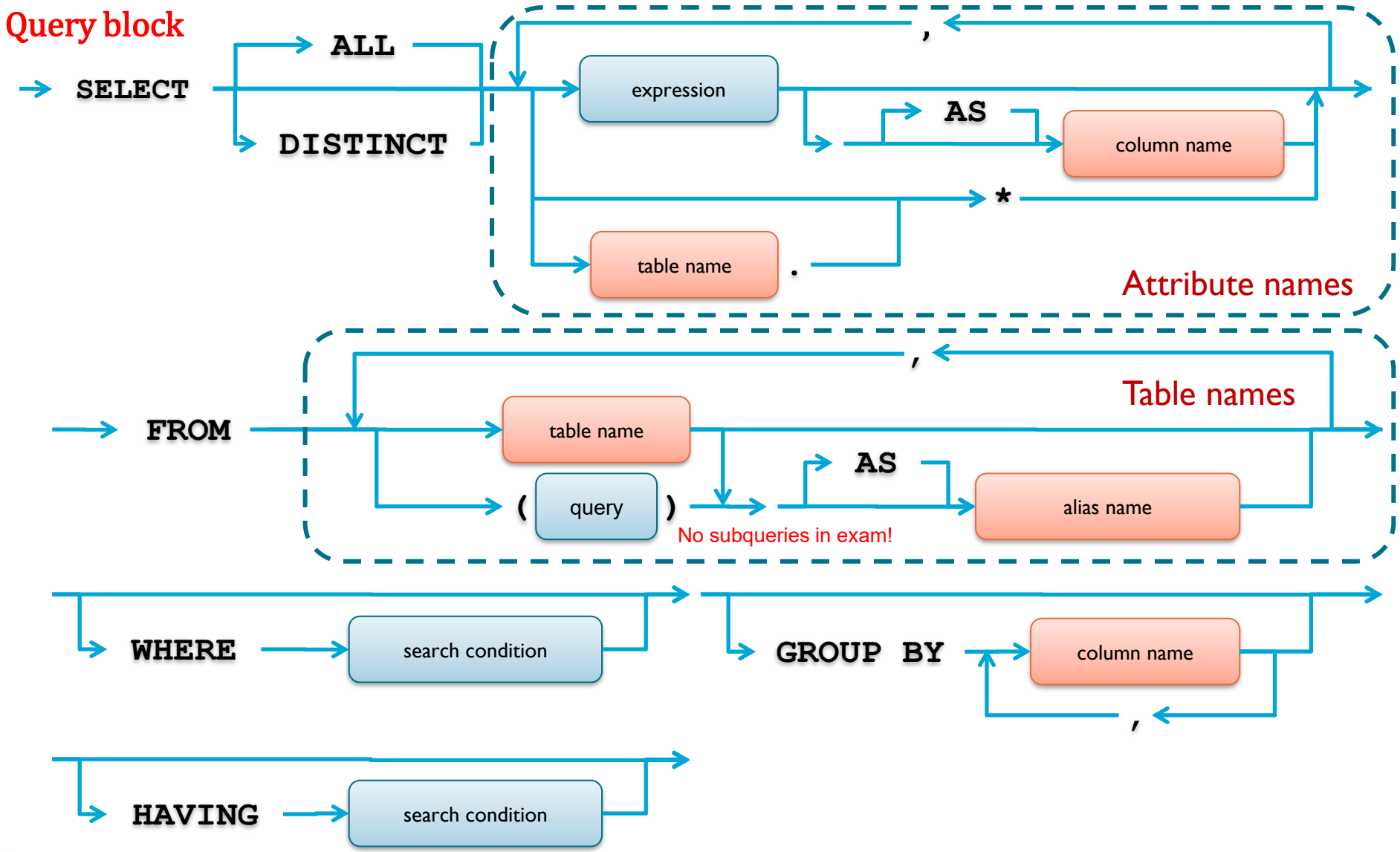
Web and Database Technology

SQL part II

Dr. Albert John Power

a.j.power@tudelft.nl

TU Delft WIS Web Information Systems


Query block
SELECT
ALL
DISTINCT
,
expression
AS
column name
*
table name
.
Attribute names
FROM
,
table name
AS
(
query
)
alias name
No subqueries in exam!
Table names
WHERE
search condition
GROUP BY
column name
,
HAVING
search condition

SQL JOINS

Addendum to last week

TU Delft WIS Web Information Systems

JOINs in SQL92

- CROSS JOIN
  - Cartesian Product
- INNER JOIN
  - Default type of join in a joined table (equivalent to JOIN)
  - Must specify JOIN attributes
    - Tuple is included in the results only if a matching tuple exists in the other relation
- LEFT OUTER JOIN
  - Every tuple in left table must appear in result
  - If no matching tuple: values for the attributes in the right table set to NULL
- RIGHT OUTER JOIN
  - Every tuple in right table must appear in result
  - If no matching tuple: values for the attributes in the left table set to NULL
- FULL OUTER JOIN
  - If no matching tuple: values for the attributes in the left and/or right tables set to NULL
- NATURAL JOIN on two relations R and S
  - No join condition specified
    - Implicit EQUI JOIN condition for each pair of attributes with the same name from R and S

Joining Tables

- A join combines tuples from one relation R1 with the matching tuples of another relation R2
- "Matching" is described by a join condition
  - These join conditions are typically (but not always) attribute value equality conditions along foreign key attributes

