Databases

Lecture 4

Querying Relational Databases Using SQL (II)

Functional Dependencies. Normal Forms

Find the average age of researchers who are at least 18 years old for each impact factor with at least 10 researchers.

```
SELECT R.ImpactFactor, AVG(R.Age) AS AvgAge
FROM Researchers R
WHERE R.Age >= 18
GROUP BY R.ImpactFactor
HAVING 9 < (SELECT COUNT(*)
           FROM Researchers R2
           WHERE R2.ImpactFactor = R.ImpactFactor)
```

Problem

See lecture query: *Find the most prolific researchers, i.e., those with the largest number of papers*, on schema *AuthorContribution(RID, PID, Year).*

Does it compute the same result on *AuthorContribution(RID, PID, Year)*? If not, change it so the intended result is computed on this schema as well.

Find the name and age of the oldest researcher.

```
SELECT R.Name, MAX(R.Age)
FROM Researchers R
```

--error: if the SELECT clause contains an aggregation operator, then it must contain **only** aggregation operators, unless the query has a GROUP BY clause

Correct query:

```
SELECT R.Name, R.Age
FROM Researchers R
WHERE R.Age = (SELECT MAX(R2.Age)
               FROM Researchers R2)
```

- ORDER BY, TOP

Sort researchers by impact factor (in descending order) and age (in ascending order).

```
SELECT *
FROM Researchers R
ORDER BY R.ImpactFactor DESC, R.Age ASC
```

Retrieve the names and ages of the top 10 researchers ordered by name.

```
SELECT TOP 10 R.Name, R.Age
FROM Researchers R
ORDER BY R.Name
```

Find the top 25% researchers (all the data) ordered by age (descending).

```
SELECT TOP 25 PERCENT *
FROM Researchers R
ORDER BY R.Age DESC
```

Find the number of researchers for each impact factor. Order the result by the number of researchers.

```
SELECT R.ImpactFactor, COUNT(*) AS NoR
FROM Researchers R
GROUP BY R.ImpactFactor
ORDER BY NoR
```

- obs. intersection queries can be expressed with IN.

Find the names of researchers who have published in IDEAS and EDBT.

```
SELECT R.Name
FROM Researchers R INNER JOIN AuthorContribution A
    ON R.RID = A.RID
  INNER JOIN Papers P ON A.PID = P.PID
WHERE P.Conference = 'IDEAS' AND
  R.RID IN (SELECT A2.RID
           FROM AuthorContribution A2 INNER JOIN Papers P2
             ON A2.PID = P2.PID
           WHERE P2.Conference = 'EDBT')
```

- obs. set-difference queries can be expressed with NOT IN.

Find the researchers who have published in EDBT but not in IDEAS.

```
SELECT A.RID
FROM AuthorContribution A INNER JOIN Papers P ON A.PID = P.PID
WHERE P.Conference = 'EDBT' AND
  A.RID NOT IN (SELECT A2.RID
               FROM AuthorContribution A2 INNER JOIN Papers P2
                 ON A2.PID = P2.PID
               WHERE P2.Conference = 'IDEAS')
```

Find researchers whose IF is greater than the IF of some researcher called *Ionescu*.

```
SELECT R.RID
FROM Researchers R
WHERE R.ImpactFactor >
  (SELECT MIN(R2.ImpactFactor)
  FROM Researchers R2
  WHERE R2.Name = 'Ionescu')
```

Find researchers whose IF is greater than the IF of every researcher called *Ionescu.*

```
SELECT R.RID
FROM Researchers R
WHERE R.ImpactFactor >
  (SELECT MAX(R2.ImpactFactor)
  FROM Researchers R2
  WHERE R2.Name = 'Ionescu')
```

- the SELECT statement:

$$\text{SELECT } \left[\begin{Bmatrix}\text{ALL}\\ \text{DISTINCT}\\ \text{TOP n [PERCENT]}\end{Bmatrix}\right] \begin{Bmatrix} * \\ \text{expr1 [AS column1] [, expr2 [AS column2]] ...}\end{Bmatrix}$$

FROM source1 [alias1] [, source2 [alias2]] ...

