Databases Exam

TDA357 (Chalmers), DIT620 (University of Gothenburg)

10 January 2017, 14:00-18:00, HA, HB, HC

Department of Computer Science and Engineering

Course responsible Steven Van Acker (EDIT 5472). Steven will visit the exam rooms around 15:00 and 17:00.

Results Will be published by the end of January 2017 at the latest.

Exam review See the course web page for time and place:
`http://www.cse.chalmers.se/edu/year/2016/course/TDA357/HT2016/`

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 Answer questions in English. 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.

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 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 E-R Modelling (2 parts, 12p)

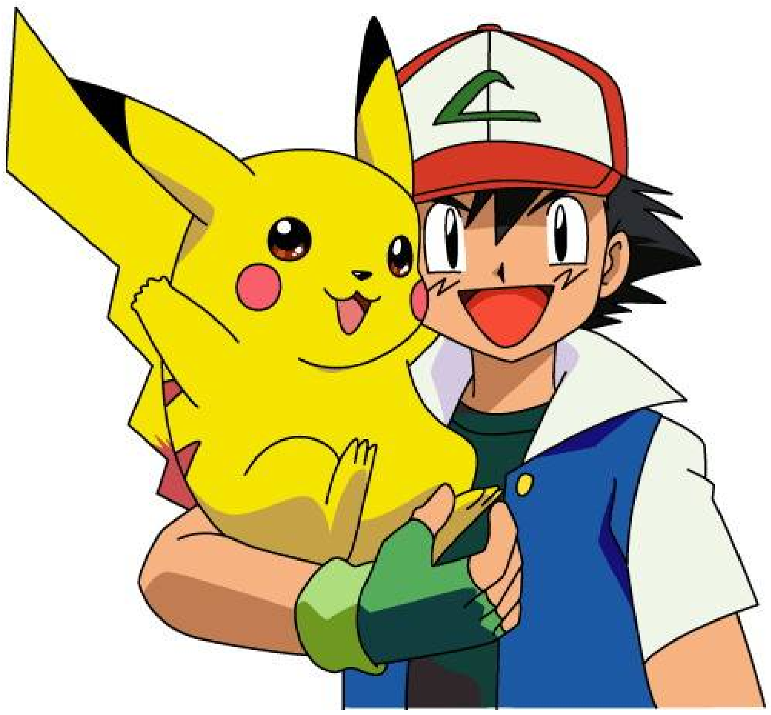


Figure 1: The Pokémon Pikachu (left) with its human trainer Ash (right)

(Note: the plural of "Pokémon" is "Pokémon". However, for the purpose of clarity, when we talk about multiple Pokémon in this text, we will explicitly use the term "Pokémons" and hopefully not insult any Pokémon fans.)

Pokémons are fictional creatures which humans, known as Pokémon Trainers, catch and train to battle each other for sport.

The domain to model is a limited subset of Pokémons:

- Each Pokémon is owned by exactly one trainer, there are no free-roaming Pokémons.
- Every trainer has a unique social security number (SSN), Pokémons have names which are not unique. Trainers rename their Pokémons so that they own no two Pokémons with the same name. (E.g. Ash can not own two Pokémons named Pikachu, he has to rename one of them. At the same time, Ash's friend Misty (also a Pokémon trainer) can own a Pokémon which is also named Pikachu).
- Trainers can become a member of any number of clubs. Clubs are located in cities, in a certain street and at a certain streetnumber, and have a unique name within that city. Cities have unique names. Streetnames are not unique. (E.g. two cities may have a street named Kyrkogatan).
- Two Pokémons can challenge each other to fight in a club. Each fight takes place in a club at a certain time and can result in one Pokémon winning, or a tie (neither Pokémon wins). Two Pokémons can only fight twice in a certain club: once as the challenger, and once as the opponent.

1a. Draw an Entity-Relationship diagram for this domain. Do not use multivalued attributes. (7p).

1b. Give the corresponding relational database schema (5p).

