE Employment
SET salary = salary*1.2
WHERE salary < 3000 and dept_id=1 and end_date is null
```

**Example 4.5:** Assign the first address to the employee *John Smith*.

```
UPDATE Employee
SET address_id = 1
WHERE first_name = 'John' and last_name = 'Smith'
```

**Example 4.6:** Convert all email addresses to uppercase (use the `UPPER` function).

```
UPDATE Employee
SET email = UPPER(email)
```

The update statement can involve the output clause to return information about rows affected by it.

**Example 4.7:** Obtaining the result of an update statement (`DELETED` represents the row copy affected by the update or delete statement).

```
--declare the local temporary table Tab to store the result of the update statement
DECLARE @Tab table ( id int, old_salary money, new_salary money )

UPDATE Employment
SET salary = salary * 1.1
OUTPUT inserted.e_id, inserted.salary, deleted.salary INTO @Tab
WHERE end_date is null

SELECT id AS "Employee ID", old_salary AS "Old salary", new_salary AS "New salary"
FROM @Tab;
```

Update statements involving more than one table are presented in ????

### 4.3 Deleting rows

The DELETE statement is applied to delete existing rows in a table [BOL12]:

```
DELETE FROM table_name
[WHERE selection_predicates]
```
If no WHERE clause is defined all rows are removed from the table.

**Example 4.8:** Remove employment data concerning employees who lost the job.

```sql
DELETE FROM Employment
WHERE end_date is not null
```

**Example 4.9:** Remove the row from `Employee` concerning Emily Lewis

```sql
DELETE FROM Employee
WHERE first_name = 'Emily' and last_name = 'Lewis'
```

The DELETE statement conflicted with the reference constraint "FK_Employment_Employee". The conflict occurred in database "companyDB", table "dbo.Employment", column 'emp_id'.

The error message results from the fact that the removed row is referenced by the row(s) of the `Employment` table which has the foreign key `FK_Employment_Employee`. In such the case, the proper row(s) from `Employment` must first be dropped:

```sql
-- 5 is the value Emily Lewis's identifier
DELETE FROM Employment
WHERE emp_id = 5
```

```sql
DELETE FROM Employee
WHERE first_name = 'Emily' and last_name = 'Lewis'
```

**Example 4.10:** The delete statement with the output clause (the deleted rows will be displayed on the screen).

```sql
DELETE FROM Employment
OUTPUT DELETED.*
WHERE end_date is not null and salary in (3000,4000,5000)
```

### 4.4 Assignment 4

Write the following T-SQL DML statements:

1. Insert two rows to the `Address` table.
2. Insert two rows to the `Employee` table. The employee Adam Parker should be the manager of these employees. Their addresses should correspond to addresses inserted in the first point.
3. Insert the proper rows to the `Employment` table for these employees. They should work in the IT department as junior programmers.
4. Increase the salary by 15% for the employees earning between 2000 and 4000.
5. Assign the second address to the employee Albert Foster.
6. Assign two new tasks to the employee Thomas Smith. Set the deadline to the current date.
7. Extend the deadline for the above task about a week.
8. Convert all email addresses to lowercase (use the `LOWER` function).
9. Remove the tasks assigned to the second employee.
10. Remove the employee John Tyler. First of all remove all rows which refer to this employee.
# 5 Fundamental T-SQL functions

## 5.1 Prerequisites

- Connect to an instance of SQL Server using SQL Server Management Studio Express 2012.
- Create a new database called *CompanyDB*.
- Launch a new query window for the *CompanyDB* database.

T-SQL provides the following functions [BOL12] (Run the example queries):

<table>
<thead>
<tr>
<th>Function</th>
<th>Return type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Date and time functions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GETDATE()</td>
<td>datetime</td>
<td>Returns current date and time</td>
<td>SELECT GETDATE()</td>
</tr>
<tr>
<td>DAY(date)</td>
<td>int</td>
<td>Returns the day part of the given date</td>
<td>SELECT DAY(GETDATE()) AS day</td>
</tr>
<tr>
<td>MONTH(date)</td>
<td>int</td>
<td>Returns the month part of the given date</td>
<td>SELECT MONTH(GETDATE()) AS month</td>
</tr>
<tr>
<td>YEAR(date)</td>
<td>int</td>
<td>Returns the year part of the given date</td>
<td>SELECT YEAR(GETDATE()) AS year</td>
</tr>
<tr>
<td>DATEPART(part, date)</td>
<td>int</td>
<td>Returns the specified part of the given date</td>
<td>SELECT DATEDIFF(month, GETDATE())</td>
</tr>
<tr>
<td>DATENAME(part, date)</td>
<td>nvarchar</td>
<td>Returns the specified part of the given date</td>
<td>SELECT DATENAME(weekday, GETDATE()) weekday, DATENAME(month, GETDATE()) month</td>
</tr>
<tr>
<td>DATEDIFF(datepart, date1, date2)</td>
<td>int</td>
<td>Returns number of datepart between dates date1 and date2</td>
<td>DECLARE @d1 datetime = GETDATE(); DECLARE @d2 datetime = '2010-05-12 14:15:09'; SELECT DATEDIFF(day, @d1, @d2) AS &quot;d2-d1&quot;;</td>
</tr>
<tr>
<td>DATEADD(datepart, data type)</td>
<td>Returns the date</td>
<td>SELECT DATEADD(year, 1, GETDATE())</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Parameter(s)</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>DATEADD</td>
<td>month, -2, GETDATE(), DATEADD(day, 10, GETDATE()), DATEADD(quarter, -3, GETDATE())</td>
<td>Returns date plus number of datepart</td>
<td>SELECT DATEADD(hour, 1, GETDATE()), DATEADD(minute, -2, GETDATE()), DATEADD(second, 10, GETDATE())</td>
</tr>
<tr>
<td>ISDATE</td>
<td>arg</td>
<td>Returns 1 if arg is correct date, time, or datetime value; otherwise, 0</td>
<td>SET DATEFORMAT dmy IF ISDATE('25-11-2007 15:21:38') = 1 PRINT 'CORRECT' ELSE PRINT 'INCORRECT'</td>
</tr>
<tr>
<td>LOWER</td>
<td>char_data</td>
<td>varchar/nvarchar</td>
<td>Selects char_data with all characters lowercase</td>
</tr>
<tr>
<td>UPPER</td>
<td>char_data</td>
<td>varchar/nvarchar</td>
<td>Selects char_data with all characters uppercase</td>
</tr>
<tr>
<td>LEN</td>
<td>char_data</td>
<td>int</td>
<td>Returns the number of characters in char_data</td>
</tr>
<tr>
<td>SUBSTRING</td>
<td>char_data, start_char, length</td>
<td>varchar/nvarchar/varbinary</td>
<td>Returns the part of char_data which starts with start_char and has the length length</td>
</tr>
<tr>
<td>REPLACE</td>
<td>char_data, pattern, substitute</td>
<td>varchar/nvarchar</td>
<td>Returns char_data in which every occurrence of pattern is replaced by substitute</td>
</tr>
<tr>
<td>LTRIM</td>
<td>char_data</td>
<td>varchar/nvarchar</td>
<td>Returns char_data without the leading blanks</td>
</tr>
<tr>
<td>RTRIM</td>
<td>char_data</td>
<td>varchar/nvarchar</td>
<td>Returns char_data without the trailing blanks</td>
</tr>
<tr>
<td>LEFT</td>
<td>char_data, number</td>
<td>varchar/nvarchar</td>
<td>Returns the left part of char_data involving number characters</td>
</tr>
<tr>
<td>RIGHT</td>
<td>char_data, number</td>
<td>varchar/nvarchar</td>
<td>Returns the right part of char_data involving number characters</td>
</tr>
<tr>
<td>ABS</td>
<td>number</td>
<td>the type of number</td>
<td>Returns the absolute value of number</td>
</tr>
<tr>
<td>Function</td>
<td>Parameters</td>
<td>Description</td>
<td>Examples</td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
<td>-------------------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>CEILING((n))</td>
<td>(n)</td>
<td>Returns the smallest integer greater than or equal to (n)</td>
<td>SELECT CEILING(5.5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT CEILING(-5.5)</td>
</tr>
<tr>
<td>FLOOR((n))</td>
<td>(n)</td>
<td>Returns the largest integer equal to or less than (n)</td>
<td>SELECT FLOOR(5.5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT FLOOR(-5.5)</td>
</tr>
<tr>
<td>POWER((n1,n2))</td>
<td>(n1), (n2)</td>
<td>Returns (n1) raised to the (n2) power</td>
<td>SELECT POWER(-10, 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT POWER(3.3, 5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT POWER(-10, -2)</td>
</tr>
<tr>
<td>SQRT((n))</td>
<td>(n)</td>
<td>Returns the square root of (n)</td>
<td>SELECT SQRT(36)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT SQRT(50.55)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT SQRT(-9)</td>
</tr>
<tr>
<td>ROUND((n1,n2))</td>
<td>(n1), (n2)</td>
<td>Returns (n1) rounded to (n2) places</td>
<td>SELECT ROUND(36.4567, 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT ROUND(345.9995, 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT ROUND(123.56, -1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT ROUND(123.56, -2)</td>
</tr>
<tr>
<td>RAND(([seed]))</td>
<td>(seed)</td>
<td>Returns a random number from 0 through 1</td>
<td>SELECT RAND()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT RAND(30)</td>
</tr>
<tr>
<td>LOG((n))</td>
<td>(n)</td>
<td>The functions return the natural and 10-base logarithm of (n), correspondingly</td>
<td>SELECT LOG(EXP(100)) --EXP(n) returns e raised to the n-th power</td>
</tr>
<tr>
<td>LOG10((n))</td>
<td>(n)</td>
<td></td>
<td>SELECT LOG10(1000)</td>
</tr>
<tr>
<td>SIN((ang))</td>
<td>(ang)</td>
<td>The functions return the trigonometric sine and cosine of the (ang) angle (in radians), correspondingly.</td>
<td>SELECT SIN(PI())/6</td>
</tr>
<tr>
<td>COS((ang))</td>
<td>(ang)</td>
<td></td>
<td>SELECT COS(PI())/3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>--PI() returns the value of PI</td>
</tr>
<tr>
<td>TAN((ang))</td>
<td>(ang)</td>
<td>The functions return the trigonometric tangent and cotangent of the (ang) angle (in radians), correspondingly.</td>
<td>SELECT TAN(PI())/4</td>
</tr>
<tr>
<td>COT((ang))</td>
<td>(ang)</td>
<td></td>
<td>SELECT COT(PI())/4</td>
</tr>
<tr>
<td>CAST((expr) AS (dt) ([(len)]))</td>
<td>(expr), (dt) ([(len)])</td>
<td>The functions convert the (expr) expression to the specified (dt) data type. (len) denotes the length of (dt) (the default value is 30)</td>
<td>SELECT CAST(GETDATE() AS varchar(12))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CONVERT(varchar(12), GETDATE())</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT CAST(658.3834223 AS money)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CONVERT(money, 658.3834223)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT CAST(123.4567 AS int)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CONVERT(int, 123.4567)</td>
</tr>
<tr>
<td>CURRENT_USER USER_NAME()</td>
<td></td>
<td>The functions return the current user name</td>
<td>SELECT CURRENT_USER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT USER_NAME()</td>
</tr>
<tr>
<td>ISDATE((expr))</td>
<td>(expr)</td>
<td>Returns 1 if the data type of (expr) is date,</td>
<td>SELECT ISDATE(GETDATE())</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT ISDATE(5)</td>
</tr>
</tbody>
</table>
### 5.2 Aggregate functions

SQL Server provides also so called *aggregate functions* that allow to make a summary of the data. The functions act on a set of values and return a single value. The main aggregate functions are COUNT, MIN, MAX, SUM, AVG (see the Lecture: Section 8.1.6 and [BOL12]).

Run the following queries:

**Query 5.1**: Calculate the number of rows in the `Employee` table.

