CSE1500 - WEB AND DATABASE TECHNOLOGY

DB LECTURE 4

MORE ON SQL

Alessandro Bozzon

cse1500-ewi@tudelft.nl

AT THE END OF THIS LECTURE, YOU SHOULD BE ABLE TO….

- **Describe** and **design** SQL programs for the retrieval of data from tables
- **Describe** and **design** SQL programs for the creation, altering, and manipulation of tables
- **Develop** logical database schema, with principled design that enforce data integrity
- **Prototype** and **deploy** database applications using open-source database systems (e.g., PostgreSQL)

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 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

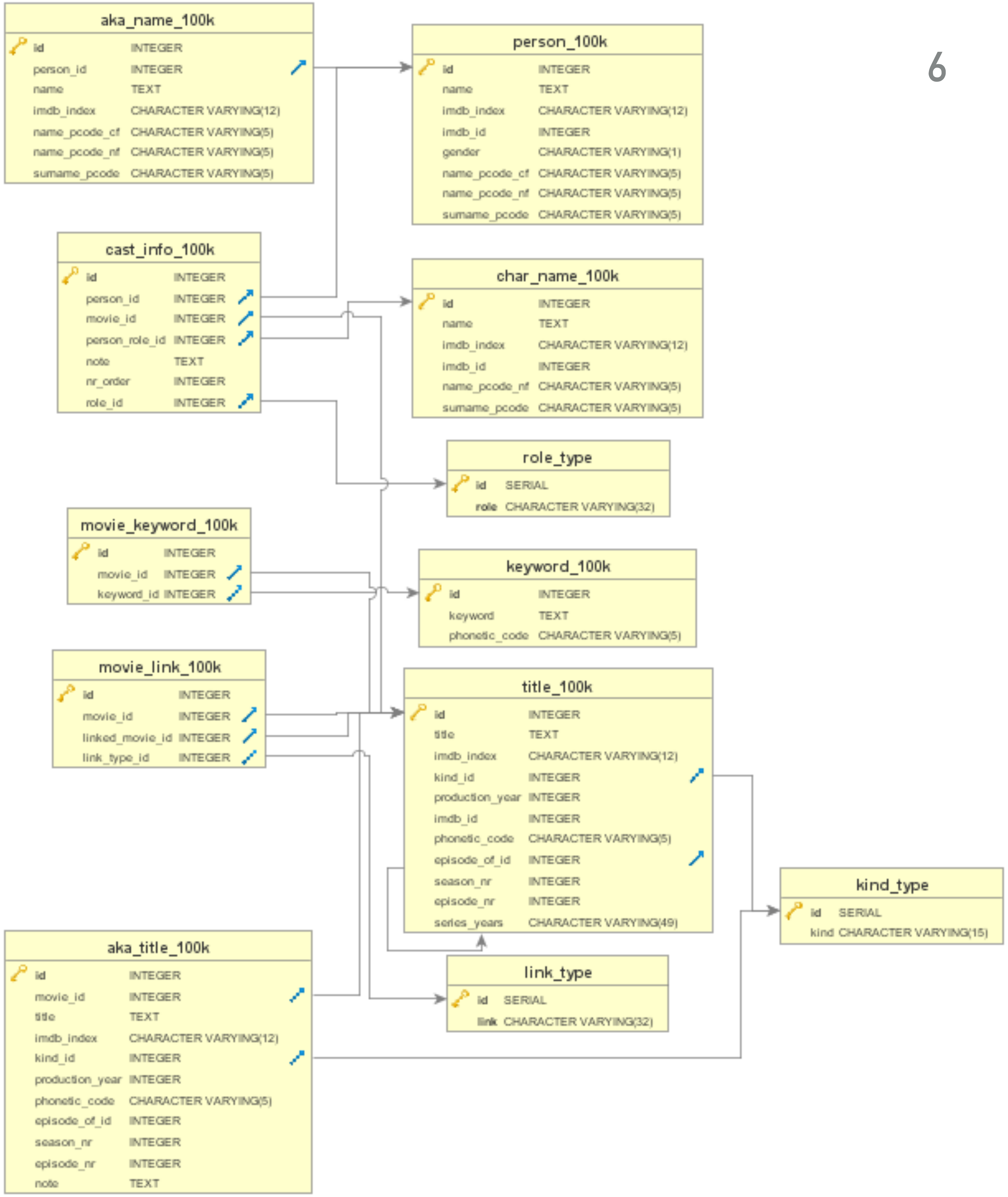
CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