2 Functional dependencies and normal forms (4 parts, 10p)

Here is a part of a planet fact box in Wikipedia:

- Name: Earth
- Star: Sun
- Position in star-system: 3
- Distance from star (in millions of km): 149.6
- Radius: 6371 km
- Surface Area: 510×10^6 km^2
- Percentage of area that is water: 71%
- Percentage of area that is land: 29%
- Mass: 5.972×10^{24} kg
- Surface gravity: 9.807 m/s^2
- Atmosphere? Present
- O_2 (oxygen) percentage: 20.95%
- other gases: 79.05%

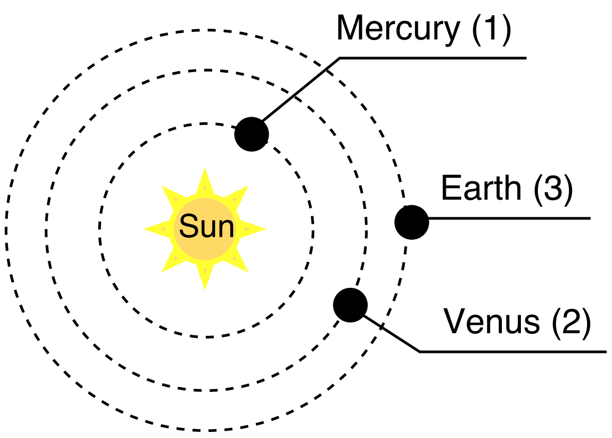


Figure 2: The planets Mercury, Venus and Earth orbit the Sun with circular orbits. The number in round brackets indicates the position in the starsystem

We assume that all stars have different names, and that planet names are only unique within their star-system. A star-system has exactly one star, all planets have circular orbits around their star at different distances. A planet's position indicates which order it has in the star-system, e.g. Earth is the 3rd planet around the Sun, after Mercury and Venus. All planets are perfect spheres, and we can thus calculate their surface area from the radius using the well-known formula $A = 4 \times \pi \times r^2$ Likewise, the surface gravity g of a planet can be derived from its mass m, its radius r and the gravitational constant G as $g = G\frac{m}{r^2}$. If a planet has O_2 or other gases, it has an atmosphere. Without an atmosphere, a planet has no gases. The surface of a planet is either all water, all land, or a combination of water and land, but nothing else.

Now, consider the task of creating a database for this data. We start with a big table with the schema

```
Planets(name, star, position, distance, radius, area, water, land, mass, gravity, atmosphere,
oxygen, otherGas)
```

2a. What functional dependencies can you find? (You dont́ need to list dependencies that follow from other ones you list.) (3p)

2b. What keys are suggested by the functional dependencies, and why? (2p)

2c. Which functional dependencies violate the Boyce Codd Normal Form (BCNF), and why? (2p)

2d. Decompose the table to bring it to BCNF. Show each step in the normalization process, and at each step indicate which functional dependency is being used. Indicate keys and references for the resulting relations. (3p)

3 SQL tables and queries (3 parts, 12p)

Consider the relation

`Planets(star, name, distance, mass, atmosphere, oxygen, water)`

which is a part of the database shown in Question 2.

3a. Write an SQL table definition with reasonable types and constraints. Store distance in millions of km (For Earth, you would store the value 149.6).(4p)

3b. Write an SQL query to determine how many planets are in orbits larger than the orbit of the fictional planet "Duna" of the fictional star "Kerbol"? (4p)

3c. We define a planet "habitable" if it satisfies all these conditions:

- orbits at a distance (in millions of km) between 100 and 200 (inclusive) from its star,
- has an atmosphere and it has an oxygen percentage between 15% and 25% (inclusive),
- has water on its surface.