[WHERE qualification]

[GROUP BY grouping_list] [redacted]

[HAVING group_qualification]

$$\left[\begin{Bmatrix}\text{UNION [ALL]}\\ \text{INTERSECT}\\ \text{EXCEPT}\end{Bmatrix} \text{ SELECT\_statement}\right]$$

$$\left[\text{ORDER BY } \begin{Bmatrix}\text{column1}\\ \text{column1\_number}\end{Bmatrix} \left[\begin{Bmatrix}\text{ASC}\\ \text{DESC}\end{Bmatrix}\right] \left[, \begin{Bmatrix}\text{column2}\\ \text{column2\_number}\end{Bmatrix} \left[\begin{Bmatrix}\text{ASC}\\ \text{DESC}\end{Bmatrix}\right]\right] \ldots\right]$$

- non-procedural query
- SELECT statement evaluation: the result is a relation (table)
- data can be obtained from one or multiple data sources; a source can have an associated *alias*, used only in the SELECT statement
- various expressions are evaluated on the data (from the above-mentioned sources)
- a source column can be qualified with the source's name (or alias)

- a data source can be:
  1. table / view in the database
  2. (SELECT_statement)
  3. join_expression:
     - source1 [alias1] join_operator source2 [alias2] ON join_condition
     - (join_expression)

  * a join condition can be of the form:
     - elementary_cond
     - (condition)
     - NOT condition
     - condition1 AND condition2
     - condition1 OR condition2

* an elementary join condition (elementary_cond) among two data sources can be of the form:

- [source1_alias.]column1 relational_operator [source2_alias.]column2
- expression1 relational_operator expression2 (expression1 and expression2 use columns from different sources)

- the WHERE clause can contain filter and join conditions
- filter conditions:
  - expression relational_operator expression
  - expression [NOT] BETWEEN valmin AND valmax
  - expression [NOT] LIKE pattern
  - expression IS [NOT] NULL
  - expression [NOT] IN (value [, value] ...)
  - expression [NOT] IN (subquery)
  - expression relational_operator {ALL | ANY} (subquery)
  - [NOT] EXISTS (subquery)
- filter conditions can be:
  - elementary (described above)
  - composed with logical operators and parentheses

- obs: not all DBMSs support TOP
  - MySQL: SELECT ... LIMIT n
  - Oracle: SELECT ... WHERE ROWNUM <= n

- rules for building expressions:
  - operands: constants, columns, system functions, user functions
  - operators: corresponding to operands

- ordering records: the ORDER BY clause

- the SELECT statement - logical processing (Transact-SQL)