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

EXAMPLE DB3: IMDB

- A subset of the schema and data from the IMDB.com website
  - Actors (person_100k), Movies (title_100k), and Actors in Movies (cast_info_100k)
  - Plus aliases, keywords, movie genres, etc.
- We will use MongoDB and Neo4J implementations of the same database (obviously, with different schemas)
- Get it (with import instructions) here
  - https://docs.google.com/document/d/1jj3cMAnk6Rc0mHkkOAIYDzYLjKisCuyj4-3KF9l-_8o



AGGREGATE QUERIES

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 or distinct values

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

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

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

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

```
SELECT COUNT (*)
 FROM  Supply
```

- Is it right?
- Equivalent to SELECT COUNT(CodeP) or SELECT COUNT(CodeS)

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 /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

- **Count** the number of suppliers that supply the product "P2"

```
SELECT COUNT(*)
 FROM  Supply
 WHERE CodeP = 'P2'
```

- Is it right?

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

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 to force a value for `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, although syntactically admissible

```
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

GROUPING ROWS

- Queries may apply aggregate operators to subsets of 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 /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

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 DeptName,D.City,COUNT(*)
 FROM Employee E JOIN Department D ON E.Dept=D.DeptName
 GROUP BY DeptName
```

- Incorrect Query

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

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 DeptName,D.City,COUNT(*)
 FROM  Employee E JOIN Department D ON E.Dept=D.DeptName
 GROUP BY DeptName, D.City
```

- Correct Query

GROUPING ROWS

- Queries may apply aggregate operators to subsets of 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				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

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
```

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

NESTED QUERIES

SUBQUERIES

- A parenthesised `SELECT-FROM-WHERE` statement can be used as a value in a number of places, including `FROM`, `WHERE`, and `HAVING` clauses
  - *subquery* or *nested* query
  - Example: in place of a table in the `FROM` clause, we can have another query, and then assert a condition on its results
- A common use of subqueries is to perform tests for **set membership**, **set comparisons**, and **set cardinality**
- The use of nested queries may produce less declarative queries, but they often improve readability
  - Complex queries can become very difficult to understand

THE IN OPERATOR

- `<tuple> IN <relation>` is true if and only if the tuple is a member of the relations
    - `<tuple> NOT IN <relation>` means the opposite
- `IN` expression can appear in `WHERE` clauses
    - The relation is often a subquery
        - e.g. `WHERE firstName IN (SELECT lastName FROM Employee)`
    - It is possible to specify a pre-defined list of values
        - e.g. `WHERE firstName IN ('Jan','Arie','Henk')`
- It allows **problem decomposition**, typically with a "bottom-up" approach

EXAMPLE /1

- Find the *name* of **all** the suppliers **of** product ”P2”

TWO SUBPROBLEMS
- CODE OF P2 SUPPLIERS
- NAME OF SUCH SUPPLIERS

WITH JOIN

```
SELECT NameS
 FROM  Supplier, Supply
 WHERE Supplier.CodeS = Supply.CodeS AND CodeP = ‘P2'
```

WITH IN AND NESTED QUERIES

```
SELECT NameS
 FROM  Supplier
 WHERE CodeS IN (SELECT CodeS
                 FROM  Supply
                 WHERE CodeP = ‘P2’)
```

EXAMPLE /2

- Find the *name* of supplier of **at least one red product**

THREE SUBPROBLEMS
- CODE OF RED PRODUCTS
- CODE OF SUPPLIERS THAT SUPPLY SUCH PRODUCTS
- NAME OF SUCH SUPPLIERS

WITH JOIN

```
SELECT NameS
 FROM  Supplier, Supply, Products
 WHERE Supplier.CodeS = Supply.CodeS AND Supply.CodeP = Products.CodePCodeP
       AND Color = ‘Red'
```

WITH IN AND NESTED QUERIES

```
SELECT NameS
 FROM  Supplier
 WHERE CodeS IN (SELECT CodeS
                  FROM  Supply
                  WHERE CodeP IN (SELECT CodeP
                                   FROM  Products
                                   WHERE Color = ‘Red’))
```

NESTED QUERIES THAT RETURN ONE TUPLE

- If a subquery is guaranteed to produce one tuple, then the result of the subquery can be used as a **value**
  - Typically, a single tuple is guaranteed by key constraints of attributes `SELECT`ed by the subquery
- A run-time error occurs if there is no tuple or more than one tuple
- Usually, the tuple has one attribute, but with a tuple constructor we might have many => **row subquery**