CROSS JOIN

A

n
1
2
3

B

c
x
y
z

```
SELECT *
FROM A
CROSS JOIN B
```

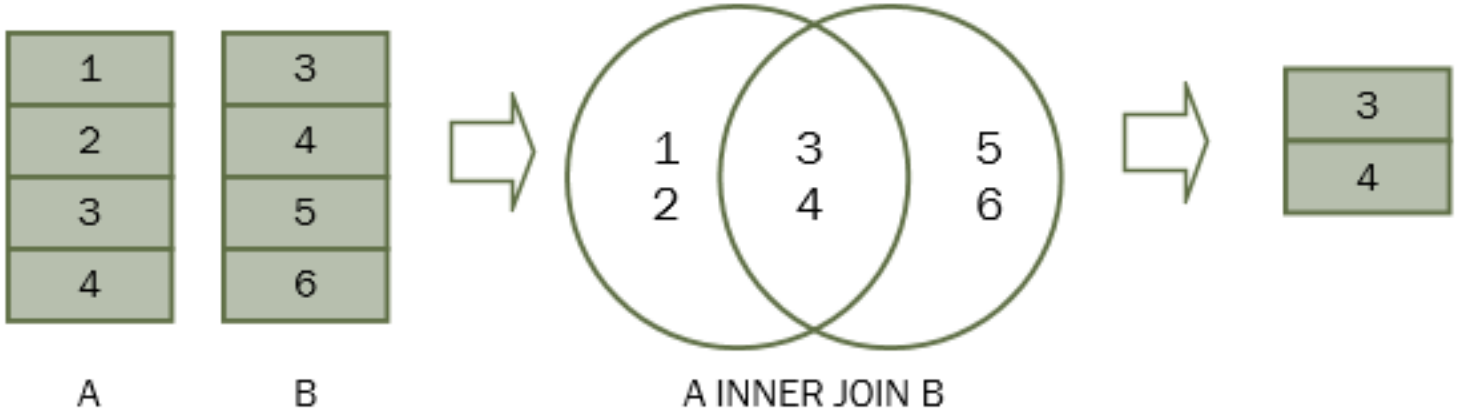
SELECT *

FROM A, B

A x B

n	c
1	x
1	y
1	z
2	x
2	y
2	z
3	x
3	y
3	z

INNER JOIN



INNER JOIN

- Remember: Find the names of suppliers of **at least one red product**

```
SELECT DISTINCT NAMES
FROM PRODUCTS JOIN SUPPLY USING (CODEP)
        JOIN SUPPLIER ON SUPPLIER.CODES=SUPPLY.CODES
WHERE COLOR = “RED"
```

Supplier

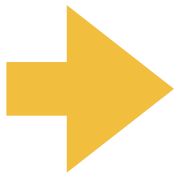
CodeS	NameS	Shareholders	Office
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

Supply

CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

Products

CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam



NameS
John
Victor

Self-Joins in SQL

- Example of Self-Joins:
  - Schema:
    - Person(id, name, mother->Person, father->Person, birthYear)
  - Question: “Who are the children of Genghis Khan?“
    - Self-Join needed for this:
    - ```
      SELECT p2.*
      FROM   person p1
      JOIN   person p2
      ON     p2.father=p1.id
      WHERE  p1.name='Genghis Khan'
      ```


      ```

Self-Joins in SQL

- Question: “Who are the GRAND-children of Genghis Khan?“
  - Person(id, name, mother-> Person, father->Person, birthYear)
  - Self-Join needed for this:
  - ```
    SELECT p3.*
    FROM   person p1
    JOIN   person p2
    ON     p2.father=p1.id
    JOIN   person p3
    ON     p3.mother=p2.id OR p3.father=p2.id
    WHERE  p1.name='Genghis Khan'
    ```
    ```

Self-Joins in SQL

- Question: “Who are the GRAND-GRAND-children of Genghis Khan?“
  - Person(id, name, mother->Person, father->Person, birthYear)
  - Self-Join needed for this:
  - ```
    SELECT p4.*
    FROM   person p1
    JOIN   person p2
    ON     p2.father=p1.id
    JOIN    person p3
    ON      p3.mother=p2.id OR p3.father=p2.id
    JOIN    person p4
    ON      p4.mother=p3.id OR p4.father=p3.id
    WHERE  p1.name='Genghis Khan'
    ```
- What about grand-grand-grand-grand children?
    ```

Self-Joins in SQL

- Question: “Who are the up to second generation descendants of Genghis Khan?“
  - Second generation descendants: children plus grand-children
  - ```
    SELECT p2.*
      FROM   person p1
      JOIN   person p2
      ON     p2.father=p1.id
      WHERE  p1.name='Genghis Khan'
    UNION
    SELECT p3.*
      FROM   person p1
      JOIN   person p2
      ON     p2.father=p1.id
      JOIN  person p3
      ON        p3.mother=p2.id OR p3.father=p2.id
      WHERE  p1.name='Genghis Khan'
    ```
    ```

More SQL

TU Delft WIS Web Information Systems

Example Databases

- Example DB1: Employees

Employee					
FirstName	**Surname**	**Dept**	**Office**	**Salary**	**City**
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

Department		
DeptName	**Address**	**City**
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San Joné

Example Databases

- Example DB2: Products

Supplier

CodeS	NameS	Shareholders	Office
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

Supply

CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

Products

CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam

Aggregate Queries

TU Delft WIS Web Information Systems

Aggregate Queries

- **Aggregate Query**: query in which the result depends on the consideration of **sets of rows**
- The result is a single (**aggregated**) value
- Expressed in the SELECT clause
  - Aggregate operators are evaluated on the rows accepted by the WHERE conditions
- SQL92 offers five aggregate operators:
  - COUNT, SUM, MAX, MIN, AVG
- Except for COUNT, these functions return a NULL value when no rows are selected

Operator COUNT

- COUNT returns the number of rows

```
COUNT (<* | [DISTINCT | ALL] TARGETLIST >)
```

- The DISTINCT keyword forces the count of distinct values in the attribute list

Students	
id	name
1	Albert
2	Melita
3	NULL
4	Albert

```
SELECT COUNT(name)
```
→ 3 (Nulls ignored)

```
SELECT COUNT(DISTINCT name)
```
→ 2 (Nulls and duplicates ignored)

```
SELECT COUNT(*)
```
→ 4 (counts all rows)

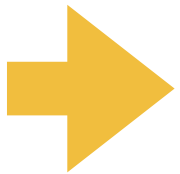
COUNT Example /1

- Find the **number of suppliers** in the database

```
SELECT COUNT (*)
 FROM  SUPPLIER
```

Supplier

CodeS	NameS	Shareholders	Office
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht



count
5

COUNT Example /2

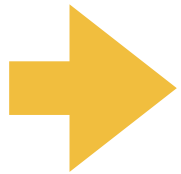
- Is this correct?

- Find the **number of suppliers** with at least one supply

  - Equivalent to SELECT COUNT(CodeP) or SELECT COUNT(CodeS)?

```
SELECT COUNT (*)
 FROM  SUPPLY
```

Supply		
CodeS	**CodeP**	**Amount**
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400



count
12

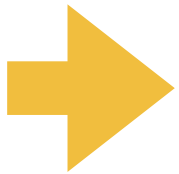
COUNT Example /2

- Is this correct?

- Find the **number of suppliers** with at least one supply
  - Equivalent to SELECT COUNT(CodeP) or SELECT COUNT(CodeS)?