FROM

WHERE

GROUP BY

HAVING

SELECT

DISTINCT

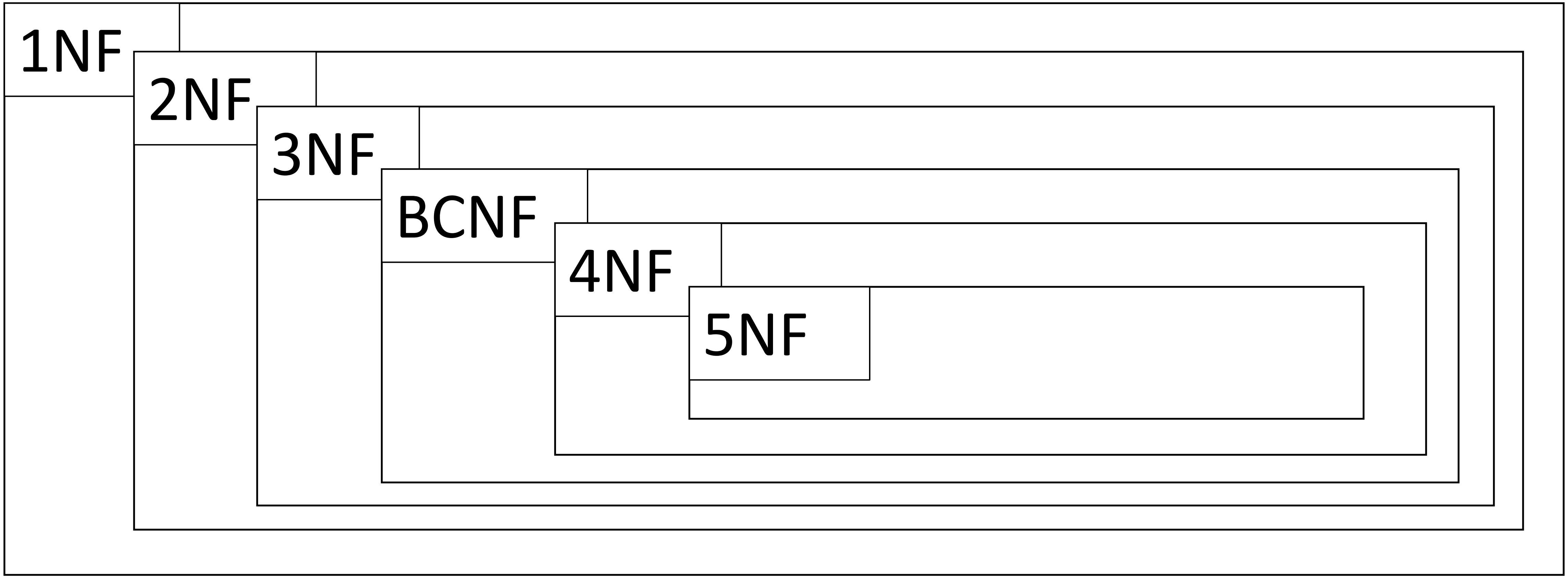
ORDER BY

TOP

Functional Dependencies. Normal Forms

- there are multiple ways to model a data collection in the relational model (using tables)
- database design should be carried out in such a manner that subsequent queries and operations are performed as smoothly as possible, i.e.:
  - no additional tests are required when changing data
  - operations can be performed through SQL statements alone
- the above mentioned operations can be easily carried out when relations satisfy certain conditions (i.e., relations are in a certain normal form)

- the most common normal forms: 1NF, 2NF, 3NF, BCNF, 4NF, 5NF
- they were defined by Codd (the first 3), Boyce and Codd (BCNF), and Fagin (4NF, 5NF)
- the following inclusions hold (for relations that satisfy various normal forms):



data redundancy

- generates a series of problems; some can be tackled by replacing a relation with a collection of smaller relations; each of the latter contains a (strict) subset of attributes from the original relation
- such decompositions can generate new problems (e.g., some data could be lost)
- ideally, only redundancy-free schemas should be allowed
- one should at least be able to identify schemas with redundancy, even if such schemas are allowed (e.g., for performance reasons)

- if a relation is not in normal form X, it can be decomposed into multiple relations that are in normal form X

Definition. The projection operator is used to decompose a relation. Let $R[A_1, A_2, \dots, A_n]$ be a relation and $\alpha = \left\{A_{i_1}, A_{i_2}, \dots, A_{i_p}\right\}$ a subset of attributes, $\alpha \subset \{A_1, A_2, \dots, A_n\}$. The projection of relation R on α is:

$$R'\left[A_{i_1}, A_{i_2}, \dots, A_{i_p}\right] = \Pi_\alpha(R) = \Pi_{\left\{A_{i_1}, A_{i_2}, \dots, A_{i_p}\right\}}(R) = \{r[\alpha] | r \in R\}$$

where $\forall r = (a_1, a_2, \dots, a_n) \in R \Rightarrow \Pi_\alpha(r) = r[\alpha] = \left(a_{i_1}, a_{i_2}, \dots, a_{i_p}\right) \in R'$,

and all elements in R' are distinct.