EXAMPLE /1

- Find the *code* of suppliers having office in the same city as S1

WITH JOIN

```
SELECT Su2.CodeS
 FROM  Supplier AS Su1, Supplier AS Su2
 WHERE Su1.Office = Su2.Office AND Su1.CodeS = ‘S1'
```

WITH NESTED QUERIES

```
SELECT CodeS
 FROM Supplier
 WHERE Office = (SELECT Office
                 FROM  Supplier
                 WHERE CodeS = ‘S1’)
```

EXAMPLE /2

- Find the code of suppliers with less shareholders than the supplier having the **maximum** number of shareholders

WITH NESTED QUERIES

```
SELECT CodeS
 FROM Supplier
 WHERE Shareholders < (SELECT MAX(Shareholders)
                        FROM  Supplier)
```

WITH JOIN??

MAYBE. THINK ABOUT IT :)

EXAMPLE /3

- Find the name of the suppliers that supply at least one product supplied by suppliers of red products

Difficult to express with joins

- Code of red products
- Code of suppliers that supply such products
- Code of products supplied by suppliers of red products
- Code of the suppliers of the products supplied by suppliers of red products
- Name of such suppliers

```
SELECT NameS
 FROM Supplier
 WHERE CodeS IN (SELECT CodeS
                 FROM  Supply
                 WHERE CodeP IN (SELECT CodeP
                                 FROM  Supply
                                 WHERE CodeS IN (SELECT CodeS
                                                 FROM  Supply
                                                 WHERE CodeP IN (SELECT CodeP
                                                                 FROM  Product
                                                                 WHERE Color = ‘Red’))))
```

EXAMPLE /3 WITH JOIN

- Find the name of the suppliers that supply at least one product supplied by suppliers of red products

```
SELECT NameS
 FROM  Supplier, Supply AS SA, Supply AS SB, Supply AS SC, Products
 WHERE Supplier.CodeS = SA.CodeS AND
       SA.CodeP = SB.CodeP AND
       SB.CodeS = SC.CodeS AND
       SC.CodeP = Products.CodeP AND
       Products.Color = ‘Red’
```

EXAMPLE WITH NOT IN

- Find the name of the suppliers that **DO NOT** supply product P2

- Can you express it with JOIN?

```
SELECT NameS
 FROM  Supplier, Supply
 WHERE Supplier.CodeS = Supply.CodeS AND CodeP <> ‘P2’
```

EXAMPLE WITH NOT IN

- Find the name of the suppliers that **DO NOT** supply product P2

- Can you express it with JOIN?

```
SELECT NameS
 FROM  Supplier, Supply
 WHERE Supplier.CodeS = Supply.CodeS AND CodeP <> 'P2'
```

- **WRONG!!!**

- The SQL above would answer the query:

- Find the name of the suppliers that supply **at least one product different from** P2

EXAMPLE WITH NOT IN/2

- Find the name of the suppliers that **DO NOT** supply product P2

- We need to exclude from the result set suppliers that supply **P2**

```
SELECT NameS
 FROM  Supplier
 WHERE CodeS NOT IN (SELECT CodeS
                     FROM  Supply
                     WHERE CodeP = ‘P2’)
```

EXAMPLE WITH NOT IN /3

- Find the name of the suppliers that supply **ONLY** product P2
- We need to exclude from the result set suppliers that supply products different from P2
- But only keep the ones that actually supply something

```
SELECT NameS
 FROM  Supplier
 WHERE CodeS NOT IN (SELECT CodeS
                     FROM  Supply
                     WHERE CodeP <>'P2')
            AND CodeS IN (SELECT CodeS
                          FROM  Supply)
```

EXAMPLE WITH NOT IN /4

- Find the name of the suppliers that **DO NOT** supply red products
- We need to exclude from the result set suppliers of red products

```
SELECT NameS
 FROM  Supplier
 WHERE CodeS NOT IN (SELECT CodeS
                     FROM  Supply
                     WHERE CodeP IN (SELECT CodeP
                                     FROM  Products
                                     WHERE Color = 'Red'))
```

ON QUANTIFIERS

- Existential Quantifiers: **Easy! :)**
- Find the name of the suppliers that supply **at least one product different from** P2

- Universal Quantifiers: **Hard! :(**
- Find the name of the suppliers that **DO NOT** supply product P2
- Find the name of the suppliers that supply **ONLY** product P2
- Find the name of the suppliers that **DO NOT** supply red products