```sql
SELECT COUNT(*) 'Number of employees'
FROM Employee
```

**Query 5.2**: How many different first names do exist?

```sql
SELECT COUNT(DISTINCT first_name)
FROM Employee
```

**Query 5.3**: Calculate the average salary of all current employees.

```sql
SELECT AVG(salary)
FROM Employment
WHERE end_date IS NULL
```

**Query 5.4**: Determine the current lowest and highest salary in the second department.

```sql
SELECT MIN(salary), MAX(salary)
FROM Employment
WHERE dept_id=2 AND end_date IS NULL
```

**Query 5.5**: Calculate the sum of all employee identification numbers.

```sql
SELECT SUM(emp_id)
FROM Employee
```
5.3 Assignment 5
Write the following T-SQL statements:

1. Calculate the number of rows in the Address table.
2. Calculate the number of all women.
3. Determine the current lowest start date in the first department.
4. Calculate the sum of salaries in the second department.
5. How many different last names do exist?
6. Determine the current highest salary for IT managers.
7. How many employees started work in January?
8. Calculate the number of tasks assigned to the first employee.
9. Determine the number of employees working in the HR department.
10. Determine the number of employees supervised directly by John Smith.

6 Complex queries

6.1 Prerequisites
• Connect to an instance of SQL Server using SQL Server Management Studio Express 2012.
• Create a new database called CompanyDB.
• Launch a new query window for the CompanyDB database.

6.2 Joins
Joins allow to retrieve data from two or more tables/views. SQL Server provides the following type of joins (see the Lecture: Section 8.1.8 and [BOL12]):

• inner join (join),
• left outer join (left join),
• right outer join (right join),
• full outer join (full join),
• cross join,
• self join.

Run the following queries and analyse their results:

Query 6.1: Inner join – Select the names of all employees living in Krakow Display also the street names.

--The INNER keyword is optional.
SELECT first_name, last_name, street
FROM Employee e INNER JOIN Address a
ON e.address_id = a.address_id
WHERE a.city = 'Krakow'

Query 6.2: The join condition can be also specified in the WHERE clause.
SELECT first_name, last_name, street
FROM Employee e, Address a
WHERE e.address_id = a.address_id AND a.city = 'Krakow'

Query 6.3: LEFT OUTER JOIN – Select the names of all employees and their addresses, including those not yet assigned to any addresses.

--The OUTER keyword is optional.
SELECT first_name, last_name, street, city
FROM Employee e LEFT OUTER JOIN Address a ON e.address_id = a.address_id

Query 6.4: RIGHT OUTER JOIN – Select the names of all employees and their addresses, including addresses not yet assigned to any employees.

--The OUTER keyword is optional.
SELECT first_name, last_name, street, city
FROM Employee e RIGHT OUTER JOIN Address a ON e.address_id = a.address_id

Query 6.5: FULL OUTER JOIN – Select the names of all employees and their addresses, including those not yet assigned to an address as well as those addresses that are not matched to any employees.

--The OUTER keyword is optional.
SELECT first_name, last_name, street, city
FROM Employee e FULL OUTER JOIN Address a ON e.address_id = a.address_id

Query 6.6: SELF JOIN – Select the names of all employees and their managers.

SELECT e1.first_name + ' ' + e1.last_name + ' is subordinate to ' +
      e2.first_name + ' ' + e2.last_name AS 'Subordinations'
FROM Employee e1 JOIN Employee e2 ON e1.manager_id = e2.emp_id

Query 6.7: Cross join.

SELECT a.city, c.name
FROM Address a CROSS JOIN Country c

The query above is semantically equivalent to the following query:

SELECT a.city, c.name
FROM Address a, Country c

Query 6.8: Joining three tables – Select the names of all current employees and their department names.

SELECT e.first_name, e.last_name, d.name
FROM Employee e JOIN Employment em ON e.emp_id = em.emp_id
    JOIN Department d ON em.dept_id = d.dept_id
WHERE em.end_date IS NULL

6.3 GROUP BY and HAVING clause

The syntax of select statement involving GROUP BY and HAVING is as follows [BOL12]:

SELECT select_list
FROM list_of_tables
[WHERE search_conditions_for_rows]
[GROUP BY grouping_expressions]
[HAVING search_conditions_for_groups]
[ORDER BY order_expression [ ASC | DESC ]]

```sql
SELECT first_name, last_name, street
FROM Employee e, Address a
WHERE e.address_id = a.address_id AND a.city = 'Krakow'

Query 6.3: LEFT OUTER JOIN – Select the names of all employees and their addresses, including those not yet assigned to any addresses.