Definition. The natural join operator is used to compose relations. Let *R[α, β]*, *S[β, γ]* be two relations over the specified sets of attributes, $\alpha \cap \gamma = \emptyset$. The natural join of relations *R* and *S* is the relation:

$$R * S[\alpha, \beta, \gamma] = \left\{\left(\Pi_\alpha(r), \Pi_\beta(r), \Pi_\gamma(s)\right) \middle| r \in R, s \in S\ and\ \Pi_\beta(r) = \Pi_\beta(s)\right\}$$

- a relation R can be decomposed into multiple new relations $R_1, R_2, \ldots, R_m$; the decomposition is good if $R = R_1 * R_2 * \cdots * R_m$, i.e., R's data can be obtained from the data stored in relations $R_1, R_2, \ldots, R_m$ (no data is added / lost through decomposition / composition)

- e.g., a decomposition that is not good:

Example 1. LearningContracts[Student, FacultyMember, Course],

and two new relations obtained by applying the projection operator to this relation: SF[Student, FacultyMember] and FC[FacultyMember, Course]

  - assume the original relation contains the following values:

LearningContracts	Student	FacultyMember	Course
r1	s1	f1	c1
r2	s2	f2	c2
r3	s1	f2	c3

- using the Π and * operators, the following values are obtained for the 2 projections and their natural join:

SF	Student	FacultyMember
r1	s1	f1
r2	s2	f2
r3	s1	f2

FC	FacultyMember	Course
r1	f1	c1
r2	f2	c2
r3	f2	c3

SF*FC	Student	FacultyMember	Course
r1	s1	f1	c1
r2	s2	f2	c2
?	s2	f2	c3
?	s1	f2	c2
r3	s1	f2	c3

SF*FC	Student	FacultyMember	Course
r1	s1	f1	c1
r2	s2	f2	c2
?	s2	f2	c3
?	s1	f2	c2
r3	s1	f2	c3

- the decomposition is not good, since the SF*FC relation contains extra records, i.e., records that were not present in the original relation

- *simple attribute*
- *composite attribute* - a set of attributes in a relation (with at least two attributes)

- in some applications, attributes (simple or composite) can take on multiple values for a record in the relation - *repeating* attributes
- when defining a relation (in the relational model), attribute values must be scalar, atomic (they cannot be further decomposed)

Example 2. Consider the relation:

STUDENT [NAME, BIRTHYEAR, GROUP, COURSE, GRADE]

- key: NAME
- *composite repeating attribute:* the pair {COURSE, GRADE}
- this relation could store the following values:

NAME	BIRTHYEAR	GROUP	COURSE	GRADE
Ioana	2000	921	Databases Distributed Operating Systems Probabilities and Statistics	10 9.99 10
Vasile	2000	925	Databases Distributed Operating Systems Probabilities and Statistics Logical and Functional Programming	10 9.98 10 10

Example 3. Consider the relation:

BOOK [BId, AuthorsNames, Title, Publisher, PublishingYear, Keywords]

- key: BId
- *simple repeating attributes*: AuthorsNames, Keywords
- the BId attribute
  - could have an actual meaning (an id could be associated with each book)
  - could be introduced as a key (with distinct values that are automatically generated)

- repeating attributes cannot be used in the relational model, hence they are to be avoided, without losing data
- let $R[A]$ be a relation; A - the set of attributes
- let α be a repeating attribute in R (simple or composite)
- R can be decomposed into 2 relations, such that α is not a repeating attribute anymore
- if K is a key in R, the two relations into which R is decomposed are:

$$R'[K \cup \alpha] = \Pi_{K \cup \alpha}(R)$$
$$R''[A - \alpha] = \Pi_{A-\alpha}(R)$$

Example 4. The STUDENT relation from example 2 is decomposed into the following 2 relations:

GENERAL_DATA [NAME, BIRTHYEAR, GROUP],

RESULTS [NAME, COURSE, GRADE].