SCOPE OF ATTRIBUTES, VARIABLES, AND CORRELATED QUERIES

- A subquery can use attributes and/or variables defined by outermost queries in their `WHERE` clause
  - this is sometimes referred to as "*transfer of bindings*"
- The two queries are said to be correlated
- Semantics: the nested query is evaluated **for each row of the external query**

THE EXISTS OPERATOR/1

- Find the name of the suppliers that supplied P2 at least once

```
SELECT NameS
 FROM  Supplier
 WHERE EXISTS (SELECT *
                FROM Supply
                WHERE CodeP='P2' AND Supplier.CodeS = Supply.CodeS)
```

- !!! We need to test the existence of a supply for the evaluation of suppliers
- `Suppliers.CodeS = Supply.CodeS` imposes a correlation between external and internal query

THE EXISTS OPERATOR /2

- Find all the homonyms

- i.e., people with the same first name and surname, but different BSN

```
SELECT *
 FROM  Person P1
 WHERE EXISTS (SELECT *
               FROM  Person P2
               WHERE P1.FirstName = P2.FirstName AND
                     P1.Surname = P2.Surname AND
                     P1.BSN <> P2.BSN)
```

LIMITATIONS

- A query cannot refer to attributes in a subquery, or in a query at the same level of nesting

```
SELECT *
 FROM  Employee
 WHERE Dept IN (SELECT DeptName
                FROM   Department D1
                WHERE DeptName = ‘Production’) OR
       Dept IN (SELECT DeptName
                FROM   Department D2
                WHERE D2.City = D1.City)
```

- The query is incorrect because variable D1 is not visible in the second nested query

EXAMPLE OF NOT EXISTS OPERATOR /1

- Find the name of the suppliers that **DO NOT** supply P2 or
- Find the name of the suppliers for whom it **does not exist at least one** supply of P2

```
SELECT NameS
 FROM  Supplier
 WHERE NOT EXISTS (SELECT *
                   FROM  Supply
                   WHERE CodeP = ‘P2’ AND
                         Supplier.CodeS = Supply.CodeS)
```

EXAMPLE OF NOT EXISTS OPERATOR /2

- Find all the persons who **do not have** homonyms

```
SELECT *
 FROM  Person P1
 WHERE NOT EXISTS (SELECT *
                   FROM  Person P2
                   WHERE P1.FirstName = P2.FirstName AND
                         P1.Surname = P2.Surname AND
                         P1.BSN <> P2.BSN)
```

TUPLE CONSTRUCTOR

- The comparison with the nested query may involve more than one attributes
- The attributes must be enclosed within a pair of curved brackets (tuple constructor)
- The query in the previous slide can be expressed as:

```
SELECT *
 FROM  Person P1
 WHERE (FirstName,Surname) NOT IN (SELECT FirstName, Surname
                                   FROM  Person P2
                                   WHERE P1.BSN <> P2.BSN)
```

ANY OPERATOR /1

- Find the employees who work in departments in London

```
SELECT FirstName, Surname
 FROM  Employee
 WHERE Dept = ANY (SELECT DeptName
                   FROM  Department
                   WHERE City = ‘London’)
```

- But also

```
SELECT FirstName, Surname
 FROM  Employee, Department
 WHERE Dept = DeptName AND Department.City = ‘London’
```

ANY OPERATOR /2

- Find the employees of the Planning department, having the same first name as a member of the Production department

```
SELECT FirstName, Surname
 FROM  Employee
 WHERE Dept = ‘Planning’ AND FirstName = ANY (SELECT FirstName
                                               FROM  Employee
                                               WHERE Dept = ‘Production’)
```

- But also

```
SELECT E1.FirstName, E1.Surname
 FROM  Employee E1, Employee E2
 WHERE E1.FirstName = E2.FirstName AND E2.Dept = ‘Production’ AND E1.Dept = ‘Planning’
```

EXAMPLE MIN AND MAX

- Queries using the aggregate operators max and min can be expressed with nested queries

```
SELECT Dept
 FROM  Employee
 WHERE Salary >= ALL (SELECT Salary
                      FROM   Employee)
```

- But also

```
SELECT Dept
 FROM  Employee
 WHERE Salary IN (SELECT MAX(Salary)
                  FROM   Employee)
```

EXAMPLE OF NEGATION WITH ALL

- Find the departments in which there is **no one** named Brown

```
SELECT DeptName
 FROM  Department
 WHERE DeptName <> ALL (SELECT Dept
                         FROM  Employee
                         WHERE Surname='Brown')
```