```
SELECT COUNT (*)
 FROM  SUPPLY
```

Supply		
CodeS	**CodeP**	**Amount**
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400



count
12

In this other example, we want to return the number of suppliers with AT LEAST one supply.
The query in the slide, while being syntactically correct, is semantically wrong. The reason is simple: counting rows does not satisfy our query, as suppliers might supply multiple products.

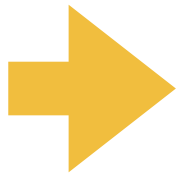
COUNT Example /3

- Find the **number of suppliers** with at least one supply

```
SELECT COUNT (DISTINCT CODES)
 FROM  SUPPLY
```

Supply

CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400



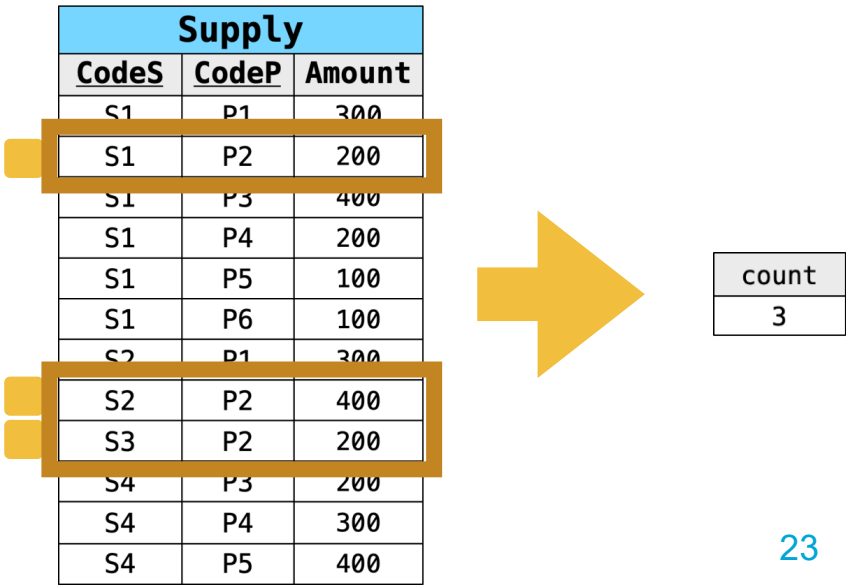
count
4

COUNT Example /4

- Is this correct?

- **Count** the number of suppliers that supply the product “P2”

```
SELECT COUNT(*)
 FROM  SUPPLY
 WHERE CODEP = ‘P2’
```

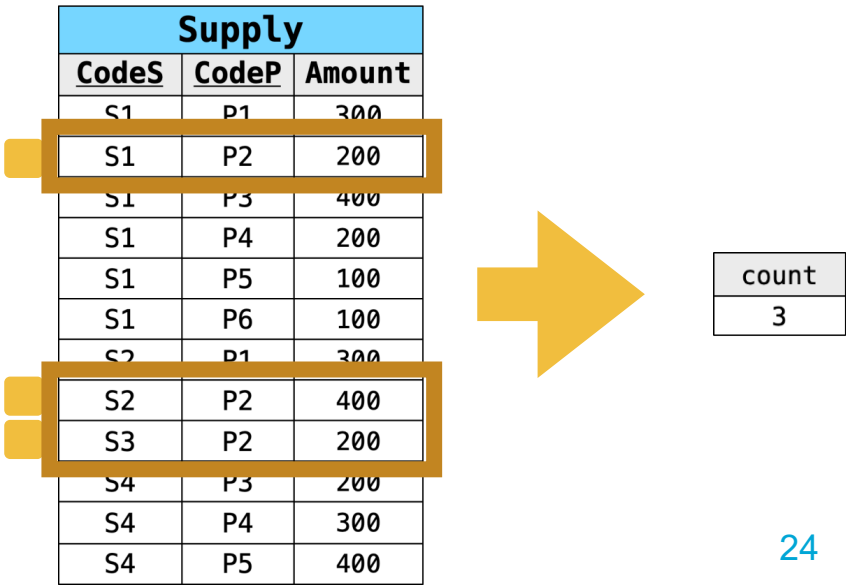


COUNT Example /4

- Is this correct?

- **Count** the number of suppliers that supply the product “P2”

```
SELECT COUNT(*)
 FROM  SUPPLY
 WHERE CODEP = ‘P2’
```



Supply		
CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

count
3

In this example, we can notice how the query might return the right result, despite the query being semantically wrong. The presence of a second supply of P2 from S1, S2, or S3 would have made the returned result incorrect.

Operators SUM, MAX, MIN, AVG

- SUM, MAX, MIN, AVG
  - Allowed arguments are attributes or expressions
- SUM, AVG
  - Only numeric types
- MAX, MIN
  - Attribute must be sortable
  - Applied also on strings and timestamps

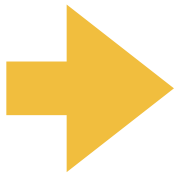
SUM Example

- Find the **total** number of supplied items for product “P2”

```
SELECT SUM(AMOUNT)
 FROM  SUPPLY
 WHERE CODEP = ‘P2’
```

Supply

CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400



CodeS	CodeP	Amount
S1	P2	200
S2	P2	400
S3	P2	200

count
800

NULL Values and Aggregates

- All aggregate operations ignore tuples with NULL values on the aggregated attributes
  - COUNT: number of input rows for which the value of expression is not NULL
  - SUM, AVG, MAX, MIN: NULL values are not considered
- The **COALESCE** function can be used in an aggregate function to change temporarily a value that was NULL

```
SELECT AVG(SEASON_NR)
 FROM  TITLE_100K
```

```
SELECT AVG(COALESCE(SEASON_NR,1))
 FROM  TITLE_100K
```

Aggregate Query and Target List

- This is an incorrect query:

```
SELECT FIRSTNAME, SURNAME, MAX(SALARY)
 FROM  EMPLOYEE JOIN DEPARTMENT ON DEPT = DEPTNAME
 WHERE DEPARTMENT.CITY = 'LONDON'
```

- Whose name?
- SQL syntax does not allow aggregate functions and attribute expressions within the same target list
- Now we operate on all the rows that are produced as the result of the query
- Queries may apply aggregate operators to subsets of rows
- The GROUP BY clause will help us

Target Lists

- This is an incorrect query

```
SELECT FIRSTNAME, SURNAME, MAX(SALARY)
 FROM  EMPLOYEE JOIN DEPARTMENT ON DEPT = DEPTNAME
 WHERE DEPARTMENT.CITY = 'LONDON'
```

- Whose name? The target list must be homogeneous
- The GROUP BY clause will help us

Aggregation operators need the query to possess specific properties.
Consider the query in the slide. On an intuitive level, this query would select the highest value of the "Salary" attribute, and thus would automatically select the attributes "FirstName" and "Surname" of the corresponding employee.
However, such semantics could not be generalized to the aggregate queries, for two reasons. In the first place, there is no guarantee that the operator will select a single element, given that there could be more than one row containing a particular salary.

In the second place, written like this, the query could be applied to the operators max and min, but would have no meaning for the other aggregate operators.

Therefore, the SQL syntax does not allow aggregate functions and attribute expressions (such as, for example, attribute names) within the same target list The examples that we have seen so far, operate on all the rows that are produced as the result of the query.

It is often necessary to apply an aggregate operator more finely, to specific subsets of rows. To use aggregate operators like this, SQL provides the GROUP BY clause, which makes it possible to specify how to divide the table up into subsets. The clause accepts as argument a set of attributes, and the query will operate separately on each set of rows that possess the same values for this set of attributes.

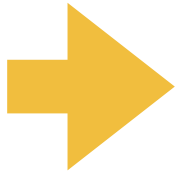
Grouping Rows

- **For each product** find the total amount of supplied items