- the values in these relations are:

NAME	BIRTHYEAR	GROUP
Ioana	2000	921
Vasile	2000	925

NAME	COURSE	GRADE
Ioana	Databases	10
Ioana	Distributed Operating Systems	9.99
Ioana	Probabilities and Statistics	10
Vasile	Databases	10
Vasile	Distributed Operating Systems	9.98
Vasile	Probabilities and Statistics	10
Vasile	Logical and Functional Programming	10

Example 5. The BOOK relation in example 3 is decomposed into the following 3 relations (there are two repeating attributes in the BOOK relation):

BOOKS [BId, Title, Publisher, PublishingYear],

AUTHORS [BId, AuthorName],

KEYWORDS [BId, Keyword].

- is a book is not associated with any authors / keywords, it needs to have one corresponding tuple in AUTHORS / KEYWORDS, with the second attribute set to *null*; in the absence of such tuples, the BOOK relation can't be obtained from the three relations using just the natural join (outer join operators are required)

Definition. A relation is in the first normal form (1NF) if it doesn't have any repeating attributes.

- obs. some DBMSs can manage non-1NF relations (e.g., Oracle)

- the following 3 relational normal forms use a very important notion: the *functional dependency* among subsets of attributes
- the database administrator is responsible with determining the functional dependencies; these depend on the nature, the semantics of the data stored in the relation

->

Definition. Let $R[A_1, A_2, \dots, A_n]$ be a relation and α, β two subsets of attributes of R. The (simple or composite) attribute α functionally determines attribute β (simple or composite), notation:

$$\alpha \rightarrow \beta$$

if and only if every value of α in R is associated with a precise, unique value for β (this association holds throughout the entire existence of relation R); if an α value appears in multiple rows, each of these rows will contain the same value for β:

$$\Pi_\alpha(r) = \Pi_\alpha(r') \ \ implies \ \ \Pi_\beta(r) = \Pi_\beta(r')$$

- in the dependency $\alpha \rightarrow \beta$, α is the *determinant*, and β is the *dependent*
- the functional dependency can be regarded as a property (restriction) that must be satisfied by the database throughout its existence: values can be added / changed in the relation only if the functional dependency is satisfied

- if a relation contains a functional dependency, some associations among values will be stored multiple times (data redundancy)
- to describe some of the problems that can arise in this context, we'll consider the EXAM relation, storing students' final exam results

Example 6. EXAM [StudentName, Course, Grade, FacultyMember]

- key: {StudentName, Course}
- a course is associated with one faculty member, a faculty member can be associated with several courses ⇒ the following restriction (functional dependency) must be satisfied: *{Course}* → *{FacultyMember}*

EXAM	StudentName	Course	Grade	FacultyMember
1	Pop Ioana	Computer Networks	10	Matei Ana
2	Vlad Ana	Operating Systems	10	Simion Bogdan
3	Vlad Ana	Computer Networks	9.98	Matei Ana
4	Dan Andrei	Computer Networks	10	Matei Ana
5	Popescu Alex	Operating Systems	9.99	Simion Bogdan

