Databases Re-exam

TDA357 (Chalmers), DIT620 (University of Gothenburg)

30 August 2018 at 14:00 in SB Multisal

Department of Computer Science and Engineering

Examiner: Aarne Ranta tel. 031 772 10 82 email aarne at chalmers.se

Results: Will be published by 20 September.

Exam review: upon agreement with the examiner

Grades: Chalmers: 24 for 3, 36 for 4, 48 for 5. GU: 24 for G, 42 for VG.

Help material: One "cheat sheet", which is an A4 sheet with hand-written notes. You may write on both sides of that sheet. If you bring a sheet, it must be handed in with your answers to the exam questions. One English language dictionary is also allowed.

Specific instructions: You can answer in English, Danish, Dutch, Finnish, French, German, Italian, Norwegian, Spanish, or Swedish (in this exam; next time it can be another set of languages ;-) Begin the answer to each question (numbers 1 to 6) on a new page. The a,b,c,... parts with the same number can be on the same page. If you need many pages for one question, number the pages as well, for instance, "Question 3 p. 2". Write the question number on every page.

Write clearly: unreadable = wrong! Fewer points are given for unnecessarily complicated solutions. Indicate clearly if you make any assumptions that are not given in the question. In particular: in SQL questions, use standard SQL or PostgreSQL. If you use any other variant (such as Oracle or MySQL), say this; but full points are not guaranteed since this may change the nature of the question.

.

1. Modelling (12p)

1a (6p)

The domain to model is the staff working at a university department, such as CSE at Chalmers and GU. You should build an ER model covering the following concepts.

- The department has a number of **divisions**, where each division has a **name** and an **acronym**.
- Both the name and the acronym uniquely identify a division, but the database uses the acronym as a primary key.
- An **employee** has a **name**, an **id number**, and a **salary**.
- The id number uniquely identifies an employee.
- Every employee belongs to exactly one division.
- Every employee has one or more titles. Thus, for instance, one and the same employee can be both a manager and a professor.
- Some divisions are **academic divisions**, which are responsible for a set of **courses**.
- A course has a uniquely identifying **name**, as well as a set of **teachers**, who are employees at the division responsible for the course.

1b (4p)

Also write a database schema, in the form of SQL `CREATE TABLE` statements, corresponding to the description in 1a. The schema will be graded independently of your ER model. You can get started by deriving the schema from your ER model, but notice that a schema can often express constraints that the ER model cannot. Therefore. don't panic if you cannot express all the constraints in your ER model.

1c (2p)

List the constraints that are expressed by your schema (1b) but not by your ER model (1a). If there are no such constraints, just say this: it is the correct answer if it is true and if your schema and ER model are both correct.

2. Dependencies (8p)

The following table shows a few courses with their levels, teachers, and examination forms:

course	level	teacher	examination
Databases	bachelor	Johnson	exam
Databases	bachelor	Johnson	lab
Databases	master	Johnson	exam
Databases	master	Johnson	lab
Compilers	master	Paulson	lab
Calculus	bachelor	Ericson	exam
Calculus	bachelor	Ericson	lab

2a (4p)

Looking at the data alone, and applying the definitions of dependencies, list the functional and multivalued dependencies that hold for the data in the table. Based on this analysis, list the possible keys of the table.

2b (4p)

Convert the table to BCNF and to 4NF. Show both the resulting schemas and the resulting tables.

3. SQL queries (12p)

Assume tables with the following schemas for a university department:

```
Divisions (acronym, budget)
Employees (id, name, division, title, salary)
  division -> Divisions(acronym)
```

This schema is related to the one in Question 1, but not exactly the same. The budget of a division is the total amount of money usable for salaries. Primary key information left out as irrelevant for this question.