```
SELECT CODEP, SUM(AMOUNT)
 FROM  SUPPLY
 GROUP BY CODEP
```

Supply

CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400



CodeS	CodeP	Amount
S1	P1	300
S2	P1	300
S1	P2	200
S2	P2	400
S3	P2	200
S1	P3	400
S4	P3	200
S1	P4	200
S4	P4	300
S1	P5	100
S4	P5	400
S1	P6	100

CodeP	Amount
P1	600
P2	800
P3	600
P4	500
P5	500
P6	100

GROUP BY clause /1

- The order of the grouping attributes does not matter
- The SELECT clause can contain:
  - Attributes specified in the GROUP BY clause
  - Aggregated functions
  - Attributes univocally determined by attributes already specified in the GROUP BY clause

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

Department

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San Joné

```
SELECT OFFICE
FROM EMPLOYEE
GROUP BY DEPT
```

▸ Incorrect Query

GROUP BY clause /1

- The order of the grouping attributes does not matter
- The SELECT clause can contain:
  - Attributes specified in the GROUP BY clause
  - Aggregated functions
  - Attributes univocally determined by attributes already specified in the GROUP BY clause

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

Department

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San Joné

```
SELECT OFFICE
FROM EMPLOYEE
GROUP BY DEPT
```

▸ Incorrect Query

The SQL syntax imposes the restriction that, whenever the GROUP BY clause is used, the attributes that can appear in the SELECT clause must be a subset of the attributes used in the GROUP BY clause. References to these attributes are possible because each tuple of the group will be characterized by the same values. The example in this figure shows the reasons for this limitation.

This query is incorrect, in that a number of values of the Office attribute will correspond to each value of the Dept attribute. Instead, after the grouping has been carried out, each sub-group of rows must correspond to a single row in the table resulting from the query.

GROUP BY clause /2

- The order of the grouping attributes does not matter
- The SELECT clause can contain:
  - Attributes specified in the GROUP BY clause
  - Aggregated functions
  - Attributes univocally determined by attributes already specified in the GROUP BY clause

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

```
SELECT DEPTNAME,D.CITY,COUNT(*)
FROM EMPLOYEE E JOIN DEPARTMENT D ON E.DEPT=D.DEPTNAME
GROUP BY DEPTNAME
```

▸ Incorrect Query

Department

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San Joné

GROUP BY clause /2

- The order of the grouping attributes does not matter
- The SELECT clause can contain:
  - Attributes specified in the GROUP BY clause
  - Aggregated functions
  - Attributes univocally determined by attributes already specified in the GROUP BY clause

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

This second example, shows again why the select clause can contain only attributes specified in the GROUP BY clause and aggregate functions.
In this case, departments with same name can actually be located in different cities.

```
SELECT DEPTNAME,D.CITY,COUNT(*)
FROM EMPLOYEE E JOIN DEPARTMENT D ON E.DEPT=D.DEPTNAME
GROUP BY DEPTNAME
```

▸ Incorrect Query

Department

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San Joné

GROUP BY clause /3

- The order of the grouping attributes does not matter
- The SELECT clause can contain:
  - Attributes specified in the GROUP BY clause
  - Aggregated functions
  - Attributes univocally determined by attributes already specified in the GROUP BY clause

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

```
SELECT DEPTNAME,D.CITY,COUNT(*)
 FROM  EMPLOYEE E JOIN DEPARTMENT D ON E.DEPT=D.DEPTNAME
 GROUP BY DEPTNAME, D.CITY
```

▸ Correct Query

Department

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San Joné

Grouping Rows

- **For each product** sold by suppliers in Den Haag, find the total amount of supplied items

```
SELECT CODEP, SUM(AMOUNT)
 FROM  SUPPLY JOIN SUPPLIER ON SUPPLY.CODES = SUPPLIER.CODES
 WHERE OFFICE = 'DEN HAAG'
 GROUP BY CODEP
```

Supplier

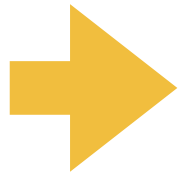
CodeS	NameS	Shareholders	Office
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

Supply

CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

Supplier				Supply		
CodeS	NameS	Shareholders	Office	CodeS	CodeP	Amount
S1	John	2	Amsterdam	S1	P1	300
S1	John	2	Amsterdam	S1	P2	200
S1	John	2	Amsterdam	S1	P3	400
S1	John	2	Amsterdam	S1	P4	200
S1	John	2	Amsterdam	S1	P5	100
S1	John	2	Amsterdam	S1	P6	100
S2	Victor	1	Den Haag	S2	P1	300
S2	Victor	1	Den Haag	S2	P1	300
S2	Victor	1	Den Haag	S2	P2	400
S3	Anna	3	Den Haag	S3	P2	200
S4	Angela	2	Amsterdam	S4	P3	200
S4	Angela	2	Amsterdam	S4	P4	300
S4	Angela	2	Amsterdam	S4	P5	400

CodeP	Amount
P1	300
P1	300
P2	200
P2	400



CodeP	Amount
P1	600
P2	600

GROUP BY clause

```
SELECT department
FROM  employees
GROUP BY department
```

```
SELECT  DISTINCT department
FROM  employees
```

HAVING clause /1

- Conditions on the result of an aggregate operator require the HAVING clause
- Only predicates containing aggregate operators should appear in the argument of the HAVING clause

- Find the departments in which the average salary of employees working in office number 20 **is higher than 25**

```
SELECT DEPT
 FROM EMPLOYEE
 WHERE OFFICE = '20'
 GROUP BY DEPT
 HAVING AVG(SALARY) > 25
```

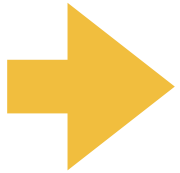
Grouping Rows

- **For each product** find the total amount of supplied items

```
SELECT CODEP, SUM(AMOUNT)
 FROM  SUPPLY
 GROUP BY CODEP
```

Supply

CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400



CodeS	CodeP	Amount
S1	P1	300
S2	P1	300
S1	P2	200
S2	P2	400
S3	P2	200
S1	P3	400
S4	P3	200
S1	P4	200
S4	P4	300
S1	P5	100
S4	P5	400
S1	P6	100

CodeP	Amount
P1	600
P2	800
P3	600
P4	500
P5	500
P6	100

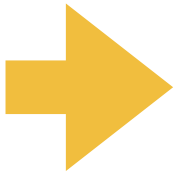
HAVING clause /2

- Find the total number of supplied items for products that count **at least 600 total supplied items**

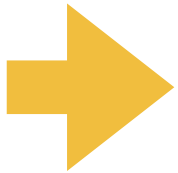
```
SELECT CODEP, SUM(AMOUNT)
 FROM  SUPPLY
 GROUP BY CODEP
 HAVING SUM(AMOUNT) >= 600
```

Supply

CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400



CodeS	CodeP	Amount
S1	P1	300
S2	P1	300
S1	P2	200
S2	P2	400
S3	P2	200
S1	P3	400
S4	P3	200
S1	P4	200
S4	P4	300
S1	P5	100
S4	P5	400
S1	P6	100



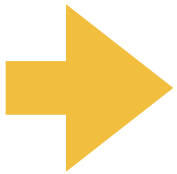
CodeP	Amount
P1	600
P2	800
P3	600

HAVING clause /3

- Find the code of red products supplied by more than one supplier