- Alternatively (more next)

```
 SELECT DeptName
  FROM  Department
EXCEPT
 SELECT Dept
  FROM  Employee
  WHERE Surname='Brown'
```

NESTED AND AGGREGATED QUERIES /1

- Find the code of the suppliers that supply ALL products

```
SELECT CodeS
 FROM  Supply
 GROUP BY CodeS
 HAVING COUNT(*) = (SELECT COUNT(*)
                     FROM  Products)
```

- All the products that can be supplied are in the Product table
- A supplier supply all the products if the number of distinct supplied product is the same as the number of available products

NESTED AND AGGREGATED QUERIES /2

- Find the code of the suppliers that supplied **as much as** S2

```
SELECT CodeS
 FROM  Supply
 WHERE CodeP IN (SELECT CodeP
                 FROM  Supply
                 WHERE CodeS = ‘S2’)
 GROUP BY CodeS
 HAVING COUNT(*) = (SELECT COUNT(*)
                    FROM  Supply
                    WHERE CodeS = ‘S2’)
```

- The number of supplies of a supplier should be equal to the number of supplies by S2

SET QUERIES

SET QUERIES

- Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:
  - `(subquery)INTERSECT[ALL](subquery)`
  - `(subquery)EXCEPT[ALL](subquery)`

CAN BE EXPRESSED WITH OTHER OPERATORS (TYPICALLY SUB-QUERIES)

  - `(subquery)UNION[ALL](subquery)`

ENHANCEMENT OF THE EXPRESSIVE POWER

SET SEMANTIC OF SET QUERIES

- Although the `SELECT-FROM-WHERE` statement uses **bag semantics**, the *default* for union, intersection, and difference is **set semantics**.
  - That is, duplicates are eliminated as the operation is applied

- Motivation: Efficiency
  - When projecting attributes, it is easier to avoid **eliminating duplicates**. Just work tuple-at-a-time.
  - When doing intersection or difference, it is most efficient to **sort the relations first** At that point you may as well eliminate the duplicates anyway

UNION

- A single `SELECT` cannot represent unions of values from two or more tables

```
A UNION [ALL] B
```

- It executes the union of two relational expressions
  - Expressions generated by `SELECT` clauses
  - Table A and Table B must be **union compatible**
    - i.e. have compatible schema
    - same number of output fields, in the same order, and with the same or compatible data types
- Duplicate removal
  - `UNION` removes duplicates
  - `UNION ALL` does not remove duplicates

UNION EXAMPLE /1

- Find the code of red products **OR** products supplied by S2 (or both)

```
 SELECT CodeP
  FROM  Products
  WHERE Color = 'Red'

UNION

 SELECT CodeP
  FROM Supply
  WHERE CodeS = 'S2'
```

UNION EXAMPLE /1

- Find the code of red products **OR** products supplied by S2 (or both)

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

CodeP
P1
P6

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

CodeP
P1
P2

CodeP
P1
P2
P6

UNION EXAMPLE /2

- Find the first names and surnames of the employees

```
 SELECT FirstName AS Name
  FROM  Employee

UNION

 SELECT Surname
  FROM Employee
```

INTERSECTION

```
A INTERSECT [ALL] B
```

- Intersection of two subqueries
  - returns all rows that are both in the result of A and in the result of B
- As for the `UNION` operator, schema must be **union compatible**
- Duplicate rows are eliminated unless INTERSECT ALL is used.
- Not supported by all RDBMS (e.g. not supported by MySQL)

INTERSECT EXAMPLE

- Find cities hosting both suppliers’ offices and products’ storehouses

```
 SELECT Office
  FROM  Supplier

INTERSECT

 SELECT Storehouse
  FROM  Product
```

- Equivalent to

```
SELECT DISTINCT Office
 FROM  Supplier, Product
 WHERE Office = Storehouse
```

INTERSECT EXAMPLE /2

- Find the surnames of employees that are also first names

```
 SELECT FirstName
  FROM  Employee

INTERSECT

 SELECT Surname
  FROM Employee
```

- Equivalent to

```
SELECT DISTINCT E1.FirstName
 FROM  Employee E1, Employee E2
 WHERE E1.FirstName = E2.Surname
```

EXCEPT

```
A EXCEPT [ALL] B
```

- Difference set operator
  - Returns all rows that are in the result of A but not in the result of B
- As for the UNION operator, schema must be **union compatible**
- Not supported by all RDBMS (e.g. not supported by MySQL)