3a (3p)

Write an SQL query that lists the names of all professors and lecturers from CS and DS divisions, together with their titles, divisions, and salaries (per year, in Bitcoin, as this is a modern university). The result should look as follows:

```
 name    | title     | division | salary
-------------------------------------
 Carlson | professor | DS       | 15
 Johnson | professor | CS       | 14
 Johnson | lecturer  | DS       | 12
 Paulson | lecturer  | DS       | 16
```

3b (4p)

Create an SQL view `Genders`, which lists the names and id numbers of all employees together with their gender. The gender is computed from the id number following the Swedish system: if the second-last digit is even, the gender is F; if odd, the gender is M. The result should look as follows:

```
 name    | id          | gender
-----------------------------
 Carlson | 660606-6666 | F
 Johnson | 770707-7777 | M
 Johnson | 990909-9999 | M
 Paulson | 550505-5555 | M
```

3c (5p)

Write an SQL query that returns the average salaries of different combinations of job titles and genders. The result should look as follows:

```
 title     | gender | salary
---------------------------
 professor | M      | 14
 professor | F      | 15
 lecturer  | M      | 14
```

Hint: you can use the `Genders` view in your query.

4. Relational algebra and semantics (8p)

4a (5p)

Write a relational algebra query that does the same job as the query in Question 3c. Now you cannot assume the Genders view to be given: if you need it, you have to include its relational algebra expression in your answer.

4b (3p)

Trace the execution of the query in 4a by showing the table corresponding to every subexpression that stands for a relation. The original tables are:

```
Divisions:

acronym | budget
-----------------
CS      | 300
DS      | 500

Employees:

id          | name    | title     | division | salary
-----------------------------------------------------
660606-6666 | Carlson | professor | DS       | 15
770707-7777 | Johnson | professor | CS       | 14
990909-9999 | Johnson | lecturer  | DS       | 12
550505-5555 | Paulson | lecturer  | DS       | 16
```

5. Constraints and triggers (12p)

5a (5p)

Let us add to the description of Question 1 the concept of a **manager** of a division, who is an employee (at that division). Divisions and employees are then defined as the following tables (where we have omitted irrelevant attributes):

```
CREATE TABLE Divisions (
  acronym TEXT PRIMARY KEY,
  manager TEXT REFERENCES Employees(ident)
  ) ;

CREATE TABLE Employees (
  ident TEXT PRIMARY KEY,
  division TEXT REFERENCES Divisions(acronym)
  ) ;

INSERT INTO Divisions VALUES ('CS','770707-7777') ;
INSERT INTO Employees VALUES ('770707-7777','CS') ;
```

Unfortunately, however, these statements are not valid in SQL, because they refer to each other. We cannot create the table Divisions before Employees, but we cannot create Employees before Divisions either. But fortunately, there is a way out. Your task is to write a sequence of SQL statements that results in the two tables being created, with the following contents:

```
SELECT * FROM Divisions ;

acronym  |   manager
---------+-------------
CS       | 770707-7777

SELECT * FROM Employees ;

ident        | division
-------------+----------
770707-7777  | CS
```

5b (7p)

Assume tables similar to Question 3 but with some irrelevant details left out:

```
Divisions (acronym, budget)
Employees (id, division, salary)
  division -> Divisions(acronym)
```

The budget of a division is the total amount of money usable for salaries.

When a new employee is taken to a division, the following conditions must hold:

- The sum of all salaries of the division's employees must not exceed the budget of the division.
- The new employee's salary must not be less than the average salary of the old employees of the division.

Your task is to write a trigger that guarantees that these conditions are obeyed when a new person is employed. Procedurally,

- If the proposed salary leads to the budget being exceeded, try the largest possible salary.
- If this salary ends up below the average, reject the new employment.

6. XML (8p)

6a. Design, 4p

Write a DTD (Document Type Declaration) modelling a department as described in Question 1. Thus

- a department is a set of divisions
- each division has a name, an acronym, and a set of employees
- each employee has a name, an id number, and a set of titles
- a division can also have a list of courses
- a course has a name and a list of teachers

Try to use ID and IDREF types to guarantee the integrity of the database.

