Databases II, Hogent

JeeVeeVee

2020/2021

Contents

1 SQL : Review **3**
- 1.1 overview . . . 3
- 1.2 SELECT . . . 3
  - 1.2.1 DML - consulting data . . . 3
  - 1.2.2 basic form of SELECT statement . . . 4
- 1.3 use of functions . . . 4
- 1.4 the case function . . . 5
- 1.5 basic concepts revisited . . . 6
  - 1.5.1 GROUP BY and statistical functions . . . 6
  - 1.5.2 Working with more then 1 table : JOIN . . . 6
- 1.6 set operators : UNION - INTERSECT - EXCEPT . . . 7

2 SQL : advanced **8**
- 2.1 subqueries . . . 8
  - 2.1.1 correlated subqueries . . . 8
  - 2.1.2 subqueries and the EXISTS operator . . . 8
- 2.2 DML basic tasks . . . 9
  - 2.2.1 Adding data - INSERT . . . 9
  - 2.2.2 modifing data - UPDATE . . . 10
  - 2.2.3 remove rows - DELETE . . . 10
- 2.3 views . . . 10
  - 2.3.1 definition of a view . . . 11
- 2.4 common table expressions . . . 11
  - 2.4.1 the WITH component . . . 11
  - 2.4.2 recursive CTE's . . . 12

3 SQL : Data Definition Language **13**
- 3.1 DDL - Database . . . 14
- 3.2 DDL - Tables . . . 14
  - 3.2.1 creating tables . . . 14
  - 3.2.2 changing tables . . . 15
  - 3.2.3 removing tables . . . 15
- 3.3 Scripts . . . 15
- 3.4 SQL Datatypes . . . 15

3.5 constraints 16
3.5.1 identity values 16

4 window functions **18**
4.1 moving aggregate 19
4.2 LAG and LEAD 19

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:

```
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 indiviudual 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 :

```
SELECTproductid AS ProductNummer, 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**
- Arithmietic
  - 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 :

```
select case region
  when 'OR' then 'West'
  when 'MI' then 'North'
  else 'Elsewhere'
  end,
  city,
  region
from supplier;
```

1.5 basic concepts revisited

1.5.1 GROUP BY and statistical functions

- SUM
- AVG
- MIN
- MAX
- COUNT
- TOP

```
    -- Select the top 5 of the cheapest products (important)
    SELECT TOP 5 productid, price
    FROM product
    ORDER BY price;

    -- 5 most expensive products: ORDER BY price DESC
    SELECT TOP 5 productid, price
    FROM product
    ORDER BY price DESC;

```

Grouping with GROUP BY
filter on groups using HAVING example :

```
SELECT
  ProductTypeID,
  count(productid)
FROM Product
GROUP BY ProductTypeID
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:

```
SELECT TeamNo, Name
FROM Teams JOIN Players
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 in 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:

```
SELECT *
FROM table a
WHERE expression operator (SELECT *
                FROM table
                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 youse NOT EXISTS. example that retursn all the players that haven't played a game yet:

```
SELECT *
FROM players as p
WHERE NOT EXISTS(
  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 date
- 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 tranactions:

```
begin transaction -- start a new transaction
--> saves previous state of the DB in buffer

--several "destructive" commands can go here
DELETE FROM Employee;
INSERT INTO product
  values (10001 'Drinking bottle', null, null, null, null, null, null);

-- you can see the changes in this session
SELECT * FROM Product WHERE ProductID = 10001;

rollback;  --> ends transaction and restores database to state before
    begin transaction
-- commit; --> if you want to make the changes permanent
```

2.2.1 Adding data - INSERT

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

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

Or if you are a sick fuck :

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

2.2.2 modifing data - UPDATE

To change all rows in a table:

```
UPDATE product
SET price = (price "1.1)
```

To change 1 row, or a group of rows:

```
UPDATE product
SET price = (price "1.1)
WHERE productName = 'Wheeler'
```

To change more then 1 value:

```
UPDATE product
SET price = (price "1.1), unitsInStock = 0
```

2.2.3 remove rows - DELETE

deleting rows :

```
DELETE
FROM product
WHERE productName = 'Wheeler';
```

deleting all rows in a table:

```
DELETE
FROM product

--> or use truncate

TRUNCATE TABLE product
```

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

```
DELETE FROM ordersDetail
WHERE orderid in
  (SELECT orderID
   FROM orders
   WHERE orderdate = (SELECT MAX(orderdate) FROM Orders)
   );
```

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 accesible and reusable. They are also handy for export to other applications.

