Databases Exam

TDA357 (Chalmers), DIT620 (University of Gothenburg)

7 June 2017, 14:00-18:00

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 June 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. Appended to this exam paper is a standardized cheat sheet you may use as well. 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)

Suppose you are given the following requirements for a simple database for a single season in the National Hockey League (NHL):

- the NHL has many teams,
- each team has a name, a city, a coach and a set of players,
- each player belongs to maximum one team,
- a team's name is only unique in their own city
- each player has a name, a unique person number (abbreviated as "pn"), a position (such as left wing or goalie), a skill level, and a set of injury records,
- an injury record has a description and an id, but the id on its own is not unique (the combination of person number and injury record id is unique),
- a game is played between two teams (referred to as host and guest) and has a date (such as May 11th, 1999) and a score (such as 4 to 2).
- there can be only 2 games per couple of teams (A, B) per season: one where A is the host and B the guest, and the other where B is the host and A is the guest
- a team can not play a game against itself

1a. Draw an Entity-Relationship diagram for this domain, listing any assumptions you make. Do not use multivalued attributes. (7p).

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

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

Suppose we have relation

```
R(A, B, C, D, E, F, G)
```

and functional dependencies

$BC \to D$, $DE \to F$, $FA \to B$, $BC \to G$.

2a. Relation R has three keys. State, with reasons, which two of the following are not keys of R:

- {A, B, C, D}
- {A, B, C, E}
- {A, C, D, E}
- {A, C, D, E, G}
- {A, C, E, F}

(2p)

2b. Decompose relation R to BCNF. Show each step in the normalisation process, and at each step indicate which functional dependency is being used. (3p)

2c. State, with reasons, which FD(s) of relation R violate Third Normal Form (3NF). (2p)

2d. Decompose relation R to 3NF. (3p)

3 SQL tables and queries (3 parts, 10p)

A database system used by a company's personnel department has the following relations:

```
Employees(empId, name, year, salary, entitlement, branch)
ParentalLeave(employee, startDay, startYear, endDay, endYear)
```

Employee identifiers (empId) are unique. Attribute year is the employee's year of birth. Attribute entitlement is the number of annual vacation days to which the employee is entitled. Employees have 30 days of annual vacation entitlement by default. Branch is the name of the city where the branch is located (assume that there is only one branch in each city). The personnel department keeps a record of all periods of parental leave taken by employees. The attributes startDay and endDay are integers in the range 1-366 that represent the day within the year. For each period of parental leave, the start date must be before the end date.

3a. Suggest keys and references for these relations. Write SQL statements that create these relations with reasonable constraints in PostgreSQL. (4p)

3b. Show an SQL query to get the amount of employees per branch that have/had parental leave spanning a period in two different calendar years (this means startYear and endYear are not the same). Example output row if the Stockholm branch has had 3 such employees: `('Stockholm', 3)`. (3p)

3c. Show an SQL query to get the branch name and average of salaries in that branch, for those branches that only have employees born after 1987. Sort the output by average salary. Do not show information about branches that have employees born in or before 1987. (3p)

4 Relational algebra (2 parts, 8p)

A multi-national company uses a relational database to manage information about its offices in different cities, and its employees. This database has the following relations:

```
Offices(city, supplement)
Departments(city, dname, departmentHead)
Employees(empId, name, salary, dept, city)
```

The company has one office in each city, and several departments can be located at each office. Attribute supplement is the monthly salary supplement that each employee working at that office receives (e.g. employees at the London office might receive a supplement of 1000 SEK per month to cover higher living costs in London). The default city supplement is 0 SEK. Attribute dname describes the departments function (e.g. sales or personnel). Attribute departmentHead is the employee identifier of the head of the department. Employee identifiers (empId) are unique. Attribute salary is an employees basic monthly salary. The total monthly salary for an employee can be calculated by adding the city supplement to the employees basic monthly salary.