Write an SQL query which returns the star and name of a planet, as well as a column `status` with value `''habitable''` if the planet is habitable, otherwise `''uninhabitable''`. (This means, return 3 values per row) (4p)

4 Relational algebra (2 parts, 8p)

4a. Given the relation `Planets(star, position, distance, mass, atmosphere, oxygen, water)` write a relational algebra query that returns, for each star with more than 5 planets, the total combined mass of all planets with an atmosphere. The query should return tuples of the form `(star, totalMass)`. (4p)

4b. Given the relations `P(star, position, distance, mass, atmosphere, oxygen, water)` and `G(star, position, gravity)`, translate the following relational algebra query to SQL (4p):

$$\tau_{maxg}(\pi_{position,maxg}(\gamma_{position,AVG(mass)\to avgm,MAX(gravity)\to maxg}(P \bowtie G)))$$

5 Views, Triggers (2 parts, 8p)

Consider an online book shop which sometimes promotes books by displaying them on the front page of their web site. Their web application uses a database created in PostgreSQL using the following statements:

```
CREATE TABLE Books (
    id INTEGER PRIMARY KEY,
    category TEXT,
    price FLOAT,
    promoted BOOLEAN DEFAULT True
);

INSERT INTO Books(id, category, price) VALUES(1, 'Dictionary', 100);
INSERT INTO Books(id, category, price) VALUES(2, 'Dictionary', 150);
INSERT INTO Books(id, category, price) VALUES(3, 'Science', 120);
INSERT INTO Books(id, category, price) VALUES(4, 'Science', 190);
INSERT INTO Books(id, category, price) VALUES(5, 'Science', 320);
```

5a. Create a new VIEW called "PromotionSummary" which outputs 3 columns named "category", "minprice" and "maxprice" containing the category name, minimum price of all promoted books and maximum price of all promoted books. A promoted book has its "promoted" attribute set to True. (4p)

5b. Create a trigger so that, when a tuple from the "PromotionSummary" view is deleted, all Books from the corresponding category have their "promoted" attribute set to False. E.g. if the entry in "PromotionSummary" for category "Novel" is deleted, all entries in "Books" with category "Novel" have their "promoted" attribute set to False. (4p)

6 Authorization, SQL Injection, Transactions (3 parts, 10p)

Consider an existing database with the following database definition in a PostgreSQL DBMS:

```
CREATE TABLE Users (
    id INTEGER PRIMARY KEY,
    name TEXT,
    password TEXT
);

CREATE TABLE UserStatus (
    id INTEGER PRIMARY KEY REFERENCES Users,
    loggedin BOOLEAN NOT NULL
);

CREATE TABLE Logbook (
    id INTEGER REFERENCES Users,
    timestamp INTEGER,
    name TEXT,
    PRIMARY KEY (id, timestamp)
);
```

6a. A database user "Alice" is granted the following permissions:

```
GRANT SELECT(id, name, password) ON Users TO Alice;
GRANT SELECT(id, loggedin) ON UserStatus TO Alice;
GRANT SELECT(id, timestamp, name) ON LogBook TO Alice;
GRANT INSERT(id, timestamp, name) ON LogBook TO Alice;
```

Alice now executes the following SQL statement:

```
INSERT INTO LogBook
    SELECT u.id, 201701101400, u.name
        FROM (UserStatus us JOIN Users u ON us.id = u.id)
        WHERE us.loggedin = True;
```

We want Alice to only have exactly the privileges that are necessary to complete this SQL statement. Does Alice have too few, exactly enough, or too many privileges? What minimal set of permissions should she be granted instead, if not the same as listed above? (4p)

6b. Users of a web application are allowed to query this database for a certain user id. This functionality is implemented in JDBC using the following code fragment:

```
...
String query =
    "SELECT * FROM UserStatus WHERE id = '" + userinput + "'";
PreparedStatement stmt = conn.prepareStatement(query);
ResultSet rs = stmt.executeQuery();
...
```

Does this code contain an SQL injection vulnerability? If it does not, why not? If it does, how would you correct the code? (2p)