--The OUTER keyword is optional.
SELECT first_name, last_name, street, city
FROM Employee e LEFT OUTER JOIN Address a ON e.address_id = a.address_id

Query 6.4: RIGHT OUTER JOIN – Select the names of all employees and their addresses, including addresses not yet assigned to any employees.

--The OUTER keyword is optional.
SELECT first_name, last_name, street, city
FROM Employee e RIGHT OUTER JOIN Address a ON e.address_id = a.address_id

Query 6.5: FULL OUTER JOIN – Select the names of all employees and their addresses, including those not yet assigned to an address as well as those addresses that are not matched to any employees.

--The OUTER keyword is optional.
SELECT first_name, last_name, street, city
FROM Employee e FULL OUTER JOIN Address a ON e.address_id = a.address_id

Query 6.6: SELF JOIN – Select the names of all employees and their managers.

SELECT e1.first_name + ' ' + e1.last_name + ' is subordinate to ' +
      e2.first_name + ' ' + e2.last_name AS 'Subordinations'
FROM Employee e1 JOIN Employee e2 ON e1.manager_id = e2.emp_id

Query 6.7: Cross join.

SELECT a.city, c.name
FROM Address a CROSS JOIN Country c

The query above is semantically equivalent to the following query:

SELECT a.city, c.name
FROM Address a, Country c

Query 6.8: Joining three tables – Select the names of all current employees and their department names.

SELECT e.first_name, e.last_name, d.name
FROM Employee e JOIN Employment em ON e.emp_id = em.emp_id
    JOIN Department d ON em.dept_id = d.dept_id
WHERE em.end_date IS NULL

6.3 GROUP BY and HAVING clause

The syntax of select statement involving GROUP BY and HAVING is as follows [BOL12]:

SELECT select_list
FROM list_of_tables
[WHERE search_conditions_for_rows]
[GROUP BY grouping_expressions]
[HAVING search_conditions_for_groups]
[ORDER BY order_expression [ ASC | DESC ]]
```
The **GROUP BY** clause divides the rows of queried tables/views into groups. Aggregate functions are commonly used with the **GROUP BY** clause.

The **HAVING** operator makes a selection on groups of returned rows. It retrieves only the groups for which the specified conditions hold.

Run the following queries and analyse their results:

**Query 6.9:** Select department names and the average salary for each of them.

```sql
SELECT d.name AS 'Department name', AVG(em.salary) AS 'Average salary'
FROM Employment em JOIN Department d ON em.dept_id=d.dept_id
WHERE em.end_date IS NULL
GROUP BY d.name
```

**Query 6.10:** Select department name if the number of its current employees is greater than 5. Display also the number of employees.

```sql
SELECT d.name AS 'Department name', COUNT(*) AS 'The number of employees'
FROM Employment em JOIN Department d ON em.dept_id=d.dept_id
WHERE em.end_date IS NULL
GROUP BY d.name
HAVING COUNT(*)>5
```

**Query 6.11:** Determine employee identification number if he/she has more than one row in the Employment table. Arrange the result based on identification numbers in descending order.

```sql
SELECT emp_id AS 'Employee identification number'
FROM Employment
GROUP BY emp_id
HAVING COUNT(*)>2
ORDER BY emp_id DESC
```

**6.4 Assignment 6**

1. Select department names and the sum of current salaries in each department.
2. Select country names and the number of cities in each country.
3. Select the names of all current employees and their salaries.
4. Select the minimum and maximum current salary in the HR department.
5. Select the names of all current employees, their cities, and countries.
6. Select a department name if the average salary in this department is greater than 2500.
7. Select the names of all current employees, their birth days, and salaries. Sort the result by last name and birth date in ascending order, and display it in the following way:

<table>
<thead>
<tr>
<th>Employee</th>
<th>Birth date</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Smith</td>
<td>8.08.1980</td>
<td>IT</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
8. Print the names of all current employees, their department names, positions and the number of workdays. Do it only for employees working longer than 100 days.

<table>
<thead>
<tr>
<th>Employee</th>
<th>Department</th>
<th>Position</th>
<th>Actual number of workdays</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Smith</td>
<td>Management</td>
<td>director</td>
<td>1069</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

9. Select position names and the number of current employees having the given position.

10. Determine the average age (in days) of all current employees for each gender.

11. Print the names of all current employees whose salary exceeded 4000 on January 1, 2000.

12. Select the names of all departments and the average number of workdays in each of them.

13. Select the names of positions which are currently not occupied.

14. Select the names of all departments and their countries in the following way:

<table>
<thead>
<tr>
<th>Department</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management</td>
<td>Poland</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

15. Select the manager identifiers and the number of their subordinates.

### 7 Subqueries

#### 7.1 Prerequisites

- Connect to an instance of SQL Server using SQL Server Management Studio Express 2012.
- Create a new database called `CompanyDB`.
- Launch a new query window for the `CompanyDB` database.

A subquery is a select statement which is nested in another statement. Run the following queries and analyse their result:

**Query 7.1:** Select employees earning more than Thomas Smith.

```sql
SELECT e.first_name, e.last_name
FROM Employee e JOIN Employment em ON e.emp_id=em.emp_id
WHERE em.salary >
  (SELECT em2.salary
   FROM Employee e2 JOIN Employment em2 ON e2.emp_id=em2.emp_id
   WHERE e2.first_name='Thomas' AND e2.last_name='Smith')
```

uncorrelated subquery without aggregate
Query 7.2: Select the oldest woman.

```
SELECT first_name, last_name
FROM Employee
WHERE birth_date =
  (SELECT MIN(birth_date)
   FROM Employee
   WHERE gender='F')
```

uncorrelated subquery with aggregate

Query 7.3: Get employees which are the age of their managers.

```
SELECT e1.first_name, e1.last_name
FROM Employee e1
WHERE e1.birth_date =
  (SELECT e2.birth_date
   FROM Employee e2
   WHERE e2.emp_id = e1.manager_id)
```

correlated subquery without aggregate

Query 7.4: Get employees earning more than the average salary in their departments.

