Databases Exam

TDA357 (Chalmers), DIT621 (University of Gothenburg)

22 March 2019 at 8:30–12:30 at Johanneberg

Department of Computer Science and Engineering

Examiner: Aarne Ranta tel. 031 772 10 82.

Results: Will be published by 10 April.

Exam review: 18 April at 10-12 in room 6106, EDIT Building.

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, Icelandic, 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 Entity-Relationship Modelling, 12p

The domain to be modelled is music: artists, albums, songs, etc. The goal of this task could be to build a database for one's own music or a streaming service. We build the model in two steps. In both steps, your task is to build

- an E-R diagram
- a database schema, either in SQL or in the usual schema notation

You are allowed to derive the schema from the E-R diagram, but you can also build them independently. They are graded separately, with 4p for the diagrams and 2p for the schemas.

1.1 Step 1: artists and bands, 6p

Here are the main elements of the domain:

- Artists, uniquely identified by their names. An artist (as indicated by the author of a song or an album), can be
  - an individual person, who has a birth date
  - a band, which has a founding date
- An individual person can be a member in a band, also in several bands. Here we want the database to tell us who is a member of which bands.

1.2 Step 2: albums and songs, 6p

You should write the diagram for these elements beside or below the first diagram, but it should of course have arrows to the first diagram when appropriate.

- An album is identified by its title and the artist that made it, for instance, "Greatest Hits" of Bob Dylan as opposed to "Greatest Hits" by The Beatles. An album also has a year of appearance.
- A song is likewise identified by its title and the artist that made it. A song also has a length.
- A song can appear on several albums. The artist of the album can be different from the artist of the song, as for instance if the album is by "Various Artists". The database should be able to list, as separate rows, all songs that appear on different albums.

2 Functional Dependencies, 8p

2.1 Dependencies and keys, 3p

Consider the following relation and functional dependencies:

```
R(A,B,C,D,E)
A -> B
C -> D
E -> A
```

Find at least one possible key, and prove that it is a superkey by showing why all attributes belong to its closure (by using for instance transitivity).

2.2 Normalization, 3p

Is the relation in BCNF? If yes, give an argument in terms of the dependencies and keys. If not, show some of the violations and a decomposition to BCNF.

2.3 Real-world example, 2p

Give a real-world example of attributes A,B,C,D,E that would have exactly these dependencies and none else (except of course derived ones). An attempted example would have the same format as

```
Lectures(courseCode, courseTitle, date, room, teacher)
```

This would reasonably satisfy

- `A -> B`, because course code determines the title
- `C -> D`, because we assume that the course has at most one lecture each date

But it would not satisfy `E -> A`, because a teacher can have several courses. It might also have the the unwanted dependency `A C -> E`, if we assume that a each lecture has one teacher.

3 SQL queries, 12p

We go back to the domain of Question 1. We assume the following schemas, which are related to the answers of Question 1, but not exactly the same.

```
Artists(name)
Bands(name)
  name->Artists(name)
Persons(name,birthyear)
  name->Artists(name)
Albums(artist,title,year)
  artist->Artists(name)
Songs(artist,title,length)
  artist->Artists(name)
Tracks(album,song)
  album->Albums(title)
  song->Songs(title)
Members(person,band)
  person->Persons(name)
  band->Bands(name)
```

Your task is to write SQL queries with the following specifications.

3.1 Query 1, 2p

All albums by Metallica from this millennium, i.e. year 2000 or later, with their years of appearance. The format of the answer is

```
       title       | year
-------------------+------
 Death Magnetic    | 2008
```

3.2 Query 2, 3p

The total length of Björk's Album "Vespertine", meaning the total length of all songs on that album, assuming for simplicity that lengths are expressed in seconds as an integer. The format of the answer is

```
 sum
-----
 3528
```

3.3 Query 3, 3p

The titles of all solo albums, i.e. albums whose artist is not a band but a person. The format of the answer is

```
 title
----------
Vespertine
```

3.4 Query 4, 4p

The name of the oldest member of Metallica. The format of the answer is

```
 name
----------
Kirk Hammett
```

4 Algebra and theory, 8p

4.1 Relational algebra, 4p

Assuming the same relation schemas as in Question 3, express Query 2 ("the total length of the album Vespertine") in relational algebra.

