

Databases II, Hogent

JeeVeeVee

2020/2021

Contents

1	SQL : Review	2
1.1	overview	2
1.2	SELECT	2
1.2.1	DML - consulting data	2
1.2.2	basic form of SELECT statement	2
1.3	use of functions	3
1.4	the case function	3
1.5	basic concepts revisited	4
1.5.1	GROUP BY and statistical functions	4
1.5.2	Working with more then 1 table : JOIN	4
1.6	set operators : UNION - INTERSECT - EXCEPT	4
2	SQL : advanced	5
2.1	subqueries	5
2.1.1	correlated subqueries	5
2.1.2	subqueries and the EXISTS operator	5
2.2	DML basic tasks	5
2.2.1	Adding data - INSERT	6
2.2.2	modifying data - UPDATE	6
2.2.3	remove rows - DELETE	6
2.3	views	6
2.3.1	definition of a view	7
2.4	common table expressions	7
2.4.1	the WITH component	7
2.4.2	recursive CTE's	7
3	SQL : Data Definition Language	9
3.1	DDL - Database	9
3.2	DDL - Tables	9
3.2.1	creating tables	9
3.2.2	changing tables	10
3.2.3	removing tables	10
3.3	Scripts	10
3.4	SQL Datatypes	10
3.5	constraints	11
3.5.1	identity values	11
4	window functions	12
4.1	moving aggregate	12
4.2	LAG and LEAD	13
5	DB programming	14
5.1	SQL as complete language	14
5.1.1	Persistent Stored Modules	14
5.2	TRANSACT SQL	14
5.2.1	local variables	14
5.2.2	Error handling in TRANACT SQL	14
5.3	Stored Procedures	14
5.4	cursors	15

Chapter 1

SQL : Review

1.1 overview

There are a lot of dialects and variants of SQL, this makes it very hard to establish some kind of general standard. Universal, there are 3 subvariants of SQL statements that we recognize:

- **Data Definition Language (DDL)**
- **Data Manipulation language (DML)**
- **Data Control Language (DCL)**

1.2 SELECT

1.2.1 DML - consulting data

- consulting 1 table
 - basic form
 - SELECT clause
 - WHERE clause
 - row formatting
 - statistical functions
 - grouping
- consulting >1 table

1.2.2 basic form of SELECT statement

statement for consulting 1 table:

```
1 SELECT [ALL | DISTINCT] {*|expression[, expression...]} FROM tablename[WHERE
    conditions(s)][GROUP BY column name [, column name ...][HAVING conditions(s)][
    ORDER BY {column name |seqnr}{ASC|DESC}[,...]]
```

- **SELECT** : specifies the columns to show in the output
- **DISTINCT** : filters out duplicates
- **FROM** : specifies table name
- **WHERE** : filter condition on individual lines in the output
- **ORDER BY** : sortin
- **GROUP BY** : groupin
- **HAVING** : filter condition for groups

you can name the output columns by using an AS, example :

```
1 SELECT productid AS ProductNumber, productname AS 'Name Product' FROM product
```

1.3 use of functions

- String
 - left
 - right
 - len
 - rtrim
 - substring
 - replace
- DateTime
 - DateAdd
 - DateDiff
 - DatePart
 - Day
 - Month
 - Year
 - **GETDATE()** returns the current DateTime
- Arithmetic
 - round
 - floor
 - ceiling
 - cos
 - sin

you can cast variables to a certain type by using
CAST(<value expression>AS <data type>)

1.4 the case function

example :

```
1  select case region
2      when 'OR' then 'West'
3      when 'MI' then 'North'
4      else 'Elsewhere'
5  end,
6      city,
7      region
8  from supplier;
```

1.5 basic concepts revisited

1.5.1 GROUP BY and statistical functions

- SUM
- AVG
- MIN
- MAX
- COUNT

Grouping with GROUP BY

filter on groups using HAVING example :

```
1 SELECT
2   ProductTypeID,
3   count(productid)
4 FROM Product
5 GROUP BY ProductTypeID
6 HAVING COUNT(PRODUCTID) > 10
```

1.5.2 Working with more then 1 table : JOIN

JOIN If you want to select from more then 1 table, you can use the join keyword, there are 3 variants:

- INNER JOIN

joins a row from one table with another one based on common criteria in the corresponding tables example:

```
1 SELECT TeamNo, Name
2 FROM Teams JOIN Players
3 ON Teams.playerno = Player.playerno
```

You can also join more then 2 tables, and also join a table with itself

- OUTER JOIN

returns all records from 1 table, even if there is no corresponding record in the other table
there are 3 types :

- LEFT OUTER JOIN : returns all rows of the first table
- RIGHT OUTER JOIN : return all rows of the second table
- FULL OUTER JOIN : returns all of the first and the second table

- CROSS JOIN

in a cross join the number of rows is equal to the number of rows in the first table multiplied by the number of rows in the second table

this is used to generate all possible combinations

1.6 set operators : UNION - INTERSECT - EXCEPT

with **UNION** you can combine the result of 2 queries, but the 2 results need to have the exact same SELECT's example :

```
1 SELECT lastname + ' ' + firstname AS name, city, postalcode)
2 FROM Employee
3 UNION
4 SELECT customername, city, postalcode
5 FROM Customer
```

INTERSECT returns all the records that are in both of the query-results **EXCEPT** returns all records that are only in the first query-result

Chapter 2

SQL : advanced

2.1 subqueries

A subquery can return a single value, or a single column, or more columns. ANY and ALL are 2 keywords for comparing a subquery which returns a column.

2.1.1 correlated subqueries

In a correlated subquery, the inner query depends on info from the outer query. In this case, the subquery is executed for each row in the main query. This makes this method not very efficient. If possible, use joins or simple subqueries. Principle:

```
1 SELECT *
2 FROM table a
3 WHERE expression operator (SELECT *
4                             FROM table
5                             WHERE expression operator a.columnname)
```

2.1.2 subqueries and the EXISTS operator

The EXISTS operator tests the existence of a result set, you can also use NOT EXISTS. example that returns all the players that haven't played a game yet:

```
1 SELECT *
2 FROM players as p
3 WHERE NOT EXISTS(
4     SELECT * FROM matches WHERE playerno = p.playerno)
```

Subqueries can also be used in the SELECT, and FROM clauses

2.2 DML basic tasks

- INSERT to add data
- UPDATE to change data
- DELETE to remove data
- MERGE combination of the previous 3

tip for not destroying a database when you are working with DML, and SQL has by default no UNDO, this is why we work with transactions:

```
1 begin transaction -- start a new transaction
2 --> saves previous state of the DB in buffer
3
4 --several "destructive" commands can go here
5 DELETE FROM Employee;
6 INSERT INTO product
7     values (10001 'Drinking bottle', null, null, null, null, null, null);
8
9 -- you can see the changes in this session
```

```

10 SELECT * FROM Product WHERE ProductID = 10001;
11
12 rollback; --> ends transaction and restores database to state before begin
    transaction
13 -- commit; --> if you want to make the changes permanent

```

2.2.1 Adding data - INSERT

The insert statement adds data in a table. You can do this by only specifying the values that are NOTNULL :

```

1 INSERT INTO product (productID, productName)
2 VALUES (10000, 'Energy bar')

```

Or if you are a sick fuck :

```

1 INSERT INTO product
2 VALUES (10000, 'Energy bar', null, null, null, null, null, null)

```

2.2.2 modifying data - UPDATE

To change all rows in a table:

```

1 UPDATE product
2 SET price = (price * 1.1)

```

To change 1 row, or a group of rows:

```

1 UPDATE product
2 SET price = (price * 1.1)
3 WHERE productName = 'Wheeler'

```

To change more than 1 value:

```

1 UPDATE product
2 SET price = (price * 1.1), unitsInStock = 0

```

2.2.3 remove rows - DELETE

deleting rows :

```

1 DELETE
2 FROM product
3 WHERE productName = 'Wheeler';

```

deleting all rows in a table:

```

1 DELETE
2 FROM product
3
4 --> or use truncate
5
6 TRUNCATE TABLE product

```

You can also delete rows based on data in another table:

```

1 DELETE FROM ordersDetail
2 WHERE orderid in
3     (SELECT orderID
4      FROM orders
5      WHERE orderdate = (SELECT MAX(orderdate) FROM Orders)
6      );

```

2.3 views

Definition : A view is a SELECT statement, it can be seen as a virtual table composed out of other tables & views. The advantages of views are that they hide complexity of the database, large and complex queries become accessible and reusable. They are also handy for export to other applications.