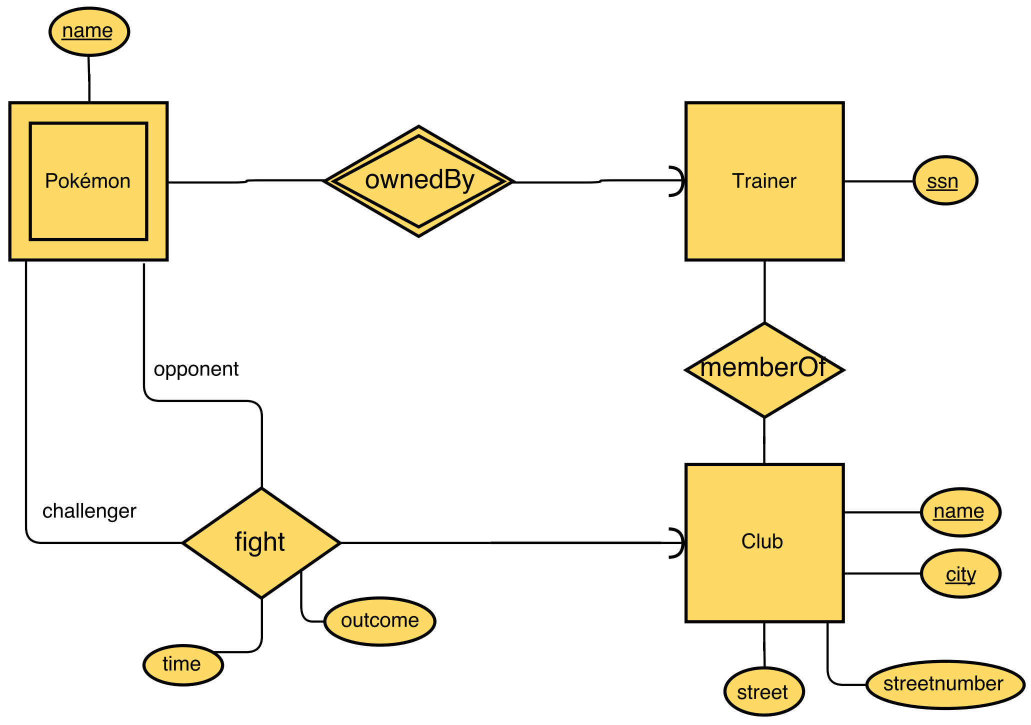
6c. The following transaction calculates the total number of entries in UserStatus as the sum of the number of logged-in and not logged-in users.

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT
   (SELECT COUNT(*) FROM UserStatus WHERE loggedin = True)
   +
   (SELECT COUNT(*) FROM UserStatus WHERE loggedin = False);
COMMIT;
```

The used transaction isolation level is not sufficient to ensure an accurate count of entries in UserStatus. Why not? Give all isolation levels that are sufficient so that the query works as expected. (4p)

Databases Exam HT2016 Solution

Solution 1a



Solution 1b

```
Trainer(ssn)
Pokemon(ssn, name)
    ssn -> Trainer.ssn
Club(name, city, street, streetnumber)
MemberOf(ssn, name, city)
    ssn -> Trainer.ssn
    (name, city) -> Club.(name, city)
Fight(ssn1, name1, ssn2, name2, name, city, time, outcome)
    (ssn1, name1) -> Pokemon.(ssn, name)
    (ssn2, name2) -> Pokemon.(ssn, name)
    (name, city) -> Club.(name, city)
```

Solution 2a

```
(1)   star, name            → * (all other attributes)
(2)   star, position        → * (all other attributes)
(3)   star, distance        → * (all other attributes)

(4)   radius                → area
(5)   area                  → radius

(6)   water                 → land
(7)   land                  → water

(8)   mass, radius          → gravity
(9)   mass, gravity         → radius
(10)  gravity, radius       → mass

(11)  atmosphere, oxygen    → otherGas
(12)  atmosphere, otherGas  → oxygen
(13)  oxygen, otherGas      → atmosphere
```

Solution 2b

These 3 subsets of attributes of the Planets relation are keys:

```
{star, name}
{star, position}
{star, distance}
```

A key is a minimal superkey. Each of these subsets is a superkey of the relation Planets because their closure is the full set of attributes of Planets. In addition, each of these superkeys is minimal because there is no subset of attributes that is also a superkey.

Solution 2c

Functional dependencies 4-13 violate BCNF (all FDs except the first 3), because the left-hand side of each of these FDs is not a superkey of the Planets relation.

Solution 2d

Step 1

FD 4 violates BCNF, so we create a new relation **Areas** and remove the **area** attribute from the **Planets** relation. After this step, the violation of FD 5 is automatically resolved.

```
Planets(* - area)
    remaining FDs: 1-3, 6-13
Areas(radius, area)
    radius -> Planets.radius
    (4) radius → area
    (5) area   → radius
```

Step 2

FD 6 violates BCNF, so we create a new relation **Surfaces** and remove the **land** attribute from the **Planets** relation. After this step, the violation of FD 7 is automatically resolved.

```
Planets(* - area - land)
    remaining FDs: 1-3, 8-13
Areas(radius, area)
    radius -> Planets.radius
    (4) radius → area
    (5) area   → radius
Surfaces(water, land)
    water -> Planets.water
    (6) water → land
    (7) land  → water
```

Step 3

FD 8 violates BCNF, so we create a new relation **Gravities** and remove the **gravity** attribute from the **Planets** relation. After this step, the violation of FD 9 and 10 is automatically resolved.

```
Planets(* - area - land - gravity)
    remaining FDs: 1-3, 11-13
Areas(radius, area)
    radius -> Planets.radius
    (4) radius → area
    (5) area   → radius
Surfaces(water, land)
    water -> Planets.water
    (6) water → land
    (7) land  → water
Gravities(mass, radius, gravity)
    mass, radius -> Planets.(mass, radius)
    (8)  mass, radius    → gravity
    (9)  mass, gravity   → radius
    (10) gravity, radius → mass
```

Step 4

FD 11 violates BCNF, so we create a new relation **Atmospheres** and remove the **otherGas** attribute from the **Planets** relation. After this step, the violation of FD 11 and 12 is automatically resolved.

```
Planets(name, star, position, distance, radius, water, mass, atmosphere,
   oxygen)
    (1) star, name     → * (all other attributes)
    (2) star, position → * (all other attributes)
    (3) star, distance → * (all other attributes)
Areas(radius, area)
    radius -> Planets.radius
    (4) radius → area
    (5) area   → radius
Surfaces(water, land)
    water -> Planets.water
    (6) water → land
    (7) land  → water
Gravities(mass, radius, gravity)
    mass, radius -> Planets.(mass, radius)
    (8)  mass, radius    → gravity
    (9)  mass, gravity   → radius
    (10) gravity, radius → mass
Atmospheres(atmosphere, oxygen, otherGas)
    atmosphere, oxygen -> Planets.(atmosphere, oxygen)
    (11) atmosphere, oxygen   → otherGas
    (12) atmosphere, otherGas → oxygen
    (13) oxygen, otherGas     → atmosphere
```

Solution 3a

```
CREATE TABLE Planets(
    star TEXT NOT NULL,
    name TEXT NOT NULL,
    distance FLOAT NOT NULL CHECK(distance > 0),
    mass FLOAT NOT NULL CHECK(mass > 0),
    atmosphere BOOLEAN NOT NULL,
    oxygen FLOAT NOT NULL CHECK((oxygen = 0 and not atmosphere) OR (
       atmosphere AND oxygen >= 0 AND oxygen <= 100)),
    water FLOAT NOT NULL CHECK(water >= 0 AND water <= 100),
    PRIMARY KEY(star, name),
    UNIQUE(star, distance)
);
```

Solution 3b

```
SELECT COUNT(*) FROM Planets WHERE
        distance > (SELECT distance FROM Planets WHERE
                    star='Kerbol' AND name='Duna');
```

Solution 3c

