EXAM
Databases (DIT620/TDA355/TDA356/TDA357)

DAY: 15 Mar 2013 TIME: 8:30 – 12:30 PLACE: M

Responsible:	Niklas Broberg, Computer Science and Engineering mobil 0706 49 35 46
Results:	Will be published on the course web page after the exam
Extra aid:	A single, hand-written A4 paper. It is legal to write on both sides. This paper should be handed in with the exam.
Grade intervals:	**U**: 0 – 23p, **3**: 24 – 35p, **4**: 36 – 47p, **5**: 48 – 60p, **G**: 24 – 41p, **VG**: 42 – 60p, **Max.** 60p.

IMPORTANT

The final score on this exam is computed in a non-standard way. The exam is divided into 7 blocks, numbered 1 through 7, and each block consists of 2 or 3 levels, named A, B, and optionally C. A level can contain any number of subproblems numbered using i, ii and so on. In the final score you can only count **ONE** level from each block. For example: if you attempt to solve the problems on all three levels in block 4 and manage to obtain 4 points for 4A (block 4, level A), 1 point for 4B and 8 points for 4C, only problem 4C (where you got your highest score) will count towards your final result, so your score for block 4 will be 8 points.

The score for each problem depends on how difficult it is (more points for harder problems) and how important I think it is (more points for more important problems). It does *not* depend on how much work it takes to answer the problem. There could very well be a 12 point problem that takes 15 seconds to answer (given that you know the right answer, of course).

The problems in each block are ordered by increasing difficulty. Hence the A problems are easy, but aim to cover the full basics of its area. The B and C level problems are more difficult, and aim to test your knowledge of the areas beyond the mere basics. If you only solve A problems your maximum score is 35 points.

Please observe the following:

- Answers can be given in Swedish or English
- Use page numbering on your pages
- Start every assignment on a fresh page
- Write clearly; unreadable = wrong!
- Fewer points are given for unnecessarily complicated solutions
- Indicate clearly if you make assumptions that are not given in the assignment

Good advice

- Most problems have been designed to give short answers. Few problem should require more than one page to answer.
- There are more problems than you are likely to solve in 4 hours. This means that you have to think about which problems you attempt to solve. If you try solve the problems in the order they are given, **you are likely to fail the exam**!

Good Luck!

Block 1 - Entity-Relationship Diagrams **max 12p**

1A (8p)

(i) (4p)

An online service for streaming video needs a database to keep track of the items in its catalogue.

The service supplies two kinds of shows: full movies, and TV series. For movies, the database needs to store the name of the movie, the year when it was released, its length, and its parental guideline rating. For simplicity we assume that the name and year together are enough to uniquely determine a movie.

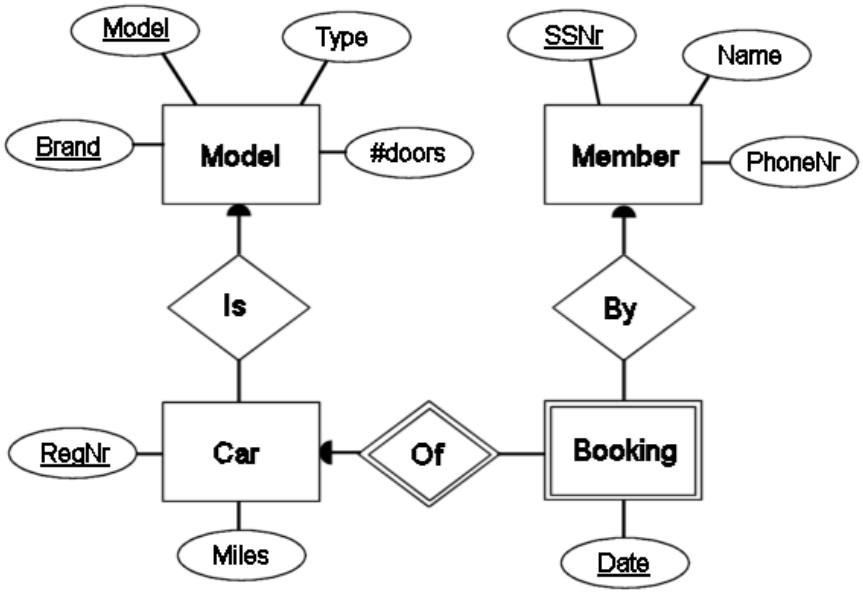
TV series are sent as episodes. Each series has one or more seasons, conveniently labeled S1, S2 etc. Within each season there are one or (typically) more episodes, labeled E01, E02 etc. Apart from the labels, the database needs to store the name and the parental guideline rating of the whole series, the year each season was originally aired, and the name and length of each episode.