6b. Data representation, 4p

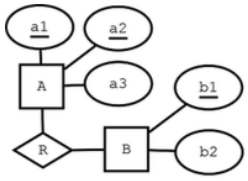
Write an XML element that represents the department described in Question 3 and is valid with respect to your DTD:

- the divisions are CS and DS
- CS has professor Johnson with salary 14
- DS has professor Carlson (salary 15) and lecturers Johnson (12) and Paulson (16)
- for ID numbers, see Question 3b
- Johnson is the manager of CS and Carlson of DS
- DS has the course Databases taught by Paulson and Johnson

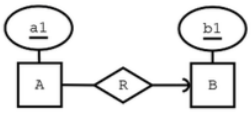
Full points for this question require not only validity with respect to your DTD, but also that your DTD is reasonable

Databases in a Nutshell ("Standard Cheatsheet")

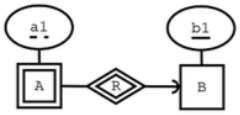
E-R diagrams and database schemas



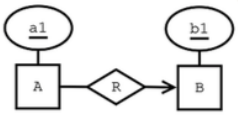

```
A(a1,a2,a3)
B(b1,b2)
R(a1,a2,b)
    (a1,a2) -> A.(a1,a2)
    b       -> B.b1
```




```
A(a1,b)
    b -> B.b1
B(b1)
```

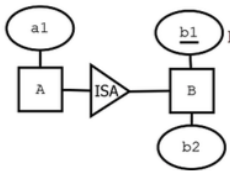



```
A(a1,b)
    b -> B.b1
B(b1)
```




```
ER-Approach:
A(a1)
B(b1)
R(a,b)
  a -> A.a1
  b -> B.b1
```

```
Null Approach:
 A(a1,b (or Null))
    b -> B.b1
 B(b1)
```




```
ER-Approach:
 B(b1,b2)
 A(b1,a1)
   b1 -> B.b1
```

```
Null-Approach:
 B(b1,b2,a1(or Null))

OO-Approach:
 A(b1,b2,a1)
 B(b1,b2)
```

Functional dependencies

Definition (tuple, attribute, value). A **tuple** has the form

$$\{A_1 = v_1, \ldots, A_n = v_n\}$$

where $A_1, \ldots, A_n$ are **attributes** and $v_1, \ldots, v_n$ are their **values**.
Definition (signature, relation). The **signature** of a tuple, S, is the set of all its attributes, $\{A_1, \ldots, A_n\}$. A **relation** R of signature S is a set of tuples with signature S. But we will sometimes also say "relation" when we mean the signature itself.
Definition (projection). If t is a tuple of a relation with signature S, the **projection** $t.A_i$ computes to the value v_i.
Definition (simultaneous projection). If X is a set of attributes $\{B_1, \ldots, B_m\} \subseteq S$ and t is a tuple of a relation with signature S, we can form a simultaneous projection,

$$t.X = \{B_1 = t.B_1, \ldots, B_m = t.B_m\}$$

Definition (functional dependency, FD). Assume X is a set of attributes and A an attribute, all belonging to a signature S. Then A is **functionally dependent** on X in the relation R, written $X \to A$, if

- for all tuples t,u in R, if $t.X = u.X$ then $t.A = u.A$.

If Y is a set of attributes, we write $X \to Y$ to mean that $X \to A$ for every A in Y.
Definition (multivalued dependency, MVD). Let X,Y,Z be disjoint subsets of a signature S such that $S = X \cup Y \cup Z$. Then Y has a **multivalued dependency** on X in R, written $X \twoheadrightarrow Y$, if

- for all tuples t,u in R, if $t.X = u.X$ then there is a tuple v in R such that
  - $v.X = t.X$
  - $v.Y = t.Y$

- $v.Z = u.Z$

Definition. An attribute A **follows** from a set of attributes Y, if there is an FD $X \to A$ such that $X \subseteq Y$.
Definition (closure of a set of attributes under FDs). The **closure** of a set of attributes $X \subseteq S$ under a set FD of functional dependencies, denoted $X+$, is the set of those attributes that follow from X.
Definition (trivial functional dependencies). An FD $X \to A$ is **trivial**, if $A \in X$.
Definition (superkey, key). A set of attributes $X \subseteq S$ is a **superkey** of S, if $S \subseteq X+$.
A set of attributes $X \subseteq S$ is a **key** of S if