```
SELECT e1.first_name, e1.last_name
FROM Employee e1 JOIN Employment em1 ON e1.emp_id = em1.emp_id
WHERE em1.end_date IS NULL AND em1.salary >
  (SELECT AVG(em2.salary)
   FROM Employment em2
   WHERE em2.end_date IS NULL AND em2.dept_id=em1.dept_id)
```

correlated subquery with aggregate

### 7.2 Assignment 7

Write the following queries:

1. Print the name of each current employee who has the highest salary. Arrange the result in ascending order based on last name.

2. Select the names of all departments which currently employ the most workers.

3. Print the names of each current employee who is employed the longest.

4. Select the name of every region having the most cities.

5. Print the names of each current employee whose salary exceeds the average salary in the IT department.
6. Find employees having more than one row in the Employment table and display their first name, last name, and the number of rows. Sort the result by last name and the number of rows in descending order.

7. Print the name of each current employee if he or she earns above 80% of their manager’s salary.

8. Print the names of current employees who are men and are older than Adam Parker and women earning above 3000.

9. Select cities in which the average salary is the highest.

10. Get all information about employees who takes the smallest number of tasks.

11. Increase the salary by 15% for the employees earning between 3500 and 80% of their manager’s salary.

12. Increase the salary by 10% for each current employee who was employed at least two times.

13. Get the names of all employees who gave at least two tasks. Use the EXISTS operator with the proper subquery to write this query.

14. Write the query above with the join operator to avoid query nesting.

### 8 Transactions in SQL Server

A transaction is a single unit of work involving the sequence of database operations. It must respect the following principles (see the Lecture: Section 10 and [BOL12]):

- atomicity,
- consistency,
- isolation,
- durability.

An example of a transaction in SQL Server:

```
BEGIN TRANSACTION
SET DATEFORMAT dmy
DECLARE @tab TABLE(id int)
INSERT INTO Employee
    OUTPUT INSERTED.emp_id INTO @tab
VALUES('Michael','Torn','20.05.1985','M','MTORN',5,null)
DECLARE @emp_id int = (SELECT id FROM @tab)
INSERT INTO Employment VALUES(@emp_id,3,12,3800,'23.01.2010',null)
COMMIT
```
The transaction above persists in the database if all of its statements are successful. If at least one of them results in a failure, then the transaction is rolled back.

The SQL Server supports the following transaction isolation level [BOL12]:

- read uncommitted,
- read committed (it is the default),
- repeatable read,
- serializable,
- snapshot.

The following T-SQL statement allows to set a transaction isolation level [BOL12]:

```
SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ |
SERIALIZABLE | SNAPSHOT }
```

To check the isolation level use the DBCC `useroptions` statement.

The `SYS.DM_TRAN_LOCKS` dynamic management view contains information about currently active locks. It presents resources (`resource_database_id`, `resource_type`) on which locks are acquired, lock types (`request_mode`), session identifiers (`request_session_id`) requesting the given resources. More details on `SYS.DM_TRAN_LOCKS` can be found in [BOL12].

### 8.1 Assignment 8

Run the script below:

```sql
use master
go
create database dbLocks
go
create login testLogin with password = 'testLogin',check_policy=off,default_database=dbLocks
go
use dbLocks
go
create user testUser for login testLogin
go
create table Table1(id int identity,number int,[desc] char(200))
go
insert into Table1 values(CAST(RAND()*100 as int),'some description')
go 1000
grant select,insert,update,delete on Table1 to testUser
```

Logon as `testLogin` to the server instance to work as `testUser` in the `dbLocks` database. Open two query windows in SQL Server Management Studio Express 2012.

Logon as an administrator to the server instance to work as `dbo` in `dbLocks`. Open a query window in SQL Server Management Studio Express 2012.
Perform the following scenarios:

1. Read uncommitted (*dirty reads*).

<table>
<thead>
<tr>
<th>The first session of testUser</th>
<th>The second session of testUser</th>
<th>Session of thedbo user</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBCC useroptions SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED</td>
<td>DBCC useroptions</td>
<td>SELECT * FROM SYS.DM_TRAN_LOCKS WHERE resource_database_id = DB_ID('dbLocks')</td>
</tr>
<tr>
<td>BEGIN TRAN SELECT * FROM Table1 WHERE id=100</td>
<td>BEGIN TRAN UPDATE Table1 SET number=number+10 WHERE id=100</td>
<td>SELECT * FROM SYS.DM_TRAN_LOCKS WHERE resource_database_id = DB_ID('dbLocks')</td>
</tr>
<tr>
<td>SELECT * FROM Table1 WHERE id=100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--It can read uncommitted modifications</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROLLBACK</td>
<td></td>
</tr>
</tbody>
</table>

2. Read committed (*no dirty reads*)

<table>
<thead>
<tr>
<th>The first session of testUser</th>
<th>The second session of testUser</th>
<th>Session of the dbo user</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBCC useroptions SET TRANSACTION ISOLATION LEVEL READ COMMITTED</td>
<td>DBCC useroptions</td>
<td>SELECT * FROM SYS.DM_TRAN_LOCKS WHERE resource_database_id = DB_ID('dbLocks')</td>
</tr>
<tr>
<td>BEGIN TRAN SELECT * FROM Table1</td>
<td>BEGIN TRAN SELECT * FROM Table1</td>
<td></td>
</tr>
<tr>
<td>BEGIN TRAN SELECT * FROM Table1</td>
<td>--Why can it read?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UPDATE Table1 SET number=number+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE id=100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SELECT * FROM SYS.DM_TRAN_LOCKS WHERE resource_database_id = DB_ID('dbLocks')</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SELECT * FROM Table1</td>
<td>--Why can't it read?</td>
</tr>
<tr>
<td></td>
<td>ROLLBACK</td>
<td></td>
</tr>
</tbody>
</table>
3. **Read committed (nonrepeatable reads)**

<table>
<thead>
<tr>
<th>The first session of testUser</th>
<th>The second session of testUser</th>
<th>Session of the dbo user</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BEGIN TRAN</strong> &lt;br&gt; <strong>UPDATE</strong> Table1 SET &lt;br&gt; number=number-40 &lt;br&gt; <strong>WHERE</strong> id=80</td>
<td><strong>UPDATE</strong> Table1 SET &lt;br&gt; number=number+15 &lt;br&gt; <strong>WHERE</strong> id=3 &lt;br&gt; <strong>COMMIT</strong></td>
<td><strong>SELECT</strong> * FROM Table1 &lt;br&gt; <strong>UPDATE</strong> Table1 SET &lt;br&gt; number=number+50 &lt;br&gt; <strong>WHERE</strong> id=120 &lt;br&gt; <strong>BEGIN TRAN</strong> &lt;br&gt; <strong>UPDATE</strong> Table1 SET number=number+50 &lt;br&gt; <strong>WHERE</strong> id=12</td>
</tr>
</tbody>
</table>

| **BEGIN TRAN** <br> **SELECT** * FROM Table1 | **UPDATE** Table1 SET <br> number=number+15 <br> **WHERE** id=3 <br> **COMMIT** | **SELECT** * FROM Table1 <br> **UPDATE** Table1 SET <br> number=number+50 <br> **WHERE** id=120 |

| **SELECT** * FROM Table1 | **UPDATE** Table1 SET <br> number=number+15 <br> **WHERE** id=3 <br> **COMMIT** | **SELECT** * FROM Table1 <br> **UPDATE** Table1 SET <br> number=number+50 <br> **WHERE** id=120 |

4. Set the **transaction isolation level** to **repeatable read** in the first session and perform the steps from the point 2. Comment the obtained results.

5. Consider the following scenario:

<table>
<thead>
<tr>
<th>The first session of testUser</th>
<th>The second session of testUser</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SET TRANSACTION ISOLATION LEVEL READ COMMITTED</strong></td>
<td><strong>SET TRANSACTION ISOLATION LEVEL READ COMMITTED</strong></td>
</tr>
<tr>
<td><strong>BEGIN TRAN</strong> &lt;br&gt; <strong>UPDATE</strong> Table1 SET number=number+50 &lt;br&gt; <strong>WHERE</strong> id=120</td>
<td><strong>BEGIN TRAN</strong> &lt;br&gt; <strong>UPDATE</strong> Table1 SET number=number+50 &lt;br&gt; <strong>WHERE</strong> id=12 &lt;br&gt; <strong>--It must wait because the update statement requires the full table scan. This operation cannot be performed because the first session has exclusive lock on the row with id=120.</strong></td>
</tr>
</tbody>
</table>

6. Create (as **dbo**) an index on **Table1.id** and repeat the steps from the point 5. What do you notice?

7. Create an index on **Table1.number** and perform the steps below. It presents so called **phantom reads**:
The first session of testUser | The second session of testUser | Session of the dbo user
---|---|---
```
BEGIN TRAN
UPDATE Table1 SET
  number=number+300
WHERE number<10
```
```
BEGIN TRAN
UPDATE Table1 SET number=1
WHERE number>90
COMMIT
```
```
SELECT *
FROM Table1
WHERE number<10
```
```
--Are there any numbers less than 10?
```

8. Set the transaction isolation level to **serializable** in the first session and perform the steps from the point 7.

9. **Snapshot isolation level (Perform the following statements).**

<table>
<thead>
<tr>
<th>The first session of testUser</th>
<th>The second session of testUser</th>
<th>Session of the dbo user</th>
</tr>
</thead>
</table>
| ```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
GO
BEGIN TRAN
SELECT number FROM Table1
WHERE id=10
UPDATE Table1 SET
  number=number+50
WHERE id=10
```
| ```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
GO
BEGIN TRAN
SELECT number FROM Table1
WHERE id=10
```
| ```
--It sees the value that was before the update statement in the first session
```