```
(SELECT star, name, 'habitable' FROM Planets WHERE
        distance >= 100 AND distance <= 200 AND
        atmosphere AND oxygen >= 15 AND oxygen <= 25 AND
        water > 0)
UNION
(SELECT star, name, 'uninhabitable' FROM Planets WHERE NOT(
        distance >= 100 AND distance <= 200 AND
        atmosphere AND oxygen >= 15 AND oxygen <= 25 AND
        water > 0));
```

or

```
WITH habitables AS (SELECT star, name FROM planets WHERE
        distance >= 100 AND distance <= 200 AND
        atmosphere AND oxygen >= 15 AND oxygen <= 25 AND
        water > 0)
SELECT star, name, 'habitable' FROM Planets WHERE
        (star, name) IN (SELECT star, name from habitables)
UNION
SELECT star, name, 'unhabitable' FROM Planets WHERE
        (star, name) NOT IN (SELECT star, name from habitables);
```

or

```
SELECT star, name, CASE WHEN
        distance >= 100 AND distance <= 200 AND
        atmosphere AND oxygen >= 15 AND oxygen <= 25 AND
        water > 0
        THEN 'habitable' ELSE 'uninhabitable' END
    FROM Planets;
```

Solution 4a

The query in SQL:

```
SELECT star, SUM(mass) AS totalMass FROM Planets WHERE atmosphere GROUP BY
   star HAVING COUNT(*) > 5;
```

The query in relational algebra:
$\pi_{star,totalMass}(\sigma_{atmosphere\&planetcount>5}(\gamma_{star,COUNT(*)\rightarrow planetcount,SUM(mass)\rightarrow totalMass}(Planets)))$

Solution 4b

```
SELECT position, MAX(gravity) AS maxg
        FROM (P NATURAL JOIN G)
        GROUP BY position
        ORDER BY maxg;
```

Solution 5a

```
CREATE VIEW PromotionSummary AS
        SELECT category, MIN(price) AS minprice, MAX(price) AS maxprice FROM
           Books
        WHERE promoted
        GROUP BY category;
```

Solution 5b

```
CREATE OR REPLACE FUNCTION demoteBooks() RETURNS TRIGGER AS $$
BEGIN
        UPDATE Books SET promoted = False WHERE category = OLD.category;
END
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER demoteBooksTrigger INSTEAD OF DELETE ON PromotionSummary
    FOR EACH ROW
    EXECUTE PROCEDURE demoteBooks();
```

it is acceptable to shorten this to:

```
demoteBooks() → UPDATE Books SET promoted = False WHERE category = OLD.
   category;

CREATE TRIGGER demoteBooksTrigger INSTEAD OF DELETE ON PromotionSummary
    FOR EACH ROW
    EXECUTE PROCEDURE demoteBooks();
```

Solution 6a

Alice has too many privileges, since she does not need to read the password in the Users table, nor the LogBook entries. The minimally required set of permissions is:

```
GRANT SELECT(id, name) ON Users TO Alice;
GRANT SELECT(id, loggedin) ON UserStatus TO Alice;
GRANT INSERT(id, timestamp, name) ON LogBook TO Alice;
```

Solution 6b

Yes this code contains an SQL injection vulnerability.

The vulnerability can be removed by either correctly sanitizing or escaping the data in the `userinput` variable. A better solution is to use a `PreparedStatement` with placeholder:

```
...
String query = "SELECT * FROM UserStatus WHERE id = ?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString(1, userinput);
ResultSet rs = stmt.executeQuery();
...
```

Solution 6c

The transaction is vulnerable to "non-repeatable read" and "phantom read" interferences, because the `READ COMMITTED` transaction isolation level does not protect against them. The stronger `REPEATABLE READ` isolation level is not sufficient because it still allows phantom reads. Only the `SERIALIZABLE` isolation level is sufficient, since it protects against dirty read, non-repeatable read and phantom read.