- X is a superkey of S
- no proper subset of X is a superkey of S

Definition (Boyce-Codd Normal Form, BCNF violation). A functional dependency $X \to A$ **violates BCNF** if

- X is not a superkey
- the dependency is not trivial

A relation **is in Boyce-Codd Normal Form** (BCNF) if it has no BCNF violations.
Definition (prime). An attribute A is prime if it belongs to some key.
Definition (Third Normal Form, 3NF violation). A functional dependency $X \to A$ **violates 3NF** if

- X is not a superkey
- the dependency is not trivial
- A is not prime

Definition (trivial multivalued dependency). A multivalued dependency $X \twoheadrightarrow A$ is trivial if $Y \subseteq X$ or $X \cup Y = S$.
Definition (Fourth Normal Form, 4NF violation). A multivalued dependency $X \twoheadrightarrow A$ **violates 4NF** if

- X is not a superkey
- the MVD is not trivial.

Algorithm (BCNF decomposition). Consider a relation R with signature S and a set F of functional dependencies. R can be brought to BCNF by the following steps:

1. If R has no BCNF violations, return R
2. If R has a violating functional dependency $X \to A$, decompose R to two relations
   - R_1 with signature $X \cup \{A\}$
   - R_2 with signature $S - \{A\}$
3. Apply the above steps to R_1 and R_2 with functional dependencies projected to the attributes contained in each of them.

Algorithm (4NF decomposition). Consider a relation R with signature S and a set M of multivalued dependencies. R can be brought to 4NF by the following steps:

1. If R has no 4NF violations, return R
2. If R has a violating multivalued dependency $X \twoheadrightarrow Y$, decompose R to two relations
   - R_1 with signature $X \cup \{Y\}$
   - R_2 with signature $S - Y$
3. Apply the above steps to *R1* and *R2*

Concept (minimal basis of a set of functional dependencies; not a rigorous definition). A **minimal basis** of a set F of functional dependencies is a set F- that implies all dependencies in F. It is obtained by first weakening the left hand sides and then dropping out dependencies that follow by transitivity. Weakening an LHS in $X \to A$ means finding a minimal subset of X such that A can still be derived from F-.
Algorithm (3NF decomposition). Consider a relation R with a set F of functional dependencies.

1. If R has no 3NF violations, return R.
2. If R has 3NF violations,
   - compute a minimal basis of F- of F
   - group F- by the left hand side, i.e. so that all depenencies $X \to A$ are grouped together
   - for each of the groups, return the schema $XA_1 \dots A_n$ with the common LHS and all the RHSs
   - if one of the schemas contains a key of R, these groups are enough; otherwise, add a schema containing just some key

Relational algebra

relation ::=

relname	**name of relation (can be used alone)**
$\mid \sigma_{\text{condition}}$ relation	**selection (sigma)** `WHERE`
$\mid \pi_{\text{projection+}}$ relation	**projection (pi)** `SELECT`
$\mid \rho_{\text{relname (attribute+)?}}$ relation	**renaming (rho)** `AS`
$\mid \gamma_{\text{attribute*,aggregationexp+}}$ relation	**grouping (gamma)** `GROUP BY, HAVING`
$\mid \tau_{\text{expression+}}$ relation	**sorting (tau)** `ORDER BY`
$\mid \delta$ relation	**removing duplicates (delta)** `DISTINCT`
| relation $\times$ relation	**cartesian product** `FROM, CROSS JOIN`
| relation $\cup$ relation	**union** `UNION`
| relation $\cap$ relation	**intersection** `INTERSECT`
| relation $-$ relation	**difference** `EXCEPT`
| relation $\bowtie$ relation	`NATURAL JOIN`
| relation $\bowtie_{\text{condition}}$ relation	**theta join** `JOIN ON`
| relation $\bowtie_{\text{attribute+}}$ relation	`INNER JOIN`
| relation $\bowtie^{o}_{\text{attribute+}}$ relation	`FULL OUTER JOIN`
| relation $\bowtie^{oL}_{\text{attribute+}}$ relation	`LEFT OUTER JOIN`
| relation $\bowtie^{oR}_{\text{attribute+}}$ relation	`RIGHT OUTER JOIN`