```
SELECT SUPPLY.CODEP
 FROM  SUPPLY JOIN PRODUCTS ON SUPPLY.CODEP = PRODUCT.CODEP
 WHERE COLOR = 'RED'
 GROUP BY SUPPLY.CODEP
 HAVING COUNT(*) > 1
```

Products					Supply		
CodeP	NameP	Color	Size	Storehouse	CodeS	CodeP	Amount
P1	Sweater	Red	40	Amsterdam	S1	P1	300
P1	Sweater	Red	40	Amsterdam	S2	P1	300
P2	Jeans	Green	48	Den Haag	S1	P2	200
P2	Jeans	Green	48	Den Haag	S2	P2	400
P2	Jeans	Green	48	Den Haag	S3	P2	200
P3	Shirt	Blu	48	Rotterdam	S1	P3	400
P3	Shirt	Blu	48	Rotterdam	S4	P3	200
P4	Shirt	Blu	44	Amsterdam	S1	P4	200
P4	Shirt	Blu	44	Amsterdam	S4	P4	300
P5	Skirt	Blu	40	Den Haag	S1	P5	100
P5	Skirt	Blu	40	Den Haag	S4	P5	400
P6	Coat	Red	42	Amsterdam	S1	P6	100



CodeP
P1

HAVING clause

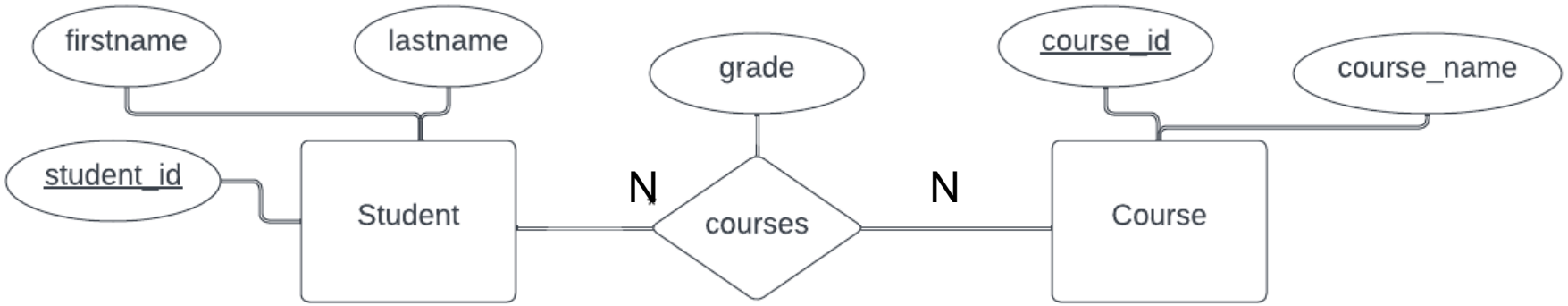
```
SELECT salary
FROM employees
HAVING salary > 5000;
```

```
SELECT salary
FROM employees
WHERE salary > 5000;
```

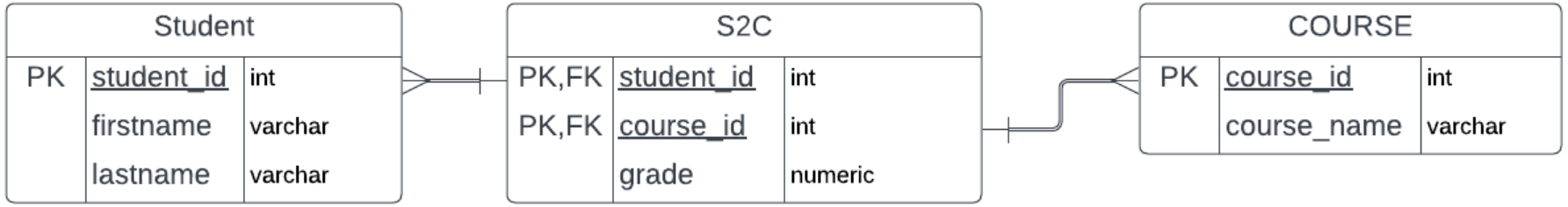
Data Definition

TU Delft WIS Web Information Systems

Simple Student DB



Conceptual Schema



Logical Relational Schema

Classroom Examples

- Create Schema
- Create Tables
  - Names
  - Domains
  - Constraints
    - Columns
    - Table
    - PK, FK

```
CREATE SCHEMA studentDB; -- This creates a new empty schema in the system catalog.
SET search_path TO studentDB; -- Use the schema

CREATE TABLE student  (
student_id int,
firstname varchar(100) NOT NULL, -- This is variable strring max length 100. It cannot be NULL.
lastname varchar(100) NOT NULL,
CONSTRAINT pl_student PRIMARY key (student_id), -- Student_id is the primary key!
UNIQUE (firstname, lastname) -- The combination of firstname&lastname needs to be unique (which is
stupid, btw. Just to demonstrate the syntax)
);

CREATE TABLE courses (
course_id int PRIMARY KEY, -- Course_id needs is the primary key (and thus automatically unique)
course_name varchar(100) UNIQUE -- Course names are unique but are not the primary key
);

CREATE TABLE student2courses (-- This table can only be created after the other 2!
student_id int REFERENCES student(student_id), -- Foreign Key Constraint! Any student_id in this
table needs to be a valid student_id in student
course_id int REFERENCES courses(course_id),
grade NUMERIC(2, 1), --This is a precise numeric type with one digit before and one after the comma
CONSTRAINT pk_s2c PRIMARY KEY (student_id, course_id) -- Composite primary key
);
```

Defining a Database Schema

- A database schema comprises:
  - Declarations for the relations (“tables”) of the database
  - Domains associated with each attribute
  - Integrity constraints
- A schema has a *name* and an *owner* (the authorization)
- Many other kinds of elements may also appear in the database schema, including:
  - Privileges, views, indexes, triggers
- Syntax:

```
CREATE SCHEMA [ SCHEMANAME ]
[[ AUTHORISATION ] AUTHORISATION ]
{ SCHEMAELEMENTDEFINITION }
```

Table Definition

- An SQL table consists of:
  - An ordered set of attributes
  - A (possibly empty) set of constraints

- Statement CREATE TABLE
  - Defines a relation schema, creating an empty instance
- Constraints: integrity checks on attributes
- Other Constraints: integrity constraints on the table
- Syntax:

```
CREATE TABLE TABLENAME (
  ATTRIBUTENAME DOMAIN [DEFAULTVALUE] [CONSTRAINTS]
  {, ATTRIBUTENAME DOMAIN [DEFAULTVALUE] [CONSTRAINTS]}
  [OTHERCONSTRAINTS]
)
```

Example of CREATE table

- Table Definition:

```
CREATE TABLE EMPLOYEE (
                                REGNO       CHARACTER(6) PRIMARY KEY
                                FIRSTNAME CHARACTER(20) NOT NULL,
                                SURNAME    CHARACTER(20) NOT NULL,
                                DEPT          CHARACTER(15) REFERENCES DEPARTMENT(DEPTNAME)
                                        ON DELETE SET NULL
                                        ON UPDATE CASCADE
                                SALARY      DECIMAL (9) DEFAULT 0,
                                CITY          CHARACTER(15),
                                UNIQUE(SURNAME, FIRSTNAME)
)
```

Domains

- Specify the content of attributes
- Two categories
  - **Elementary** (predefined by the standard)
  - **User-defined** (not available in all RDBMs implementations)

```
CREATE DOMAIN GRADE AS SMALLINT
        DEFAULT NULL
        CHECK (GRADE >= 0 AND GRADE <=10)
```

Elementary Domains (Data Types)

Category	Data Types
Numeric	SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE PRECISION, SERIAL, BIGSERIAL, SMALLSERIAL
Character	CHAR(n), VARCHAR(n), TEXT
Boolean	BOOLEAN
Date/Time	DATE, TIME, TIMESTAMP, INTERVAL
Binary	BYTEA
UUID	UUID
Geometric	POINT, LINE, LSEG, BOX, PATH, POLYGON, CIRCLE
Network	INET, CIDR, MACADDR
JSON/XML	JSON, JSONB, XML
Array	ARRAY
Bit	BIT(n), BIT VARYING(n)
Special	ENUM, TSVECTOR, TSQUERY

Elementary Domains (Data Types) /1

- Numeric Domains Integers
  - **INTEGER** (or **INT**): a finite subset of the integers that is machine-dependent (4 bytes)
  - **SMALLINT**: a machine-dependent subset of the integer domain type (2 bytes, range −32,768 to 32,767)
  - **BIGINT** (8 bytes)

  - **SERIAL** (PostgreSQL), AUTO_INCREMENT (MySQL), (4 bytes)
  - **SMALLSERIAL** (2 bytes , range: 1–32,767)
  - **BIGSERIAL** (8 bytes)

Range?

```
CREATE TABLE student  (
student_id INT PRIMARY KEY,
firstname varchar(100) NOT NULL
);
```

```
CREATE TABLE student  (
student_id SERIAL PRIMARY KEY,
firstname varchar(100) NOT NULL
);
```

Elementary Domains (Data Types) /2

- **Real values (Exact and Approximate)**
  - **NUMERIC** [(Precision [, Scale])]: fixed point number, with user-specified Precision digits, of which Scale digits to the right of decimal point.
  - **DECIMAL** [(Precision [, Scale])]: functionally equivalent to NUMERIC

  - **REAL**: floating point numbers, with machine-dependent precision (4 bytes)
  - **DOUBLE PRECISION**: double-precision floating point numbers, with machine-dependent precision (8 bytes)
  - **FLOAT** [(Precision)]: floating point number, with user-specified precision of at least n digits.

  PostgreSQL (Float(n), n ≤ 24 Real, n > 25 Double Precision),

  MySQL (FLOAT[(precision, scale)]),

  Microsoft SQL Server (FLOAT[(n)]),

  Oracle, (FLOAT[(n)])

Elementary Domains (Data Types) /3

- Character Domains
  - **CHAR(n)** (or **CHARATER(n)**)
  - **VARCHAR(n)**
  - **TEXT**

Elementary Domains (Data Types) /4

- Temporal **Instants**
  - DATE: format yyyy-mm-dd
  - TIME [(Precision)] [with time zone]: format hh:mm:ss:p with an optional decimal point and fractions of a second following.
    - TIMESTAMP [(Precision)] [with time zone]: format yyyy-mm-dd hh:mm:ss:p
- Temporal **intervals**
  - INTERVAL FirstUnitOfTime [TO LastUnitOfTime]
  - Units of time are divided into two groups:
    - Year, month
    - Day, hour, minute, second
  - In PotgreSQL the syntax is different: interval "2 months ago"

Elementary Domains (Data Types) /5

- Geometric Types: two-dimensional spatial object
  - Point, line, lseg, box, path, Open path, polygon, circle
- Network Address Types: to store IPv4, IPv6, and MAC addresses
  - cidr, inet, macaddr, macaddr8
- JSON Types
  - Json: data is stored an exact copy of the input text
  - Jsonb: data is stored in a decomposed binary format
- XML Type, used to store XML data
- Composite Types: represents the structure of a row or record
- UUID, Array, Ranges, Text Search (to support full text search)

Default Domain Values

- Define the value that the attribute must assume when a value is not specified during row insertion
- Syntax:

```
DEFAULT < GENERICVALUE | USER | CURRENT_USER | SESSION_USER | SYSTEM_USER | NULL >
```

```
status BOOLEAN DEFAULT TRUE
```

- GenericValue represents a value compatible with the domain, in the form of a constant or an expression
- USER* is the login name of the user who issues the command

```
CREATE TABLE EMPLOYEE (
 REGNO      CHARACTER(6) PRIMARY KEY
 FIRSTNAME CHARACTER(20) NOT NULL,
 SURNAME    CHARACTER(20) NOT NULL,
 DEPT       CHARACTER(15) REFERENCES DEPARTMENT(DEPTNAME)
                ON DELETE SET NULL
                ON UPDATE CASCADE
 SALARY     DECIMAL (9) DEFAULT 0,
 CITY       CHARACTER(15),
 UNIQUE(SURNAME, FIRSTNAME)
 )
```

Constraints /1

- Constraints are conditions that must be verified by every database instance
- Defined in the CREATE or ALTER TABLE operations
- Automatically verified by the DB after each operation
- Advantages:
  - Declarative specification of constraints
  - Unique centralized verification
- Disadvantages:
  - Might slow down execution
  - Pre-defined type of constraints
    - E.g., no constraint on aggregated data
    - But triggers can help