4.2 Joins, 4p

Still using to the schemas in Question 3, consider the tables

```
Albums:
        title       |  artist   | year
  ------------------+-----------+------
   Vespertine       | Björk     | 2001
   Master of Puppets | Metallica | 1986

Songs:
         title       |  artist   | year | length
  -------------------+-----------+------+--------
   Hidden Place      | Björk     | 2001 |    264
   Cocoon            | Björk     | 2001 |    264
   Master of Puppets | Metallica | 1986 |    513
```

Show the tables resulting from these two different queries:

```
SELECT *
FROM Albums NATURAL JOIN Songs

SELECT *
FROM Albums FULL OUTER JOIN Songs USING (title,artist)
```

(Optional question: no points just kudos ;-) Do these joins make sense?

5 Constraints and Triggers, 12p

Consider the following definition of playlists, by reference to the table `Songs` in Question 3.

```
CREATE TABLE Playlists (
  id TEXT PRIMARY KEY,
  name TEXT,
  owner TEXT
  ) ;

CREATE TABLE PlaylistSongs (
  playlist TEXT REFERENCES Playlists(id),
  song TEXT,
  artist TEXT,
  position INTEGER,
  FOREIGN KEY (song,artist) REFERENCES Songs(title,artist),
  PRIMARY KEY (playlist,position)
  ) ;
```

5.1 Constraints, 4p

Which of the following properties are guaranteed by the given constraints?

1. Every song in the playlist exists.
2. All playlists of one and the same owner have different names.
3. All positions in a given playlist are unique.
4. The positions can be listed in order 1,2,3,... with no numbers missing.

Answer as follows:

- If a property is guaranteed, say by which constraints exactly.
- If a property is not guaranteed, give a counterexample.

5.2 Views, 3p

Create a view that, for a playlist with id `M123`, shows the contents of the playlist with the following layout:

```
 position | song          | artist    | length
----------+---------------+-----------+--------
        1  | Hidden Place  | Björk     | 264
        2  | Unforgiven    | Metallica | 383
```

5.3 Triggers, 5p

Create a trigger that enables directly building the playlist M123 via the view defined in the previous question. The behaviour should be as follows:

- Insertion of (position,song,artist,length) in M123 means an insert in `PlaylistSongs` such that
  - the song and artist are inserted as they are given by the user
  - the length is ignored (even if it contradicts the `Songs` table)
  - the position given by the user is respected, at the same time maintaining the sequential order 1,2,3,... of the playlist

Maintaining the sequential order implies the following: assume that the positions in the old playlist are 1,2,3,4. Then

- if the user inserts in the next position, 5, then 5 is also used as the position of the row inserted to the table
- if it is larger, say 8, then the position of the row inserted to the table is still 5
- if it is smaller, say 3, then the position of the row inserted is 3, but the positions of the old rows 3 and 4 are changed to 4 and 5, respectively

Note: You may notice the problem that, while the trigger is running, the unicity of positions is temporarily violated. But you can ignore the complications with this: just make sure that, when the trigger has finished its work, all positions are unique.

6 JSON/XML, 8p

The following questions can be answered in two alternative formats: either JSON/JSONPath or XML/XPath. Choose just one of the alternative formats!

6.1 SQL data representation, 2p

A natural way to represent a list of bands is to list every band with all the artists belonging to it, so that every band name is mentioned only once. For example, consider the following lists with band members and the years they joined the band (the lists are not complete):

- The Beatles: John Lennon 1960, Paul McCartney 1960, Ringo Starr 1962, George Harrison 1960
- Metallica: James Hetfield 1981, Lars Ulrich 1981, Kirk Hammett 1983, Dave Mustaine 1982, Cliff Burton 1982

Is this possible in the relational data model, such as SQL? If yes, show the table. If no, explain why.
Notice: We expect the solution to use just band names, artist names, and years as attributes - no other attributes such as ID numbers.

6.2 JSON schema, 2p

The previous question is probably easier in JSON than in SQL. Write a JSON schema suitable for listings of bands with their members and joining years. (If your answer is in XML, use either a DTD or an XML schema.)