projection ::=

expression	**expression, can be just an attribute**
| expression $\rightarrow$ attribute	**rename projected expression** `AS`

aggregationexp ::=

aggregation(*|attribute)	**without renaming**
| aggregation(*|attribute) $\rightarrow$ attribute	**with renaming** `AS`

expression, condition, aggregation, attribute ::=

as in SQL, but excluding subqueries

SQL

```
statement ::=
      CREATE TABLE tablename (
    * attribute type inlineconstraint*
    * [CONSTRAINT name]? constraint deferrable?
    ) ;
  |
      DROP TABLE tablename ;
  |
      INSERT INTO tablename tableplaces? values ;
  |
      DELETE FROM tablename
    ? WHERE condition  ;
  |
      UPDATE tablename
      SET setting+
    ? WHERE condition ;
  |
      query ;
  |
      CREATE VIEW viewname
      AS ( query ) ;
  |
      ALTER TABLE tablename
   +  alteration ;
  |
      COPY tablename FROM filepath ;
        ## postgresql-specific, tab-separated

query ::=
      SELECT DISTINCT? columns
    ? FROM table+
    ? WHERE condition
    ? GROUP BY attribute+
    ? HAVING condition
    ? ORDER BY attributeorder+
  |
      query setoperation query
  |
      query ORDER BY attributeorder+
        ## no previous ORDER in query
  |
      WITH localdef+ query

table ::=
      tablename
  |   table AS? tablename  ## only one iteration allowed
  |   ( query ) AS? tablename
  |   table jointype JOIN table ON condition
  |   table jointype JOIN table USING (attribute+)
  |   table NATURAL jointype JOIN table

condition ::=
      expression comparison compared
  |   expression NOT? BETWEEN expression AND expression
  |   condition boolean condition
  |   expression NOT? LIKE 'pattern*'
  |   expression NOT? IN values
  |   NOT? EXISTS ( query )
  |   expression IS NOT? NULL
  |   NOT ( condition )

type ::=
      CHAR ( integer ) | VARCHAR ( integer ) | TEXT
    | INT | FLOAT

inlineconstraint ::=     ## not separated by commas!
      PRIMARY KEY
    | REFERENCES tablename ( attribute ) policy*
    | UNIQUE | NOT NULL
    | CHECK ( condition )
    | DEFAULT value

constraint ::=
      PRIMARY KEY ( attribute+ )
    | FOREIGN KEY ( attribute+ )
        REFERENCES tablename ( attribute+ ) policy*
    | UNIQUE ( attribute+ ) | NOT NULL ( attribute )
    | CHECK ( condition )

policy ::=
      ON DELETE|UPDATE CASCADE|SET NULL
deferrable ::=
      NOT? DEFERRABLE (INITIALLY DEFERRED|IMMEDIATE)?
tableplaces ::=
      ( attribute+ )

values ::=
      VALUES ( value+ )  ## VALUES only in INSERT
    | ( query )

setting ::=
      attribute = value

alteration ::=
      ADD COLUMN attribute type inlineconstraint*
    | DROP COLUMN attribute

localdef ::=
      WITH tablename AS ( query )

columns ::=
      *         ## literal asterisk
    | column+

column ::=
      expression
    | expression AS name

attributeorder ::=
      attribute (DESC|ASC)?

setoperation ::=
      UNION | INTERSECT | EXCEPT

jointype ::=
      LEFT|RIGHT|FULL OUTER?
    | INNER?

comparison ::=
      = | < | > | <> | <= | >=
```

```
expression ::=
      attribute
  |   tablename.attribute
  |   value
  |   expression operation expression
  |   aggregation ( DISTINCT? *|attribute)
  |   ( query )

value ::=
      integer | float | string ## string in single quotes
  |   value operation value
  |   NULL

boolean ::=
      AND | OR

## triggers

functiondefinition ::=
  CREATE FUNCTION functionname() RETURNS TRIGGER AS $$
  BEGIN
*   triggerstatement
  END
  $$ LANGUAGE 'plpgsql'
  ;

triggerdefinition ::=
  CREATE TRIGGER triggernane
    whentriggered
    FOR EACH ROW|STATEMENT
  ? WHEN ( condition )
    EXECUTE PROCEDURE functionname
    ;

whentriggered ::=
    BEFORE|AFTER events ON tablename
  | INSTEAD OF   events ON viewname

events ::= event | event OR events
event  ::= INSERT | UPDATE | DELETE

triggerstatement ::=
    IF ( condition ) THEN statement+ elsif* END IF ;
  | RAISE EXCEPTION 'message' ;
  | statement ;  ## INSERT, UPDATE or DELETE
  | RETURN NEW|OLD|NULL ;

elsif ::= ELSIF ( condition ) THEN statement+

compared ::=
     expression
   | ALL|ANY values

operation ::=
     "+" | "-" | "*" | "/" | "%"
   | "||"

pattern ::=
    % | _ | character  ## match any string/char
  | [ character* ]
  | [^ character* ]

aggregation ::=
    MAX | MIN | AVG | COUNT | SUM


## privileges

statement ::=
   GRANT  privilege+ ON object TO user+ grantoption?
 | REVOKE privilege+ ON object FROM user+ CASCADE?
 | REVOKE GRANT OPTION FOR privilege
     ON object FROM user+ CASCADE?
 | GRANT rolename TO username adminoption?

privilege ::=
   SELECT | INSERT | DELETE | UPDATE | REFERENCES
 | ALL PRIVILEGES ## | ...

object ::=
   tablename (attribute+)+ | viewname (attribute+)+
 | trigger ## | ...

user ::= username | rolename | PUBLIC

grantoption ::= WITH GRANT OPTION

adminoption ::= WITH ADMIN OPTION

## transactions

statement ::=
  START TRANSACTION mode* | BEGIN | COMMIT | ROLLBACK

mode ::=
   ISOLATION LEVEL level
 | READ WRITE | READ ONLY

level ::=
   SERIALIZABLE | REPEATABLE READ | READ COMMITTED
 | READ UNCOMMITTED

## indexes

statement ::=
   CREATE INDEX indexname ON tablename (attribute+)?
```