Finally, to help users find shows they like, each show can be assigned to one or more predefined categories, e.g. “Drama”, “Horror”, “Family”, etc.

Your task is to draw an ER diagram that correctly models this domain and its constraints.

(ii) (4p)

The E/R diagram below depicts a rudimentary database used for a carpool.



Translate the ER diagram above into a set of relations. Mark keys and references clearly in your answer.

1B (12p)

Below is part of a database schema used for the catalogue of a music-streaming service:

Tracks(<u>*trackId*</u>, *title*, *length*)
Acts(<u>*actId*</u>)
Artists(<u>*artistId*</u>, *name*, *yearOfBirth*)
    artistId → *Acts.actId*
Groups(<u>*groupId*</u>, *name*, *yearStarted*)
    groupId → *Acts.actId*
PlaysIn(<u>*artist*</u>, <u>*group*</u>, *instrument*)
    artist → *Artists.artistId*
    group → *Groups.groupId*
Albums(<u>*albumId*</u>, *title*, *yearReleased*)
SingleActAlbums(<u>*album*</u>, *act*)
    album → *Albums.albumId*
TracksOnAlbum(<u>*album*</u>, <u>*track*</u>, *trackNr*)
    album → *Albums.albumId*
    track → *Tracks.trackId*
Participates(<u>*track*</u>, <u>*act*</u>)
    track → *Tracks.trackId*
    act → *Acts.actId*

All of this should be self-explanatory (or part of the problem), but if there is something that you don't understand, don't hesitate to ask. (Note that this schema is different from the one used in block 3 and onwards, and that the domain modeled here is not identical to the one modeled there.)

(i) (8p)
Reconstruct the ER diagram that led to these relations and constraints.

(ii) (4p)
During the translation that led to the above schema, the ER approach will have been used used consistently. For each case where applicable in the diagram you have reconstructed, state what constraints you would lose by instead using the NULL approach.

Block 2 - Dependencies and Normal Forms **max 12p**

2A (8p)

A website that hosts a centralized revision control system needs a database to store information about repositories, branches, users and patches.

The revision control system stores information about documents contained in some *repository*, and how those documents are successively changed. Users with access to a repository may look at the revision history to see what changes have been made, and look at particular older versions of documents. Revisions are done in the form of *patches*, where each patch specifies a set of changes to the repository. Further, several *branches* of a particular repository can exist, each specifying a different set of revisions (i.e. patches), thus allowing several versions of a repository to exist in parallel.
For the site in question, users must be registered in order to create repositories or supply patches. For each user the database should store their unique login name, the name to be displayed, a unique email adress, and an encrypted password.
Each repository is owned by some user. It also has a name, a description, and a unique identifying short name, used as part of the web page adress for the repository. On the page for the repository, the contact details of the owner will be displayed.
Each branch of the repository is identified by another short name (e.g. "master", "experimental", etc), unique within the repository. Users other than the owner can be given read or write access to different branches.
Each patch is created as a set of changes against a particular branch, by some user with write access to that branch. For each patch, the system stores a unique patch id (a hash code); the branch and repository that the patch applies to; the user details; a user-supplied title; the time the patch was applied; and finally the changes themselves. The order of patches is important, so for a given repository, no two patches can be applied at exactly the same time.

You are given the following schema of their intended database:

$Users(\underline{login}, name, encPwd)$
$Repositories(\underline{shortName}, repoName, owner, ownerName, ownerEmail)$
    $owner \rightarrow Users.login$
$Branches(\underline{branchName}, \underline{repository})$
    $repository \rightarrow Repositories.shortName$
$HasAccess(\underline{user}, \underline{branch}, \underline{repository}, level)$
    $user \rightarrow Users.login$
    $(branch, repository) \rightarrow Branches.(branchName, repository)$
$Patches(\underline{patchId}, repository, branch, byUser, userEmail, title, time, changes)$
    $(branch, repository) \rightarrow Branches.(branchName, repository)$
    $byUser \rightarrow Users.login$

This schema is not fully normalized, and thus suffers from a number of problems. It is your task to solve these by normalization of the schema.

(i) (4p)
For the given domain, identify all functional dependencies that are expected to hold.

(ii) (1p)
With the dependencies you have found, identify all BCNF violations in the relations of the database.