10. **Deadlocks (Perform the following statements).**

<table>
<thead>
<tr>
<th>The first session of testUser</th>
<th>The second session of testUser</th>
</tr>
</thead>
</table>
| ```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
GO
BEGIN TRAN
UPDATE Employment SET salary=salary+500
WHERE emp_id=3
```
| ```
```
```
```
9 Views and indices

9.1 Prerequisites

- Connect to an instance of SQL Server using SQL Server Management Studio Express 2012.
- Create a new database called *CompanyDB*.
- Launch a new query window for the *CompanyDB* database.

A view is a named query whose definition is stored in the database. The rows of the view are retrieved from the tables and/or views on which it is based.

The basic syntax of the CREATE VIEW statement is as follows [BOL12]:

```
CREATE VIEW view_name
AS select_statement
[ WITH CHECK OPTION ]
```

9.2 Assignment 9.1

1. Create the *V_Emp* view which stores the identifiers, names and salaries of current employees whose salary exceeds 3500, together with their department identifiers.
2. Display data from all rows and all columns of the *V_Emp* view.
3. Use the *V_Emp* view to select the last names and salaries of employees whose last name contains the letter 'a' or 'A'.
4. Use the view to increase the salary by 10% for the employees working in the first department.
5. Use the view to set the salary to 5000 and last name to *Braun* for the employee *Adam Parker*. Is it possible?
6. Drop the view. Create again the *V_Emp* view with the *check* option.
7. Use the view to increase the salary by 5% for the employees working in the second department.
8. Use the view to set the salary to 2000 for the employee *Adam Parker*. Is it possible?
9. Create a new *V_Emp* view with the *SCHEMABINDING* clause (this clause prevents
modifications of the view tables which influence its definition):

```sql
DROP VIEW V_Emp
CREATE VIEW dbo.V_Emp
WITH SCHEMABINDING
AS SELECT e.emp_id,first_name, last_name, salary, dept_id
FROM dbo.Employee e JOIN dbo.Employment em ON e.emp_id=em.emp_id
WHERE em.end_date IS NULL AND salary>3500
```

10. Drop the first name column of Employee to test the SCHEMABINDING option.

Indices are auxiliary transparent data structures applied to speed query evaluation up. The basic syntax of the CREATE INDEX statements is as follows [BOL12]:

```sql
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
ON table_name (column_names)
[INCLUDE (column_names)]
[WHERE <selection_predicate>]
```

The DROP INDEX statement allows to remove an index:

```sql
DROP INDEX index_name ON table_name
```

**Working with indices**

Perform the following steps:

1. Create the TabTest table in the companyDB database:
   ```sql
   create table TabTest(id int identity, data char(1000) default 'introDB')
   set nocount on
   insert into TabTest default values
   go
   10000
   ```

2. Enable Input/Output statistics:
   ```sql
   set statistics io on
   ```

3. Include Actual Execution Plan (Ctrl+M) and run the query (1):
   ```sql
   select * from TabTest
   ```

4. Analyse the query execution plan and note the number of data pages which SQL must read to process this query.

5. Run the query (2):
   ```sql
   select * from TabTest
   where id=3
   ```

6. Compare the execution plans of the queries (1) and (2). Compare also the number of data pages required to evaluate these queries.

7. Why the execution plans are the same?

8. Create a non-clustered index on the id column of TabTest.
   ```sql
   create index TabTest_idx1 on TabTest(id)
   ```

9. Run the queries (1) and (2). Compare and interpret their execution plans. How many data pages SQL Server needs to process the query (2).
10. Exclude Actual Execution Plan (Ctrl+M) and disable I/O statistics (set statistics io off).

11. Run the following statements:

```sql
create table TabTest2(id int identity, number int, data char(1000) default 'introDB')
set nocount on
insert into TabTest2 values(convert(int,rand()*10), 'introDB')
go 10000
select number,count(*)
from TabTest2
group by number
```

12. Create the following indices:

```sql
create index TabTest2_idx1 on TabTest2(id)
create index TabTest2_idx2 on TabTest2(number)
```

13. Include Actual Execution Plan (Ctrl+M) and enable I/O statistics.

14. Run the following query (3) and analyse its execution plan and I/O statistics.

```sql
select * from TabTest2
where id=3
```

15. Run the query (4) and analyse its execution plan and I/O statistics.

```sql
select * from TabTest2
where number=5
```

16. Why does SQL Server perform the full table scan to process the query (4)?

17. Run the query (5) and analyse its execution plan and I/O statistics.

```sql
select number from TabTest2
where number=5
```

18. Display information about indices devoted to the TabTest2 table.

```sql
select * from sys.indexes
where object_id=object_id('TabTest2','U');
```

19. Display detailed information about these indexes.

```sql
select * from sys.dm_db_index_physical_stats(db_id('companyDB'),
object_id('TabTest2','U'),null,null,'detailed')
```

20. Note the values stored in the avg_fragmentation_in_percent column.

21. Run the following update statement.

```sql
update TabTest2
set number=number*5
```

22. Run again the statement from Point 19. How the update statement above does affect the fragmentation?

23. Rebuild the TabTest2_idx2 index to reduce the fragmentation.

```sql
alter index TabTest2_idx2 on TabTest2 rebuild
```

### 9.3 Assignment 9.2

1. SQL Server automatically creates a unique index on a primary key column or columns. By
default this index is clustered. Check whether the tables in the companyDB database have clustered indexes.

2. Set the UNIQUE constraint to the name column of Department. Does it imply the creation of an index? What kind of the index is to be created?

3. Create a nonclustered index on the postal_code column of Address.

4. Create a nonclustered composite index on the emp_id and dept_id columns of Employment.

5. Create a unique nonclustered index on the name column of Country.

6. Create a nonclustered index with one key column (Employee.last_name) and two non-key columns (Employee.first_name, Employee.birth_date).

7. Remove the index created in the point 4. Create a nonclustered filtered index on (Employment.emp_id, Employment.dept_id). The filter predicate should select only the rows for which the value in end_date is null.

8. Remove the index created on the name column of Department in the point 2.

9. How to remove indices created by defining PRIMARY KEY or UNIQUE constraints?

10. There is no distinct object in the database for the view unless it is indexed using the unique clustered index [BOL12]. Indexed views called materialized are database objects which store the result of their queries. They are applied to ensure better performance for user queries. Create a unique clustered index on the column emp_id of V_Emp to materialize it.

10 Stored procedures, functions and triggers

10.1 Prerequisites

- Connect to an instance of SQL Server using SQL Server Management Studio Express 2012.
- Create a new database called CompanyDB.
- Launch a new query window for the CompanyDB database.

Procedures

A stored procedure is a sequence of T-SQL statements kept on the server. The basic syntax of the CREATE PROCEDURE statement is as follows [BOL12]:

```
CREATE { PROC | PROCEDURE } procedure_name
[ procedure_parameters ]
AS sql_statements
```

The ALTER PROCEDURE and DROP PROCEDURE statements allow to modify and remove procedures.