2.3.1 definition of a view

syntaxis:

```
1 CREATE VIEW view_name [(column_list)]
2 AS select_statement
3 [with check option]
```

If you use a lot of views, this may become some kind of a mess, since they are all stored within the database. Views are also updateable, use the keyword **ALTER**. A view can also update a table, instead of a SELECT statement, a INSERT, UPDATE or DELETE clause is used then. The **CHECK** option is used to check if an update makes it so that a certain row is no longer part of the view. If the check option is enabled, there will be an error generated.

2.4 common table expressions

2.4.1 the WITH component

The **WITH** component has 2 application areas:

1. Simplify SQL-instructions, avoid repetition of SQL constructs
2. Traverse recursively hierarchical and network structs

Using the **WITH** component, you can give the subquery its own name (with column names) and reuse it in the rest of the query (as much as needed).

```
1 WITH fines(number)
2 AS (SELECT count(pe.playerno)
3      FROM players AS p1
4      LEFT JOIN penalties AS pe
5      GROUP BY p1.playerno
6      )
7
8 SELECT AVG(number * 1.0)
9 FROM fines;
```

We use CTE to abbreviate Common Table Expression.

CTE's look a lot like views, the difference is that a CTE only exists during the SELECT-statement and that the CTE is not visible for other users and applications. They also look a bit like Subqueries, since they are both virtual tables, difference here is that a CTE can be reused, a CTE is defined on top of the query, instead of within the clause where it is used. A simple subquery can always be replaced by a CTE.

2.4.2 recursive CTE's

recursive means that we continue to execute a CTE until a condition is reached. This allows us to solve problems like :

- Who are the friends of my friends?
- What is the hierarchy of an organisation
- find the parts and subparts of a product

example: next CTE gives you all numbers from 1 to 5

```
1 with numbers(number) as
2 (SELECT 1
3  UNION all
4  SELECT number + 1
5  FROM numbers
6  WHERE number < 5)
```

How does this work :

1. SQL searches table expressions that do not contain recursivity and executes them one by one
2. execute all recursive expressions. The numbers table, that got a value of 1 in step 1, is used . A new row is added to the numbers table (2)

3. the second expression is re-executed, giving (3) as a result?
4. since step 3 still gave us a result, the recursive expression is used again, giving (4) as a result.
5. again.(5)
6. if the expression now is processed again, it does not return a result since the previous step no rows were added. SQL stops the processing of the table and the final result is known.

The max number of recursions is 100, but if you use the option maxrecursion N, you can change this to N.

Chapter 3

SQL : Data Definition Language

DDL can be used for :

- defining databases
- defining tables
- determining data types in SQL server
- defining constraints - data integrity
- defining indexes
- defining views (see previous chapter)

3.1 DDL - Database

The data from a database is stored within data files, these often have a .mdf or .ndf extension. A database also stores log files, these have a .ldf extension. A database is created by a sysadmin, who has the correct permissions. It is done by making a copy of a "model" database. The command for creating a new database is:

```
1  -- the simple version
2  CREATE DATABASE database_name
3
4  -- the version for wizards
5  CREATE DATABASE database_name
6  [ON [<filespec> [ ,...n ]] [<filegroup>[ ,...n ]]]
7  [LOG ON { < filespec> [ ,...n ] } ] [COLLATE collation_name]
8  [FOR LOAD | FOR ATTACH]
9  < filespec > :: =
10 [PRIMARY](
11 [NAME =logical_file_name,]
12  FILENAME = 'os_file_name'
13  [,SIZE =size]
14  [,MAXSIZE={ max_size| UNLIMITED } ]
15  [,FILEGROWTH =growth_increment]) [ ,...n]
16 < filegroup > :: = FILEGROUP filegroup_name < filespec > [ ,...n]
```