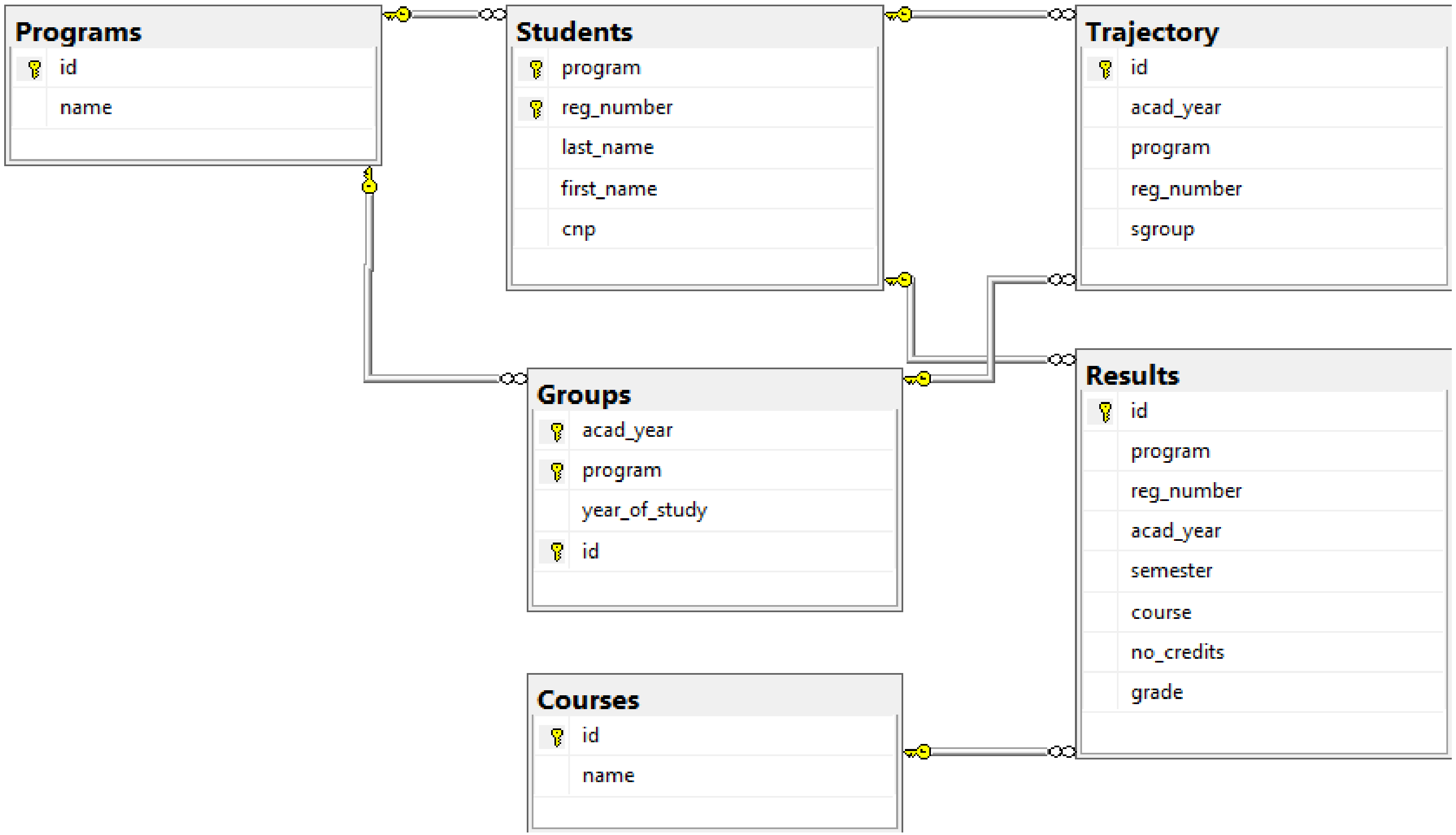
- if the relation contains such a functional dependency, the following problems can arise (some of them need to be solved through additional programming efforts - executing a SQL command is not enough):
  - wasting space: the same associations are stored multiple times, e.g., the one among *Computer Networks* and *Matei Ana* is stored 3 times;
  - update anomalies: if the *FacultyMember* is changed for course *c*, the change must be carried out in all associations involving course *c* (without knowing how many such associations exist), otherwise the database will contain errors (it will be inconsistent); if the *FacultyMember* value is changed in the 2nd record, but not in the 5th record, the operation will introduce an error in the relation (the data will be incorrect);
  - insertion anomalies: one cannot specify the *FacultyMember* for a course *c*, unless there's at least one student with a grade in course *c*;

- <u>deletion anomalies</u>: when some records are deleted, data that is not intended to be removed can be deleted as well; e.g., if records 1, 3 and 4 are deleted, the association among *Course* and *FacultyMember* is also removed from the database