Example 10.1. Creating a procedure without input parameters.

```
CREATE PROCEDURE sp_emp_dept_sal
AS SELECT e.emp_id, e.first_name, e.last_name, em.salary, d.name
```
FROM Employee e JOIN Employment em ON e.emp_id=em.emp_id 
    JOIN Department d ON em.dept_id=d.dept_id 
WHERE em.end_date IS NULL 
ORDER BY em.salary DESC

Example 10.2. Invoking the procedure.
EXEC sp_emp_dept_sal

Example 10.3. Creating and invoking a procedure with input parameters
CREATE PROCEDURE GetEmpsByDeptPos 
    @dept varchar(40), @pos varchar(40) 
AS 
BEGIN 
SELECT e.emp_id, e.last_name, e.gender, em.salary 
FROM Employee e JOIN Employment em ON e.emp_id=em.emp_id 
    JOIN Position p ON em.p_id=p.p_id 
    JOIN Department d ON em.dept_id=d.dept_id 
WHERE em.end_date IS NULL AND d.name=@dept AND p.name=@pos 
END;

--Invoking the procedure 
EXEC GetEmpsByDeptPos 'IT', 'senior programmer'

Example 10.4. Creating and invoking a procedure with an input and output parameter
CREATE PROCEDURE GetNoOfEmpsFromDept 
    @dept varchar(40), @noOfEmps int OUTPUT 
AS 
BEGIN 
SELECT @noOfEmps = COUNT(*) 
FROM Employment em JOIN Department d ON em.dept_id=d.dept_id 
WHERE d.name=@dept 
END;

GO

DECLARE @noOfEmps int; 
EXEC GetNoOfEmpsFromDept 'IT', @noOfEmps OUTPUT; 
SELECT @noOfEmps 'Number of employees';

Functions
A function is a sequence of T-SQL statements stored on the database server. It can take input parameters and must return some value. SQL Server supports two types of user-defined functions: scalar and table-valued functions (see the Lecture: Section 9.3 and [BOL12]).

Example 10.5. Creating a scalar function.
CREATE FUNCTION GetNumberofEmps(@deptName varchar(30)) RETURNS money 
BEGIN 
DECLARE @avgSal money; 
SELECT @avgSal = AVG(em.salary) 
FROM Employment em JOIN Department d ON em.dept_id=d.dept_id 
WHERE d.name=@deptName AND ;
Example 10.6. Creating a multi statement table-valued function. What does the following function return?

```
CREATE FUNCTION GetEmpsNoOfTask(@lastName varchar(30))
RETURNS @EmpsTasks TABLE
(emp_id int,
 last_name varchar(40),
 task_id int,
 taskNnumber int)
AS
BEGIN
  INSERT INTO @EmpsTasks
  SELECT e.emp_id, e.last_name, t.task_id, COUNT(*) 'Number of days of delay'
  FROM Employee e JOIN Employment em ON e.emp_id=em.emp_id
    JOIN Task t ON em.emp_id=t.emp_id
  WHERE e.last_name LIKE '%'+@lastName+'%' AND em.end_date IS NULL AND t.end_date IS NOT NULL AND DATEDIFF(day, t.deadline, t.end_date)>0
  GROUP BY e.emp_id, e.last_name, t.task_id
RETURN
END
```

--Invoking the function
SELECT * FROM GetEmpsNoOfTask('S')

Triggers

Triggers are stored procedures which SQL Server runs automatically when the proper even occurs (see the Lecture: Section 9.4 and [BOL12]).

Example 10.7. Using a DML trigger to ensure that only adults people can work in the company.

```
CREATE TRIGGER TR_Adults_Emps
ON Employee
AFTER INSERT AS BEGIN
  IF EXISTS(SELECT 1
    FROM inserted AS i
    WHERE DATEDIFF(YEAR,i.birth_date,GETDATE())<18)
    BEGIN
    PRINT 'An employee must be adult person';
    ROLLBACK;
  END;
END;
GO
```

Insert the proper rows into the Employee table to test the trigger above.

Example 10.8. Using a DDL trigger to prevent modification or deletion of tables.

```
CREATE TRIGGER trgPreventDDL_AD
ON DATABASE
FOR ALTER_TABLE, DROP_TABLE
AS BEGIN
  PRINT 'Modification or deletion is not allowed';
  ROLLBACK;
END;
```
Test the trigger: Try to change or remove some tables in the *CompanyDB* database.

**Example 10.9.** Using an INSTEAD OF trigger on a table.

```sql
SELECT * INTO Emp FROM Employee

SELECT * FROM Emp

--Disable the trgPreventDDL_AD trigger
DISABLE TRIGGER trgPreventDDL_AD ON DATABASE

ALTER TABLE Emp
ADD WAS_REMOVED bit

UPDATE Emp
SET was_removed=0

--Test the trigger below and explain how it works
CREATE TRIGGER tr_Reomove_Emp
ON Emp
INSTEAD OF DELETE AS BEGIN
    UPDATE e
    SET was_removed=1
    FROM Emp e JOIN Deleted d on e.emp_id=d.emp_id;
END;
```

Test the trigger (try to delete some rows from the *Emp* table) above and explain how it works.

### 10.2 Assignment 10

1. Write a procedure which enables to hire an employee in a given department. The procedure should take the following parameters: first name, last name, birth date, gender, email, manager's identifier, department name). The procedure should also assign a new task called *init_task* to this employee. Set all dates (start date, end date, deadline) for this task to today.

2. Write a procedure which enables to fire an employee (it should take one parameter: employee's identifier). If there is no such employee it has to display some message. The procedure should also do the following:
   - Set the value of the *end_date* column of *Employment* to today.
   - Remove all incomplete tasks assigned to the employee.

3. Write a procedure which moves an employee to another department. It should takes two parameters: employee's identifier, department name, position identifier. The procedure should do the following:
   - Set the value of the *end_date* column of *Employment* to today.
   - Insert a new row into the *Employment* table (the employee has to earn 10 percent more than in the previous department).
   - Change the manager for the employee. The employee should be supervised by Adam Parker.
4. Write a function that returns departments together with the number of their current employees.

5. Write a function which displays the employment history (employee's name, department name, salary, start date, end date, position) of employees who at least once changed their department or position.

6. Write a function which returns the result of adding/subtraction/multiplying/dividing of two numbers. The function should take two numbers and the type of operation as parameters.

7. Using an instead of trigger on a view.
   - Create two tables: `Person(personId, fName, lName, birthDate, gender)` and `Phone(phoneId, number, type, pid)`. The columns `personId` and `phoneId` should be primary key columns. The `pid` column of `Phone` should be a foreign key which refers to `Person.personId`. Insert some rows to these tables.
   - Create a view named `v_PersonPhone` which stores data from all rows and all columns of `Employee` and `Phone`.
   - Try to insert a row to the view.
   - Create an instead of trigger to insert rows to the view (CREATE TRIGGER `trg_PersonPhone_Insert` ON `v_PersonPhone` INSTEAD OF INSERT AS .............). The trigger should execute two insert statements.

8. Write a DML trigger which makes it impossible to assign a new task to an employee if he/she has at least two tasks in progress. Test the trigger.

9. Create a DDL trigger which prevents creating new tables in the `companyDB` database. Test the trigger. Disable the trigger.

10. Write an instead of delete trigger on the `Employee` table. The trigger should do the following:
    - Change the manager for people supervised by the removed employee. Their manager should be the direct supervisor of the removed employee.
    - Set the `Employment.end_date` column value to today.
    - Remove all incomplete tasks assigned to the removed employee.