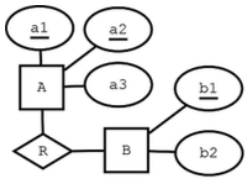
6.3 JSON data, 2p

Write a JSON object that lists The Beatles and Metallica with their members and joining years as listed above, conformant to your schema. (Or a corresponding XML object.)

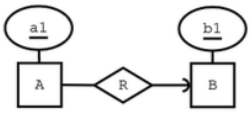
6.4 JSONPath, 2p

Formulate a JSONPath query that finds all members of The Beatles that joined later than 1960. (Or a corresponding XPath query.)

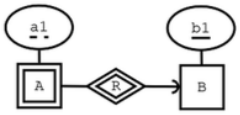
E-R diagrams and database schemas



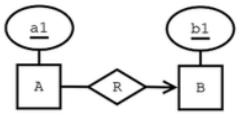

```
A(a1,a2,a3)
B(b1,b2)
R(a1,a2,b)
     (a1,a2) -> A.(a1,a2)
     b       -> B.b1
```




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

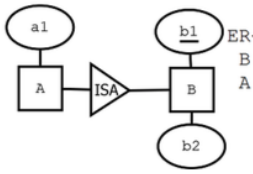



```
A(a1,b)
    b -> B.b1
B(b1)
```




```
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 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+)?
```

JSON

Both json* and member* indicate comma-separated lists. Strings are in double-quotes, numbers use decimal dot.

```
json ::= object | array | string | number | boolean
object ::= "{" member* "}"
member ::= string ":" json
array ::= "[" json* "]"
```

JSON Path: Expressions are built from operators, the result is an array with all matching json elements.
`$` is the path for the root of the document
`.` is the child operator (e.g. `$.name` gives the value of the name attribute of the root node)
`*` is the wild-card operator, it selects all attribute values of an object, or all items in an array
`..` is the recursive descent operator (e.g. `$..name` gives the value of the name attribute of all nodes)
`[n]` is array indexing (n is an integer)
`[n:m]` is array slicing, selecting all indexes from n to m in an array
`[a,b,c]` selects multiple attributes (in double quotes) or array indexes
`[?(condition)]` is used to filter values
`@` is the current object in conditions (`$.*[?(@.x>1)]` gets attributes of the root node whose x attribute exceeds 1)

JSON Schema: Each schema is a JSON document.
false matches nothing
true matches everything (same as {})
Objects contain any number of keywords (as keys), that limit what is accepted. Keywords and types of values:

- `"enum"` (array) accepts only the listed values.
- `"type"` (string) accepts only the given type, one of object/array/string/number/integer/boolean.
- `"minimum","maximum","minLength","maxLength","minProperties","maxProperties","minItems","maxItems"` (integer) specifies bounds for numbers, string lengths, array lengths and number of attributes respectively.
- `"properties"` (object with name:schema pairs) specifies schemas for attributes of objects.
  E.g. `{"properties":{"x":{"type":"string"}, "y":false}}` accepts only objects where the type of attribute "x" is a string (or "x" does not exist) and attribute "y" does not exist.
- `"additionalProperties"` (schema) specifies the schema for all attributes not mentioned in "properties".
- `"required"` (array of strings) accepts only objects that have all the listed attributes
- `"items"` (schema) accepts only arrays where all items are accepted by the given schema
- `"contains"` (schema) accepts only arrays that where at least one item is accepted by the given schema
- `"uniqueItems"` (boolean) if boolean is true, accepts only arrays where items are unique
- `"allOf", "anyOf", "oneOf"` (array of schemas) accepts only what is accepted by all of, at least one of, or exactly one of the given schemas.
- `"not"` (schema) accepts only what is not accepted by the given schema.
- `"definitions"` (object with name:schema pairs) specifies named schemas, that can be used with `"$ref"`. Only used in the root object of a schema.
- `"$ref"` (string) accepts values that are accepted by the referenced schema. Use `"#"` to refer back to the root of the schema. Use `"#\definitions\x"` to refer to definition "x".

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