2.3.1 definition of a view

syntaxis:

```
CREATE VIEW view_name [(column_list)]
AS select_statement
[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 repitition of SQL constructs
2. Traverse recursively hierachrical 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).

```
WITH fines(number)
  AS (SELECT count(pe.playerno)
      FROM players AS p1
        LEFT JOIN penalties AS pe
      GROUP BY p1.playerno
    )

SELECT AVG(number * 1.0)
FROM fines;
```

We use CTE to abreviate 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

```
with numbers(number) as
  (SELECT 1
   UNION all
    SELECT number + 1
    FROM numbers
    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 or 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
- defning 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 extenstion. 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:

```
-- the simple version
CREATE DATABASE database_name

-- the version for wizards
CREATE DATABASE database_name
[ON [<filespec> [ ,...n ]] [,<filegroup>[ ,...n ]]]
[LOG ON { < filespec> [ ,...n ] } ] [COLLATE collation_name]
[FOR LOAD | FOR ATTACH]
< filespec > :: =
[PRIMARY](
[NAME =logical_file_name,]
  FILENAME = 'os_file_name'
  [,SIZE =size]
  [,MAXSIZE ={ max_size| UNLIMITED } ]
  [,FILEGROWTH =growth_increment]) [ ,...n]
< filegroup > :: = FILEGROUP filegroup_name < filespec > [ ,...n]
```

Deleting a datbase is a lot easier:

```
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.

```
CREATE TABLE table_name (
  {<column_defination> |
   <computed_column_definition> |
   <column_set_definition}
  [table_constraint] [, ...n] )
```

3.2.2 changing tables

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

```
-- adding the address column
ALTER TABLE student
  ADD address varchar(40) NULL

--changing the address column
ALTER TABLE student
  MODIFY COLUMN address varchar(50) NULL

--removing the address column
ALTER TABLE student
  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. Charachter strings
   - char[(n)]
   - varchar([n | max])
   - nchar[(n)]
   - nvarchar[(n)]
5. Unicode charachter 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 occors once), entity (the value is unique) and referential (you can safely use this column to refer to this table). Example:

```
create table  student(
  studentno int identity(1, 1) not null primary key,
  lastname varchar(30) not null,
  firstname varchar(30) not null,
  gender char(1) default 'M' check(gender in ('M', 'F')) not null,
  ssno int not null,
  class smallint null,
  photograph varbinary(max) null,
  constraint ssno_u unique(ssno),
  constraint class_fk foreignkey(class)
    references class(classID)
)
```

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
  - CASCASE : 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 numberng, 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:

```
SELECT orderid, orderdate, orderamount,
  (SELECT sum(orderamount)
   FROM orders
   WHERE year(orderdate) = year(o.orderdate)
         and orderid <= o.orderid) YTD
FROM orders o
ORDER BY orderid
```

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

```
SELECT orderid, orderdate, orderamount,
  sum(orderamount) OVER
  (partition by year(o.orderdate) order by o.orderid) YTD
FROM orders o
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, dupplicates can occur : 1, 2, 3, 3, 5.
dense_rank() makes it so that there are no gaps withing the ranking $\longrightarrow$ 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:

```
SELECT orderid, orderdate, orderamount,
sum(orderamount) OVER
  (partition by year(o.orderdate) order by o.orderid
  range between unbouded preceding and current row) YTD
FROM orders o
ORDER BY orderid
```

with range, you have 3 valid options:

```
range between unbouded preceding and current row
range between current row and unbouded following
range between unbouded preceding and unbouded 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:

```
rows between N preceding and current row
rows between current row and N following
rows between N preceding and M following
```

4.2 LAG and LEAD

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