11. **User authorization in SQL Server**

11.1 **Assignment 11**

1. Check the authentication mode using SQL Server Management Studio Express 2012 (SSMSE12).

2. Set the authentication mode to the mixed mode.
3. Enable the sa (system administrator) login. Change the name of the sa login to sqlAdmin and set the strong password for it.

4. Logon to the SQL Server instance as sqlAdmin. Check whether this login is a member of the sysadmin server role.

5. Create a new Windows account called sql12.

6. Create a login from the sql12 account in SSMSE12.

7. Logon to the Windows system as the sql12 user.

8. Run SSMSE12 and logon to the SQL Server instance using the sql12 login.

9. Check whether the sql12 login has access to the instance databases.

10. In the companyDB database create a database user called usql12 for the sql12 login. Perform these operations in the context of sqlAdmin.

11. Check whether the usql12 user can retrieve data from the tables stored in the companyDB database.

12. Grant select, insert, update, delete permissions on the dbo schema to usql12 in the companyDB database. Check whether usql12 can perform DML statements on the tables stored in companyDB.

13. Revoke the delete permission on the dbo schema from usql12. Grant the delete permission on the Employment and Task tables to usql12. Check whether usql12 can delete rows from these tables.

14. Make the usql12 an administrator of the companyDB database (usql12 should be a member of the db_owner database role in companyDB).

15. Create a new role called r_company in the companyDB database.

16. Add the following privileges to the r_company role:
   • select and insert permissions on the dbo schema,
   • execute permission on all procedures and functions stored in dbo.

17. Remove usql12 from the db_owner role in companyDB. Add this user to the r_company role.

18. Create a new SQL Server login called introDB. Create a database user with the same name for this login in the companyDB database.

19. Ensure that the introDB user can execute all procedures and functions stored in the dbo schema in companyDB. Check whether the user can perform DML statements on the tables which are used by these procedures or functions.

20. Ensure that the introDB user can perform any DDL statements in the companyDB database.

   • Create three SQL Server logins and database users for them.
Create two procedures (sp_proc1, sp_proc2) and two functions (sp_fun1, sp_fun2).

Grant the proper privileges on these objects to database users.

Verify when the ownership chain is broken.

12 Working with MongoDB

Running the MongoDB server

It is assumed that the MongoDB archived is extracted to c:\data\db\mongodb.

To run the MongoDB server perform the following operations:

- Run Command Prompt.
- Type the command: c:\data\db\mongodb\bin\mongod.exe (it starts the core MongoDB server process).

Connecting to the MongoDB server

To connect to the server run Command Prompt and type the following command:
c:\data\db\mongodb\bin\mongo.exe

12.1 Assignment 12

1. Run the following statement (it creates a collection called products and inserts one document into it).

   db.products.insert(
   {
       _id:ObjectId(), name:"lamp", price:50.50, quantity:10, category: "A"
   })

2. Insert five documents into the products collection.

3. Select all documents in the products collection.

4. Insert the following two documents into products. Is it possible to add documents with the colour field?

   db.products.insert(
   [
       {
           name:"desk", price:340.0, quantity:10, colour:"red", category:"C"
       },
       {
           name:"table", price:900.5, quantity:2, colour:"black", category:"B"
       }
   ])

5. Select all documents in the products collection which contain the colour field.

6. Select all documents in products where the category field value is either "A" or "B". Sort the query result (in ascending order) by price.
7. Select all documents in products where the price field has the value greater than 100. The resulting documents should involve three fields: _id, name, price.

8. Select all documents in products where the quantity field value is neither 10 nor 20.

9. Select all documents in products where the name field value begins with d to l.

10. Increase the price filed value by 5% for products from the “A” category.

11. Remove the colour field from all documents where the quantity field value is less than 5.

12. Rename the field quantity to qty, and the field colour to clr.

13. Remove all documents from the products collection whose the category field value equals "C".

14. Use an aggregation pipeline to select categories and the number of products in each category whose price exceeds 10. Write the same query using the map-reduce approach and the single purpose aggregation.

15. Create a new collection called employees with the following three documents:

   ```
   { _id : ObjectId(), empId : "A37", firstName : "John", lastName : "Smith",
     deptId : "IT", hireDate : ISODate("2000-12-19T08:00:00Z"),
     salary : 3500, gender : "M", position : "programmer",
     bonuses : [{amount : 1200, date : ISODate("2003-04-14T09:00:00Z")},
                {amount : 1200, date : ISODate("2006-11-05T09:00:00Z")}] }
   
   { _id : ObjectId(), empId : "A37", firstName : "John", lastName : "Harris",
     deptId : "IT", hireDate : ISODate("2000-12-19T08:00:00Z"),
     salary : 3500, gender : "M", position : "tester",
     bonuses : [{amount : 1200, date : ISODate("2003-04-14T09:00:00Z")},
                {amount : 1200, date : ISODate("2006-11-05T09:00:00Z")}] }
   
   { _id : ObjectId(), empId : "A37", firstName : "Merry", lastName : "Lorens",
     deptId : "HR", hireDate : ISODate("2000-12-19T08:00:00Z"),
     salary : 3500, gender : "F", position : "tester",
     bonuses : [{amount : 1200, date : ISODate("2003-04-14T09:00:00Z")},
                {amount : 1200, date : ISODate("2006-11-05T09:00:00Z")}] }
   ```

16. Select department identifiers and the number of employees in each department (see the Lecture: Example 12.13).

17. Calculate the average salary for women (see the Lecture: Example 12.14).

18. Run the query from Example 12.16 and analyse its result.

19. Run the query from Example 12.18 and analyse its result.

20. Count all documents in the employees collection.

### 13 Bibliography

| Lecture | M. Bleja: Introduction to databases (Lecture) |
CREATE TABLE Country
(
  country_id int IDENTITY(1,1) CONSTRAINT PK_Country_C_id PRIMARY KEY,
  name varchar(50) NOT NULL);

CREATE TABLE Address
(
  address_id int IDENTITY(1,1) CONSTRAINT PK_Address_A_id PRIMARY KEY,
  street varchar(40) NOT NULL,
  city varchar(50) NOT NULL,
  postal_code varchar(10) NOT NULL,
  country_id int NOT NULL,
  CONSTRAINT FK_Address_C_id FOREIGN KEY(country_id) REFERENCES Country(country_id));

CREATE TABLE Employee
(
  emp_id int IDENTITY(1,1) CONSTRAINT PK_Employee_Emp_id PRIMARY KEY,
  first_name varchar(25) CONSTRAINT NN_Employee_fn NOT NULL,
  last_name varchar(30) NOT NULL,
  birth_date date NOT NULL,
  gender char(1) CONSTRAINT CH_Employee_gen CHECK(gender='M' OR gender='F'),
  email varchar(40) CONSTRAINT U_Employee_Email UNIQUE,
  manager_id int,
  address_id int NOT NULL,
  CONSTRAINT FK_Employee_Mgr_id FOREIGN KEY(manager_id) REFERENCES Employee(emp_id),
  CONSTRAINT FK_Employee_Address FOREIGN KEY(address_id) REFERENCES Address(address_id));

CREATE TABLE Position
(
  p_id int IDENTITY(1,1) CONSTRAINT PK_Position_P_id PRIMARY KEY,
  name varchar(30) NOT NULL);

CREATE TABLE Department
(
  dept_id int IDENTITY(1,1),
  name varchar(50) NULL,
  CONSTRAINT PK_Department_Dept_id PRIMARY KEY(dept_id),
  address_id int NOT NULL,
  CONSTRAINT FK_Department_Address FOREIGN KEY(address_id) REFERENCES Address(address_id));