```
CREATE TABLE EMPLOYEE (
REGNO       CHARACTER(6) PRIMARY KEY
FIRSTNAME CHARACTER(20) NOT NULL,
SURNAME    CHARACTER(20) NOT NULL,
DEPT          CHARACTER(15) REFERENCES
                    DEPARTMENT(DEPTNAME)
                    ON DELETE SET NULL
                    ON UPDATE CASCADE
SALARY      DECIMAL (9) DEFAULT 0,
CITY          CHARACTER(15),
UNIQUE(SURNAME, FIRSTNAME))
```

Constraints /2

- An operation that violates a constraints might cause two type of reactions:
  - The operation is **aborted**, causing an application error
  - A **compensation action** is taken, to reach a new consistent state

- Three type of constraints:
  - Intra-relational constraints (or **table constraints**)
  - Inter-relational constraints (or **referential integrity constraints -> Used for foreign keys**)
  - *Generic* integrity constraints and **assertions**

Intra-relational Constraints

- Intra-relational constraints involve a single relation
- **NOT NULL** (on single attributes)
  - Upon tuple insertion, the attribute MUST be specified
- **UNIQUE**: permits the definition of candidate keys
  - For single attributes: UNIQUE, after the domain
  - For multiple attributes: UNIQUE (Attribute, Attribute)
- **PRIMARY KEY**: defines the primary key
  - Once for each table
  - Implies NOT NULL,
  - (typically) implies UNIQUE,
  - Syntax like UNIQUE

```
CREATE TABLE student  (
student_id int,
firstname varchar(100) NOT NULL,
lastname varchar(100) NOT NULL,
CONSTRAINT pl_student PRIMARY key (student_id),
UNIQUE (firstname, lastname));
```

```
CREATE TABLE student  (
student_id int,
firstname varchar(100) NOT NULL,
lastname varchar(100) NOT NULL,
PRIMARY key (student_id));
```

```
CREATE TABLE student  (
student_id int PRIMARY key,
firstname varchar(100) NOT NULL,
lastname varchar(100) NOT NULL);
```

```
CREATE TABLE employee  (
firstname varchar(100),
lastname varchar(100),
PRIMARY key (firstname, lastname));
```

PRIMARY KEY vs. UNIQUE

- The SQL standard allows DBMS implementers to make their own distinctions between PRIMARY KEY and UNIQUE
  - Example: some DBMS might automatically create an index (data structure to speed search) in response to PRIMARY KEY, but not UNIQUE

- However, standard SQL requires these distinctions:
  - There can be only one PRIMARY KEY for a relation, but several UNIQUE attributes
  - No attribute of a PRIMARY KEY can ever be NULL in any tuple
  - Attributes declared UNIQUE may have NULLs
    - And there may be several tuples with NULL!

Unique Constraints

- Each pair of FirstName and Surname uniquely identifies each element:

```
FIRSTNAME CHARACTER(20) NOT NULL
SURNAME CHARACTER(20) NOT NULL
UNIQUE(FIRSTNAME, SURNAME)
```

- Note the difference with the following (stricter) definition:

```
FIRSTNAME CHARACTER(20) NOT NULL UNIQUE
SURNAME CHARACTER(20) NOT NULL UNIQUE
```

Foreign Key Constraints

- Constraints may take into account several relations. **REFERENCES** and **FOREIGN KEY** permit the definition of referential integrity constraints. Syntax:
  - For single attributes: REFERENCES, after the domain
  - For multiple attributes: FOREIGN KEY (Attribute1 , Attribute2) REFERENCES Table (Attribute1 , Attribute2)

Foreign Key Constraints

Referenced

```
CREATE TABLE EMPLOYEE (
        REGNO     CHARACTER(6) PRIMARY KEY,
        FIRSTNAME CHARACTER(20) NOT NULL,
        SURNAME   CHARACTER(20) NOT NULL,
        DEPT      CHARACTER(15)  REFERENCES DEPARTMENT(DEPTNAME)
                                 ON DELETE SET NULL
                                 ON UPDATE CASCADE
        SALARY    DECIMAL (9,2) DEFAULT 0,
        CITY      CHARACTER(15),
        UNIQUE(SURNAME, FIRSTNAME)
)
```

Referencing

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown		10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

Department

DeptName	Address	City
	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San Joné

It is possible to associate reaction policies to violations of referential integrity

Reaction Policies

- Reactions operate on the referencing table, after changes to the referenced table
- Violations may be introduced
  - By updates on the referred attribute (if the primary key of a referenced row is modified)
  - By row deletions (if a referenced row is deleted)
- Reactions (can be specific to an event)
  - CASCADE: propagate the change
  - SET NULL: nullify the referring attribute
  - SET DEFAULT: assign the default value to the referring attribute
  - NO ACTION: reject the modification or the deletion on the external table (it cannot be performed)
  - The default action is:
    - `ON DELETE NO ACTION ON UPDATE NO ACTION`
- Syntax:

```
ON < DELETE | UPDATE > < CASCADE | SET NULL | SET DEFAULT | NO ACTION >
```

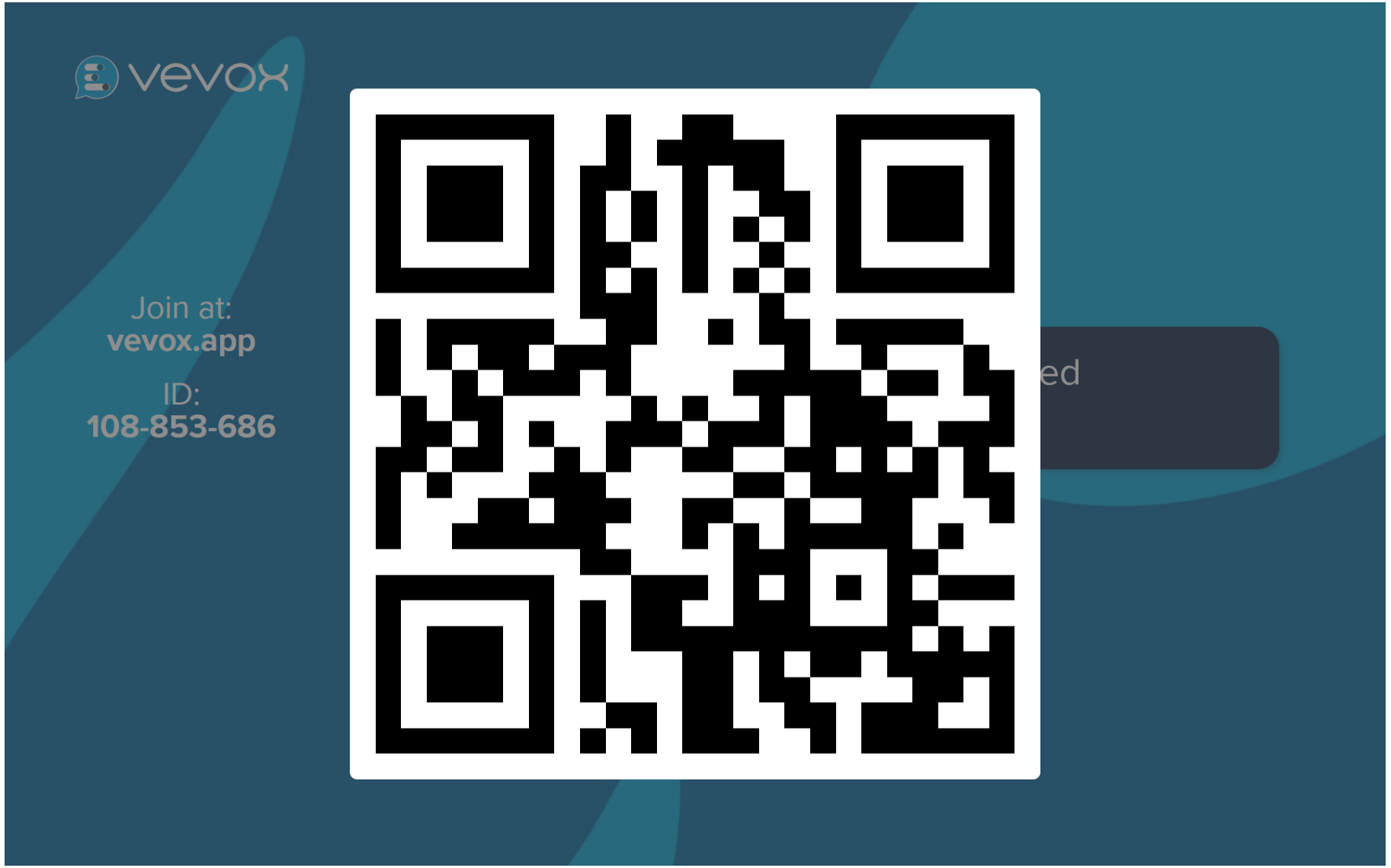
Check - Constraint

- **Column constraints**:
  - Restricts possible values for the current column
  - May have a **unique name** indicated by **CONSTRAINT** <name>
    - If the name is omitted, the system creates a default name
  - **CHECK**: user-defined constraint. To be valid, values have to satisfy the condition.
  - Example:
    - ```
      CREATE TABLE student(
            name VARCHAR(200),
            grade INTEGER CONSTRAINT validGrade
                CHECK (grade >= 0 and grade <= 10)
      )
      ```
      ```

Schema Update & Delete

- SQL statements:
  - **ALTER**: to modify a domain, the schema of a table, or a user
  - **DROP**: to remove schema, domain, table, etc.

```
ALTER TABLE DEPARTMENT ADD COLUMN NOOFOFFICES NUMERIC(4)

ALTER TABLE DEPARTMENT ADD CONSTRAINT UNIQUEADDRESS UNIQUE(ADDRESS)

DROP TABLE TEMPTABLE CASCADE
```

Data Manipulation

TU Delft WIS Web Information Systems

Data Modification in SQL

- Statements for:
  - Insertion - **INSERT**
  - Deletion - **DELETE**
  - Change of attribute values - **UPDATE**
- All the statements **can operate on a set of tuples** (set-oriented)
- In the condition, it is possible to access other relations

Insertions /1

- Using Values:

```
INSERT INTO TABLENAME [(ATTRIBUTELIST)] <VALUES(LISTOFVALUES)|SELECT SQL>
```

```
INSERT INTO DEPARTMENT(DEPTNAME,CITY) VALUES ('PRODUCTION','TOULOUSE')
```

- Using a subquery:

```
INSERT INTO LONDONPRODUCTS(
   SELECT CODE, DESCRIPTION
   FROM PRODUCT
   WHERE PRODAREA = `LONDON`
)
```

Insertions /2

- The ordering of the attributes (if present) and of the values is meaningful (first value with the first attribute, and so on)
- If *AttributeList* is omitted, all the relation attributes are considered, in the order in which they appear in the table definition
- If *AttributeList* does not contain all the relation attributes, to the remaining attributes it is assigned:
  - The DEFAULT value (if defined)
  - The NULL value
  - PRIMARY KEYs might get special handling

Deletions /1

- The DELETE statement removes from the table all the tuples that satisfy the condition

```
DELETE FROM TABLENAME [WHERE CONDITION]
```

- The removal may produce deletions from other tables if a referential integrity constraint with CASCADE policy has been defined
- If WHERE clause is omitted, DELETE removes all the tuples

Deletions /2

- To remove all the tuples from DEPARTMENT keeping the table:

```
DELETE FROM DEPARTMENT
```

- To remove table DEPARTMENT completely (content and schema):

```
DROP TABLE DEPARTMENT CASCADE
```

- Remove the Production department:

```
DELETE FROM DEPARTMENT WHERE DEPTNAME = `PRODUCTION`
```

- Remove the departments without employees:

```
DELETE FROM DEPARTMENT WHERE DEPTNAME NOT IN (SELECT DEPT FROM EMPLOYEE)
```

Updates /1

- Examples:

```
UPDATE TABLENAME
  SET ATTRIBUTE = <EXPRESSION | SELECT SQL | NULL | DEFAULT>
  {, ATTRIBUTE = <EXPRESSION | SELECT SQL | NULL | DEFAULT>}
  [WHERE CONDITION]
```

```
UPDATE EMPLOYEE
  SET SALARY = SALARY + 5
  WHERE FIRSTNAME = 'MARY' AND LASTNAME = 'BROWN'
```

```
UPDATE EMPLOYEE
  SET SALARY = SALARY * 1.1
  WHERE DEPT = 'ADMINISTRATION'
```

Updates /2

- Since the language is set oriented, the order of the statements is important

```
UPDATE EMPLOYEE
  SET SALARY = SALARY * 1.1
  WHERE SALARY <= 30
```

```
UPDATE EMPLOYEE
  SET SALARY = SALARY * 1.15
  WHERE SALARY > 30
```

- If the statements are issued in this order, some employees may get a double raise

Updates /3

- Replace the real name of each hero with NULL.
  - `UPDATE hero SET real_name = NULL`
- Multiply all power_strength values by 10.
  - ```
    UPDATE has_power
        SET power_strength = power_strength * 10
    ```
- Change the name of the hero with id 1.
  - ```
    UPDATE hero
        SET name= 'Charles Francis Xavier'
    WHERE id = 1
    ```
- Change the name and id of Jean Grey.
  - ```
    UPDATE hero
        SET (id, name) = ('3', 'Jean Grey-Summers')
    WHERE name = 'Jean Grey'
    ```
  - Change of id is propagated to other tables when **`ON UPDATE CASCADE`** is used in table definition
- Again, subqueries can be used in the **`WHERE`** clause
    ```

QUIZ



Web & Database Technology

See you next time!

TU Delft WIS Web Information Systems