Deleting a database is a lot easier:

```
1 DROP DATABASE database_name
```

By using ALTER, you can change the characteristics of database

3.2 DDL - Tables

3.2.1 creating tables

When creating a new table, you have to specify, the name of the table, the definition of its columns and the definition of constraints.

```

1 CREATE TABLE table_name (
2   {<column_definition> |
3     <computed_column_definition> |
4     <column_set_definition>}
5   [table_constraint] [, ...n] )

```

3.2.2 changing tables

Adding, changing or removing columns. Using the ALTER keyword followed by MODIFY, ADD, DROP.

```

1 -- adding the address column
2 ALTER TABLE student
3   ADD address varchar(40) NULL
4
5 --changing the address column
6 ALTER TABLE student
7   MODIFY COLUMN address varchar(50) NULL
8
9 --removing the address column
10 ALTER TABLE student
11   REMOVE COLUMN address

```

3.2.3 removing tables

Use the DROP keyword

3.3 Scripts

Scripts are used for batch processing and creating a test and production environment.

3.4 SQL Datatypes

There are a few categories:

1. exact numerics
 - bigint
 - int
 - smallint
 - tinyint
 - bit
 - decimal/numeric
2. approximate numerics
 - float
 - real
3. Date and time
 - datetime
 - smalldatetime
 - date
 - time
4. Character strings
 - char[(n)]
 - varchar[(n — max)]
 - nchar[(n)]
 - nvarchar[(n)]

5. Unicode character strings

6. Binary strings

binary[(n)]

varbinary[(n — max)]

7. other

type conversion

There is implicit (automatic) and explicit type conversion. for explicit, use CAST and CONVERT (chapter 1)

3.5 constraints

3.5.1 identity values

An identity column contains a unique value for each row, system generated sequential values. There is only 1 identity column allowed per table, this column always uses integer datatypes, the value of an identity column can't be NULL. This column also is not updateable. Identity columns ensure data integrity in 3 ways: domain (each value only occurs once), entity (the value is unique) and referential (you can safely use this column to refer to this table). Example:

```
1 create table student(  
2     studentno int identity(1, 1) not null primary key,  
3     lastname varchar(30) not null,  
4     firstname varchar(30) not null,  
5     gender char(1) default 'M' check(gender in ('M', 'F')) not null,  
6     ssno int not null,  
7     class smallint null,  
8     photograph varbinary(max) null,  
9     constraint ssno_u unique(ssno),  
10    constraint class_fk foreignkey(class)  
11        references class(classID)  
12 )
```

The **CHECK** constraint: checked with INSERT and UPDATE

The **UNIQUE** constraint: specifies that 2 rows can have the same value for a certain column

The **PRIMARY KEY** constraint: only 1 of these per table, can be defined as 1 or a combo of columns (= composed key), the value has to be unique and NOTNULL

The **FOREIGN KEY** constraint: used to link 2 tables, NULL values are allowed, this constraint guarantees referential integrity. There are a few extra options;

- ON DELETE

- CASCADE : cascaded delete
- NO ACTION : delete only if no referring values, otherwise : error, this is the default
- SET NULL : referring values are set to NULL (only possible if no NOTNULL constraint on FK columns)
- SET DEFAULT : referring values are set to the default value.

- ON UPDATE

- CASCADE : cascaded update
- NO ACTION : update only if no referring valeus, otherwise: error, this is the default
- SET NULL : referring values are set to NULL
- SET DEFAULT : referring values are set to their defaults.

Chapter 4

window functions

Example: you want to compare the sales numbers from last year to those of this year, window functions offer a solution to these kind of problems in a single, efficient SQL query. They use the OVER clause to do so. The results of a SELECT are partitioned, there is numbering, ordering and aggregate functions per partition. The partition behaves as a window that shifts over the data, hence the name window function.

Instead of solving the problem with a inefficient subquery:

```
1 SELECT orderid, orderdate, orderamount,
2     (SELECT sum(orderamount)
3      FROM orders
4      WHERE year(orderdate) = year(o.orderdate)
5            and orderid <= o.orderid) YTD
6 FROM orders o
7 ORDER BY orderid
```

We wil now use a window function to simply and improve the query:

```
1 SELECT orderid, orderdate, orderamount,
2     sum(orderamount) OVER
3     (partition by year(o.orderdate) order by o.orderid) YTD
4 FROM orders o
5 ORDER BY orderid
```

The partition is optional, the order by is mandatory.