CREATE TABLE Employment
(
  e_id int IDENTITY(1,1),
  emp_id int NOT NULL,
  dept_id int NOT NULL,
  p_id int NOT NULL,
  salary money NOT NULL,
  start_date date NOT NULL CONSTRAINT D_EMPLOYMENT_SD DEFAULT GETDATE(),
  end_date date NULL,
  CONSTRAINT PK_Employment_E_id PRIMARY KEY(e_id),
  CONSTRAINT FK_Employment_Department_Dept_id FOREIGN KEY(dept_id)
    REFERENCES Department(dept_id),
  CONSTRAINT FK_Employment_Employee_Emp_id FOREIGN KEY(emp_id) REFERENCES Employee(emp_id),
  CONSTRAINT FK_Employment_Position_P_id FOREIGN KEY(p_id) REFERENCES Position(p_id));
CREATE TABLE Task
(
    task_id int IDENTITY(1,1) NOT NULL,
    name varchar(40) NOT NULL,
    e_id int NOT NULL,
    start_date datetime NOT NULL,
    end_date datetime NOT NULL,
    deadline datetime NOT NULL,
    [desc] varchar(max) NULL,
    CONSTRAINT FK_Task_Employment FOREIGN KEY(e_id) REFERENCES Employment(e_id));

set dateformat dmy

insert into Country values('Poland')
insert into Country values('United Kingdom')
insert into Country values('Germany')

insert into Address values('22 Banacha', 'Lodz', '90-238', 1)
insert into Address values('11 Nowa', 'Krakow', '60-438', 1)
insert into Address values('14 Zieona', 'Warsaw', '00-983', 1)
insert into Address values('23 Granitowa', 'Lodz', '90-922', 1)
insert into Address values('145 Piotrkowska', 'Lodz', '90-289', 1)
insert into Address values('90 Targowa', 'Warsaw', '00-987', 1)

insert into Address values('56 Kwaitowa', 'Lodz', '90-454', 1)
insert into Address values('54 Bednarska', 'Krakow', '60-238', 1)
insert into Address values('76 Mickiewicza', 'Warsaw', '00-424', 1)
insert into Address values('53 Rojna', 'Lodz', '90-253', 1)
insert into Address values('89 Sarnia', 'Lodz', '90-532', 1)
insert into Address values('33 Zachodnia', 'Warsaw', '00-231', 1)

insert into Address values('28 Polna', 'Lodz', '90-532', 1)
insert into Address values('23 Wschodnia', 'Krakow', '60-124', 1)
insert into Address values('16 Senatorska', 'Warsaw', '00-342', 1)
insert into Address values('63 Akacjowa', 'Lodz', '90-922', 1)
insert into Address values('22 Bema', 'Lodz', '90-482', 1)
insert into Address values('10 Jaracza', 'Warsaw', '00-225', 1)

insert into Address values('45 Narutowicza', 'Lodz', '90-238', 1)
insert into Address values('14 Kresowa', 'Krakow', '60-438', 1)
insert into Address values('53 Banacha', 'Warsaw', '00-233', 1)

insert into Employee values('John', 'Smith', '08.08.1980', 'M', 'JSMITH@COMPEDU.EU',null,1)
insert into Employee values('Adam', 'Parker', '24.05.1985', 'M', 'APARKER@COMPEDU.EU',1,2)
insert into Employee values('Filip', 'Miller', '12.04.1984', 'M', 'FMILLER@COMPEDU.EU',1,3)
insert into Employee values('Henry', 'Bell', '23.05.1976', 'M', 'HBELL@COMPEDU.EU',2,4)
insert into Employee values('James', 'Collins', '14.09.1986', 'M', 'JCOLLINS@COMPEDU.EU',2,5)
insert into Employee values('Adrien', 'PoweI', '11.06.1988', 'M', 'APOWELL@COMPEDU.EU',3,6)

insert into Employee values('Albert', 'Foster', '08.04.1980', 'M', 'AFOSTER@COMPEDU.EU',1,7)
insert into Employee values('Alice', 'Adams', '25.07.1984', 'F', 'AADAMS@COMPEDU.EU',2,8)
insert into Employee values('Eva', 'Owen', '16.02.1979', 'F', 'EOWEI@COMPEDU.EU',3,9)
insert into Employee values('Hannah', 'Mills', '30.04.1987', 'F', 'HMILLS@COMPEDU.EU',3,10)
insert into Employee values('Justina', 'Stevens', '23.09.1980', 'F', 'JSTEVENS@COMPEDU.EU',2,11)
insert into Employee values('Laura', 'Fisher', '11.03.1982', 'F', 'LFISHER@COMPEDU.EU',4,12)

insert into Employee values('Monica', 'Shaw', '23.04.1980', 'F', 'MSHAW@COMPEDU.EU',5,13)
insert into Employee values('Richard', 'Cook', '11.06.1899', 'M', 'RCOOK@COMPEDU.EU',4,14)
insert into Employee values('Thomas', 'Smith', '14.12.1980', 'M', 'TSMITH@COMPEDU.EU',2,15)
insert into Employee values('Victor', 'Palmer', '18.11.1980', 'M', 'VPALMER@COMPEDU.EU',1,16)
insert into Employee values('Simon', 'Mills', '28.09.1983', 'M', 'SMILLS@COMPEDU.EU',6,17)
insert into Employee values('John', 'Tyler', '28.05.1977', 'M', 'JTYLOR@COMPEDU.EU',3,18)
insert into Position values('director')
insert into Position values('vice-director')
insert into Position values('junior programmer')
insert into Position values('senior programmer')
insert into Position values('analyst')
insert into Position values('senior tester')
insert into Position values('junior tester')
insert into Position values('IT manager')
insert into Position values('sales representative')
insert into Position values('sales manager')
insert into Position values('accountant')

insert into Department values('IT',19)
insert into Department values('HR',20)
insert into Department values('Marketing',21)

insert into Employment values(1,1,1,19000, '05.10.2010',null)
insert into Employment values(2,1,2,12000, '12.11.2011',null)
insert into Employment values(3,1,3,11000, '12.11.2010',null)
insert into Employment values(4,1,2,10000, '14.12.2010',null)
insert into Employment values(5,2,3,9500, '05.12.2010',null)
insert into Employment values(6,2,2,9000, '14.04.2011',null)

insert into Employment values(7,1,3,9000, '12.06.2011',null)
insert into Employment values(8,1,2,8000, '15.05.2011', '15.05.2012')
insert into Employment values(9,2,3,7000, '18.09.2011',null)
insert into Employment values(10,2,4,7000, '05.11.2011',null)
insert into Employment values(11,3,4,6000, '29.01.2010',null)
insert into Employment values(12,3,3,5000, '15.10.2011', '15.10.2013')

insert into Employment values(13,3,3,5000, '15.10.2011',null)
insert into Employment values(14,3,2,6000, '02.06.2011',null)
insert into Employment values(15,3,1,7000, '01.10.2010',null)
insert into Employment values(16,1,4,4000, '08.12.2011',null)
insert into Employment values(16,1,4,6000, '08.09.2011',null)
insert into Employment values(16,2,3,5000, '10.06.2011',null)

insert into Employment values(8,2,2,9000, '16.05.2012',null)
insert into Employment values(12,2,2,6000, '16.10.2013',null)

insert into Task values('Task1',1, '06.10.2010', '05.11.2010', '05.11.2010',null)
insert into Task values('Task6',6, '15.04.2011', '14.05.2011', '14.05.2011',null)

insert into Task values('Task8',8, '16.05.2011', '15.06.2011', '15.06.2011',null)
insert into Task values('Task10',10, '06.11.2011', '05.12.2011', '05.12.2011',null)
insert into Task values('Task11',11, '30.01.2010', '27.02.2010', '25.02.2010',null)
insert into Task values('Task12',12, '16.10.2011', '15.11.2011', '15.11.2011',null)

insert into Task values('Task14',14, '03.06.2011', '02.07.2011', '02.07.2011',null)
insert into Task values('Task15',15, '02.10.2010', '10.11.2010', '08.11.2010',null)
insert into Task values('Task17',17, '09.09.2011', '08.10.2011', '08.10.2011',null)