4a. Write a relational algebra expression that finds the employee identifier, name and total monthly salary of all employees (recall that the total monthly salary for an employee can be calculated by adding the city supplement to the employees basic monthly salary). The results should be sorted by employee name. (4p)

4b. Write a relational algebra expression that finds the names of cities where there is a sales department (named "sales") and, for each of these departments, the average basic salary of the employees in that department. You can assume that every department has at least one employee. (4p)

5 Views, Triggers (2 parts, 8p)

In the year 2127, the first spaceship to colonize Mars carries 1337 colonists. When they arrive on the planet, they will build a city and live there. Following democratic principles, the spaceship captain, captain Picard, asks you to improve an SQL database to help with the voting process.

The existing SQL database was created with the following statement:

```
CREATE TABLE Votes (
    cityname TEXT PRIMARY KEY,
    votecount INT
);
```

To add a vote, you can use either `INSERT` or `UPDATE`, as shown below:

```
INSERT INTO Votes(cityname, votecount) VALUES('Mars City One', 34);
INSERT INTO Votes(cityname, votecount) VALUES('New Gothenburg', 11);
INSERT INTO Votes(cityname, votecount) VALUES('Picardia', 1);
UPDATE Votes SET votecount=votecount+3 WHERE cityname='New Gothenburg';
```

5a. Create a new VIEW called "VoteSummary" which outputs 2 columns named "cityname" and "percentage" containing the cityname and percentage of votes cast for that cityname. The output is sorted according to the votecount, highest votecount first. After the example votes above, there would be 34 votes for "Mars City One" out of a total of 49 votes, so the top row of the "VoteSummary" VIEW would be `('Mars City One', 69.3878)`. There is no need to round off the percentage. (3p)

5b. Create a trigger to update the "Votes" table, to keep track of how many colonists have not voted yet. This count will appear next to the special cityname "<not voted>". In the example above, 49 votes have been cast out of 1337 possible votes. This means the trigger needs to create or update an entry with cityname "<not voted>" and votecount 1288 (= 1337 - 49). Keep in mind that you need to create this entry if it does not exist yet. There is no need for a trigger on DELETE. Be careful with infinite recursion! (5p)

6 Authorization, SQL Injection, Transactions (3 parts, 12p)

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

Alice now executes the following SQL statement:

```
INSERT INTO LogBook
    SELECT u.id, 201706071400, 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 the correct 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 + "'";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);
...
```

The `userinput` variable is controlled directly by an attacker. What can an attacker input into `userinput` so that the SQL query returns all data in `UserStatus`? What is this specific security problem called? How should the above code be corrected to prevent this problem? (4p)

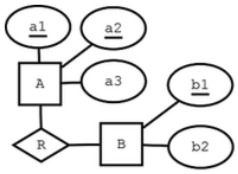
6c. The following transaction calculates the total number of entries in UserStatus as the sum of the number of logged-in and not logged-in users.

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

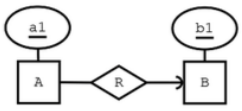
The used transaction isolation level is not sufficient to ensure an accurate count of entries in UserStatus. Why not? Give all isolation levels that are sufficient so that the query works as expected. (4p)

A Standardized cheat sheet

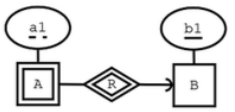
E-R diagrams and database schemas



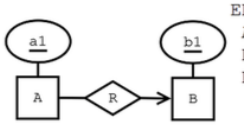

```
A(a1,a2,a3)
B(b1,b2)
R(a1,a2,b)
    (a1,a2) -> A.(a1,a2)
    b       -> B.b1
```




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

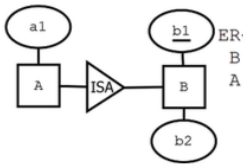



```
A(a1,b)
    b -> B.b1
B(b1)
```




```
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`
$\mid$ relation $\bowtie_{\text{condition}}$ relation	**theta join** `JOIN ON`
$\mid$ relation $\bowtie_{\text{attribute+}}$ relation	`INNER JOIN`
$\mid$ relation $\bowtie^{o}_{\text{attribute+}}$ relation	`FULL OUTER JOIN`
$\mid$ relation $\bowtie^{oL}_{\text{attribute+}}$ relation	`LEFT OUTER JOIN`
$\mid$ 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
    ) ;
  |
      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
               ## alternatives: CASCADE and SET NULL

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 | NOT? DEFERRABLE

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