The function **row_number()** gives each row a running sequence number, no duplicates occur in the same partition.

rank() gives each row a rank withing the partition, duplicates can occur : 1, 2, 3, 3, 5.

dense_rank() makes it so that there are no gaps withing the ranking → 1, 2, 3, 3, 4.

4.1 moving aggregate

The real meaning of window functions is to have a window that shifts over the result set, previous examples work with default window : start of resultset to current row. The solution of the previous section could also have been:

```
1 SELECT orderid, orderdate, orderamount,
2     sum(orderamount) OVER
3     (partition by year(o.orderdate) order by o.orderid
4      range between unbounded preceding and current row) YTD
5 FROM orders o
6 ORDER BY orderid
```

with range, you have 3 valid options:

```
1 range between unbounded preceding and current row
2 range between current row and unbounded following
3 range between unbounded preceding and unbounded following
```

When you use range, the current row is compared to other rows and grouped based on the ORDER BY predicate. This is not always desirable, you might actually want a physical offset. In this case you would specify ROWS instead of RANGE, this also gives you 3 options:

```
1 rows between N preceding and current row
2 rows between current row and N following
3 rows between N preceding and M following
```

4.2 LAG and LEAD

Window functions LAG and LEAD refer to previous and next line respectively

Chapter 5

DB programming

5.1 SQL as complete language

5.1.1 Persistent Stored Modules

Originally SQL was not a complete programming language, by adding variables, constants, datatypes, operators, control structures (if, case, while, for), it became one. The dialect is very important here, the code you will find here can only be used for Microsoft Transact SQL. We can now make use of user defined functions, since some of the built-in functions don't meet our requirements.

5.2 TRANSACT SQL

5.2.1 local variables

```
1 -- variable names always start with @
2 DECLARE @variable_name1 data_type [, @variable_name2 datatype ...]
3
4 --assign value to a variable
5 SET @variable_name = expression
6 SELECT @variable_name = column_spec
7
8 --set and select are equivalent, but only set is ANSI
```

5.2.2 Error handling in TRANSACT SQL

- **RETURN** : immediate end of execution of the batch procedure
- **@@error** : contains error number of last executed SQL instruction, if value = 0, OK
- **RAISERROR** : returns user defined error or system error
- Use of TRY ... CATCH block

5.3 Stored Procedures

example: delete a customer

```
1 create procedure DeleteCustomer @custno int = NULL
2 as
3 if @custno IS NULL
4     begin print 'Please provide a customerid'
5         return
6     end
7 if not exists (select null from customer where customerid = @custno)
8 begin
9     print 'The customer doesn't exist.'
10    return
11 end
12 if exists (select null from orders where customerid = @custno)
13 begin
14     print 'The customer already has orders and can't be removed'
15     return
```

```

16 end
17 delete from customer where customerid = @custno
18 print 'The customer has been succesfully deleted.'

```

A stored procedure is a collection of SQL and control-of-flow commands (program) that is stored as a database object.

5.4 cursors

SQL statements are processing complete resultsets and not individual rows. Cursors allow to process individual rows to preform complex, row specifk operations that can't (easily be preformed with a single SELECT, UPDATE or DELETE statement. It is a datbase object that refers to the result of a query. It allows to specify the row from the resultset you wish to process. There are 5 important cursor related statements:

1. **DECLARE CURSOR** : creates and defines the cursor

```

1 DECLARE <cursor_name> [INSENSITIVE][SCROLL] CURSOR FOR
2 <SELECT_statement>
3 [FOR {READ ONLY | UPDATE[OF <column list>]}]

```

- **INSENSITIVE** : the cursor uses a temporary copy of the data, the cursor can't be used to change data
- **SCROLL** : all fetch operations (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE) are allowed, if scroll is omitted, only NEXT is available
- **READ ONLY** : prohibits data changes
- **UPDATE** : data changes are allowed

2. **OPEN** : opens the declared cursor

```

1 OPEN <cursor_name>

```

3. **FETCH** : fetches 1 row
4. **CLOSE** : closes the cursor (counterpart to OPEN)
5. **DEALLOCATE** : remove the cursor definition (counterpart to DECLARE)