XML

```
document ::= header? dtd? element

header ::= "<?xml version=1.0 encoding=utf-8 standalone=no?>"
  ## standalone=no if with DTD

dtd ::= <! DOCTYPE ident [ definition* ]>

definition ::=
    <! ELEMENT ident rhs >
  | <! ATTLIST ident attribute* >

rhs ::=
   EMPTY | #PCDATA | ident
  | rhs"*" | rhs"+" | rhs"?"
  | rhs , rhs
  | rhs "|" rhs

attribute ::= ident type #REQUIRED|#IMPLIED

type ::= CDATA | ID | IDREF

element   ::= starttag element* endtag | emptytag

starttag  ::= < ident attr* >
endtag    ::= </ ident >
emptytag  ::= < ident attr* />

attr ::= ident = string ## string in double quotes

## XPath

path ::=
    axis item cond? path?
  | path "|" path

axis ::= / | //

item ::= "@"? (ident*) | ident :: ident

cond ::= [ exp op exp ] | [ integer ]

exp  ::=  "@"? ident | integer | string

op   ::= = | != | < | > | <= | >=
```

Grammar conventions

- CAPITAL words are SQL or XML keywords, to take literally
- small character words are names of syntactic categories, defined each in their own rules
- | separates alternatives
- + means one or more, separated by commas in SQL, by white space in XML
- * means zero or more, separated by commas in SQL, by white space in XML
- ? means zero or one
- in the beginning of a line, + * ? operate on the whole line; elsewhere, they operate on the word just before
- ## start comments, which explain unexpected notation or behaviour
- text in double quotes means literal code, e.g. "*" means the operator *
- other symbols, e.g. parentheses, also mean literal code (quotes are used only in some cases, to separate code from grammar notation)
- parentheses can be added to disambiguate the scopes of operators, in both SQL and XML