EXCEPT EXAMPLE

- Find cities hosting suppliers' offices, but not products' storehouses

```
 SELECT Office
  FROM  Supplier

EXCEPT

 SELECT Storehouse
  FROM Products
```

- Can be represented with a nested query using the NOT IN operator

```
SELECT DISTINCT Office
 FROM  Supplier
 WHERE Office NOT IN (SELECT Storehouse
                     FROM Products)
```

EXCEPT EXAMPLE /2

- Find the surnames of employees that are not also first names

```
 SELECT FirstName
  FROM  Employee

EXCEPT

 SELECT Surname
  FROM Employee
```

SCHEMA DEFINITION

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 authorisation)
- Many other kinds of elements may also appear in the database schema, including:
  - privileges, views, indexes, triggers
- Syntax:

```
CREATE SCHEMA [ SchemaName ]
[[ authorisation ] Authorisation ]
{ SchemaElementDefinition }
```

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)/1

- Bit
  - Single boolean values or strings of boolean values (may be variable in length)
  - Syntax: `BIT [varying] [(Length)]`
- Exact numeric domains
  - Exact values, integer or with a fractional part
  - Four Alternatives
    - `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`
    - `INTEGER`: a finite subset of the integers that is machine-dependent
    - `SMALLINT`: a machine-dependent subset of the integer domain type

ELEMENTARY DOMAINS (DATA TYPES)/2

- **Approximate real values**
  - Based on floating point representation
  - `FLOAT [(Precision)]`: floating point number, with user-specified precision of at least `n` digits. By default `n` is 53, but it can be less
  - `REAL`: floating point numbers, with machine-dependent precision
  - `Double Precision`: double-precision floating point numbers, with machine-dependent precision

ELEMENTARY DOMAINS (DATA TYPES)/3

- 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)/4

- 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)

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
- OtherConstraints: integrity constraints on the table
- Syntax:

```
CREATE TABLE TableName (
  AttributeName Domain [DefaultValue] [Constraints]
  {, AttributeName Domain [DefaultValue] [Constraints]}
  [OtherConstraints]
)
```

EXAMPLE OF CREATE TABLE

```
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)
)
```

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 >
```

- `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

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 centralised verification
- Disadvantages
  - might slow down execution
  - pre-defined type of constraints
    - e.g. no constraint on aggregated data
    - but triggers can help

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**)
  - *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`
  - syntax like `UNIQUE`

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.
  - But attributes declared `UNIQUE` may have `NULL`s
    - <u>and there may be several tuples with `NULL`!</u>

EXAMPLE OF INTRA-RELATIONAL 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
```

INTER-RELATIONAL CONSTRAINTS

- Constraints may take into account several relations
- `REFERENCES` and `FOREIGN KEY` 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)`
- **It is possible to associate reaction policies to violations of referential integrity**

REACTION POLICIES FOR REFERENTIAL INTEGRITY CONSTRAINTS

- Reactions operate **on** the referencing table, after changes **to** the referenced table
- Violations may be introduced
  - by updates on the referred attribute
  - by row deletions
- 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`: forbid the change on the external table
- Syntax:

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

EXAMPLE OF INTER-RELATIONAL CONSTRAINT

```
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)
)
```

Referencing

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

Referenced

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

SCHEMA UPDATES

- Two 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
```

RELATIONAL CATALOGUES

- The catalog contains:
  - The data dictionary
  - The description of the data contained in the data base (tables, etc.)
  - Statistics about the data (distribution, access, growth)
- It is based on a relational structure (reflexive)
- It can be queried!
- The SQL92 standard describes a `Definition Schema` (composed of tables) and an `Information Schema` (composed of views)

```
SELECT table_name
FROM information_schema.tables
WHERE table_schema = 'public'
```

DATA MANIPULATION

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

```
INSERT INTO TableName [(AttributeList)] <VALUES(ListofValues)|SELECT SQL>
```

- Using Values

```
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 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 KEY`s 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

```
UPDATE TableName
   SET Attribute = <Expression | SELECT SQL | NULL | default>
   {, Attribute = <Expression | SELECT SQL | NULL | default>}
   [WHERE Condition]
```

- Examples

```
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

WRAPPING UP

TODAY WE COVERED

- SQL as
  - a Retrieval Language
  - a Schema Creation and Modification Language
  - a Data Manipulation Language

END OF LECTURE