- the functional dependency among sets of attributes is what causes the previous anomalies; to eliminate these problems, the associations (dependencies) among values should be kept once in a separate relation, therefore, the initial relation should be decomposed (via a good decomposition - data should not be lost or added); such a decomposition is performed in the database design phase, when functional dependencies can be identified

Appendix - More SQL Queries

- more query examples on the database:



Programs
- 🔑 id
- name

Students
- 🔑 program
- 🔑 reg_number
- last_name
- first_name
- cnp

Trajectory
- 🔑 id
- acad_year
- program
- reg_number
- sgroup

Groups
- 🔑 acad_year
- 🔑 program
- year_of_study
- 🔑 id

Results
- 🔑 id
- program
- reg_number
- acad_year
- semester
- course
- no_credits
- grade

Courses
- 🔑 id
- name

- A student's trajectory (academic year and group).

```
SELECT acad_year, sgroup
FROM Trajectory
WHERE program = 2 AND reg_number = '7654'
```

- A student's grades.

```
SELECT acad_year, course, no_credits, grade
FROM Results
WHERE program = 2 AND reg_number = '7654'
```

- Students who belonged to group 915 in the academic year 2017-2018.

```
SELECT last_name, first_name, s.reg_number
FROM Students s INNER JOIN Trajectory t
  ON s.program=t.program AND s.reg_number=t.reg_number
WHERE acad_year=2017 AND sgroup='915'
```

```
SELECT last_name, first_name, s.reg_number
FROM Students s INNER JOIN
  (SELECT *
  FROM Trajectory
  WHERE acad_year=2017 AND sgroup='915') t
ON s.reg_number=t.reg_number AND s.program=t.program
```

- Students who belong to a group, but have no grades - in the academic year 2017-2018.

```
SELECT last_name, first_name
FROM Students AS s
WHERE EXISTS (SELECT *
          FROM Trajectory t
          WHERE acad_year=2017 AND t.program=s.program
               AND t.reg_number=s.reg_number)
     AND NOT EXISTS (SELECT *
                    FROM Results r
                    WHERE acad_year=2017 AND
                         s.program=r.program AND
                         s.reg_number=r.reg_number)
--ORDER BY last_name, first_name
```

- Students who belong to a group, but have no grades - in the academic year 2017-2018.

```
SELECT last_name, first_name
FROM (Students s INNER JOIN
       (SELECT *
       FROM Trajectory WHERE acad_year=2017) t
       ON s.program=t.program AND s.reg_number=t.reg_number
     )
     LEFT JOIN
       (SELECT *
       FROM Results
       WHERE acad_year=2017) r
         ON s.program=r.program AND s.reg_number=r.reg_number
WHERE grade IS NULL
```

- The number of students in the database.

```
SELECT COUNT(*) AS NoS
FROM Students
```

- The number of students born on the same day, regardless of year and month.

```
SELECT SUBSTRING(cnp,6,2) AS day, COUNT(*) AS NoS
FROM Students
GROUP BY SUBSTRING(cnp,6,2)
```

- The grades of a given student (only the maximum grade is required for each course).

```
SELECT course, no_credits, MAX(grade) AS maxgrade
FROM Results
WHERE program = 3 AND reg_number='virtualstudent11'
GROUP BY course, no_credits
--ORDER BY course
```

- The grades of a given student (only the maximum grade is required for each course). Include the name of the course in the answer set.

```
SELECT id, name, no_credits, maxgrade AS grade
FROM Courses c INNER JOIN
  (SELECT course, no_credits, MAX(grade) AS maxgrade
  FROM Results
  WHERE program = 3 AND reg_number='virtualstudent11'
  GROUP BY course, no_credits) r
ON c.id = r.course
--ORDER BY name
```

- For each student name that appears at least 3 times, retrieve all students with that name.

```
SELECT *
FROM Students
WHERE last_name IN
     (SELECT last_name
     FROM Students
     GROUP BY last_name
     HAVING COUNT(*)>=3)
--ORDER BY last_name, first_name
```

* rewrite the query without IN

- The number of students in each program and year of study in 2018.

```
SELECT g.program, g.year_of_study, COUNT(*) AS NoS
FROM Trajectory t INNER JOIN Groups g
 ON t.program=g.program AND t.sgroup=g.id AND
   t.acad_year=g.acad_year
WHERE t.acad_year=2018
GROUP BY g.program, g.year_of_study
```

- The last name, first name, program (id), registration number, number of passed exams, number of credits, and gpa for each student with at least 30 credits in passed courses at the end of 2017.

```
SELECT last_name, first_name, s.program, s.reg_number, COUNT(*)
  AS noc, SUM(no_credits) AS nocr,
  SUM(gradem*no_credits)/SUM(no_credits) AS gpa
FROM Students s INNER JOIN
  (SELECT program, reg_number, course, no_credits, MAX(grade)
    AS gradem
  FROM Results
  WHERE acad_year=2017 AND grade>=5
  GROUP BY program, reg_number, course, no_credits
  ) r
  ON s.program=r.program AND s.reg_number=r.reg_number
GROUP BY s.program, s.reg_number, last_name, first_name
HAVING SUM(no_credits)>=30
ORDER BY 3,1,2
```

- For every course, the number of grades and the number of passing grades in 2017.

* a. MySQL *

```
SELECT name, COUNT(*) AS nrg, SUM(IF(r.grade >= 5,1,0))
AS nrpg
FROM Courses c INNER JOIN
  (SELECT * FROM Results WHERE acad_year = 2017) r
  ON c.id = r.course
GROUP BY name
--ORDER BY 1
```

- For every course, the number of grades and the number of passing grades in 2017.

* b. Oracle *

```
SELECT name, COUNT(*) AS nrg, SUM(CASE WHEN (r.grade >= 5) THEN
1 ELSE 0 END) AS nrpg
FROM Courses c INNER JOIN
  (SELECT * FROM Results WHERE acad_year = 2017) r
  ON c.id = r.course
GROUP BY name
--ORDER BY 1
```

- * The grade average for each group in 2017. *

a. Maximum & passing grades in 2017.

```
CREATE VIEW studgrades AS
SELECT program, reg_number, course, no_credits, MAX(grade) AS
 mgrade
FROM Results
WHERE acad_year=2017 AND grade>=5
GROUP BY program, reg_number, course, no_credits
```

b. Students' grade average (in 2017).

```
CREATE VIEW studavg AS
SELECT program, reg_number,
  SUM(no_credits*mgrade)/SUM(no_credits) AS gradeavg
FROM studgrades
GROUP BY program, reg_number
HAVING SUM(no_credits) >= 30
```

- * The grade average for each group in 2017. *

c. Final query:

```
SELECT sgroup, AVG(gradeavg) AS gravg
FROM
  (SELECT program, reg_number, sgroup
  FROM Trajectory
  WHERE acad_year=2017) t
    INNER JOIN studavg AS s
      ON t.reg_number=s.reg_number AND t.program=s.program
GROUP BY sgroup
```

- what happens if two groups in different programs have the same id?

- * The grade average for each program in 2017. *

```
CREATE VIEW programsavg AS
SELECT id, name, AVG(gradeavg) AS pravg
FROM Programs p INNER JOIN studavg s ON p.id=s.program
GROUP BY id, name

SELECT * FROM programsavg
```

References

- [Ta13] ȚÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2nd Edition), McGraw-Hill, 2000
- [Da03] DATE, C.J., An Introduction to Database Systems (8th Edition), Addison-Wesley, 2003
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ha96] HANSEN, G., HANSEN, J., Database Management And Design (2nd Edition), Prentice Hall, 1996
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, http://codex.cs.yale.edu/avi/db-book/
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems, http://infolab.stanford.edu/~ullman/fcdb.html