(iii) (3p)
Do a complete normalization of the schema, so that all relations are in BCNF. Also ensure that all key constraints are properly captured. (It's the end product that's important, not the steps you take to get there.)

2B (12p)

Consider the domain presented for the revision control system above, and then consider the hypothetical independencies listed below. For each of those, state whether you expect it to hold for this domain, and explain why/why not.

i. $login \twoheadrightarrow encPwd$

ii. $shortName \twoheadrightarrow branchName$

iii. $branchName, repository \twoheadrightarrow login, level$

iv. $login, repository \twoheadrightarrow branchName, level$

(Note for students having taken the course several years ago: Back then we referred to independencies as "multi-valued dependencies (MVDs)". The two terms are equivalent.)

Block 3 - SQL Data Definition Language **max 8p**

The domain for this block, and for several following blocks as well, is that of a database for the catalogue of an online music streaming site.

You are given the following schema of their intended database:

Tracks(<u>*trackId*</u>, *title*, *length*)
    length > 0
Artists(<u>*artistId*</u>, *name*)
Albums(<u>*albumId*</u>, *title*, *yearReleased*)
TracksOnAlbum(<u>*album*</u>, <u>*trackNr*</u>, *track*)
    album $\rightarrow$ *Albums.albumId*
    track $\rightarrow$ *Tracks.trackId*
    (*album*, *track*) *unique*
    trackNr > 0
Participates(<u>*track*</u>, <u>*artist*</u>)
    track $\rightarrow$ *Tracks.trackId*
    artist $\rightarrow$ *Artists.artistId*
Users(<u>*username*</u>, *email*, *name*)
    email unique
Playlists(<u>*user*</u>, <u>*playlistName*</u>)
    user $\rightarrow$ *Users.username*
InList(<u>*user*</u>, <u>*playlist*</u>, <u>*number*</u>, *track*)
    (*user*, *playlist*) $\rightarrow$ *Playlists.*(*user*, *playlistName*)
    track $\rightarrow$ *Tracks.trackId*
PlayLog(*user*, *time*, *track*)
    user $\rightarrow$ *Users.username*
    track $\rightarrow$ *Tracks.trackId*
    (*user*, *time*) *unique*

An artist can be either a solo artist or a group, the design makes no difference between the two kinds. Tracks are recorded by one or more artists, and each track can appear on one or more albums (but no more than once on each album) to account for e.g. "Greatest hits" or collection albums.
Users of the site can register, in order to create playlists, which are simply ordered collections of tracks.
Finally, the system stores a log over all songs played by registered users, to calculate statistics and to give suggestions and feedback.
(Note: The actual music files to be streamed is considered to be stored separately, outside the scope of this schema.)

3A (4p)

Write SQL DDL code that correctly implements these relations as tables in a relational DBMS. Make sure that you implement all given constraints correctly. Do not spend too much time on deciding what types to use for the various columns. We will accept any types that are not obviously wrong. Don't forget to implement all specified constraints, including checks.

3B (6p)

Note that the relation *PlayLog*, storing the log of songs played by registered users, curiously has no primary key specified. It does, however, have a uniqueness constraint. Explain why this choice was made, and what the benefits and drawbacks are for this particular situation.

3C (8p)

When a user right-clicks a song in the online interface, they get the option "Add song to playlist". If they choose this option, they may pick one of their existing playlists, or choose the option "Create new playlist". If they opt for the latter, they supply a name for the new playlist, which is created with the song in question in it.
Sketch an overview of how to ensure, through the use of views, and/or triggers, and privileges, that the database stores the correct information in the correct tables.

For any views you want to use, give the schema and explain its intended contents.

For any trigger you might include, list the trigger head ([BEFORE/AFTER/INSTEAD OF] [INSERT/DELETE/UPDATE] ON which element), and describe its intended operation in broad terms (a simple overview in plain English would be fine, you don't have to write any code).

Also specify what privileges the front-end should be granted in order to handle this use case.

Block 4 - SQL Queries **max 8p**

Use the relations for the music site from the previous block when answering the following problems.

When you are asked to list all X, you need only return the key attributes of X.

4A (4p)

(i) (2p)
Write an SQL query that lists all artists appearing on any album released this year (2013).

(ii) (2p)
Write an SQL query that lists, for each user, how many playlists that user has.

4B (6p)

Write an SQL query that lists, for each track, its trackId and title, together with the number of times that track has been played, and the number of distinct users that have played it.

4C (8p)

Write an SQL query that finds the title, length and album title of the longest track in the database. If the track appears on more than one album, list the album where it appeared first. If more than one track of the same length qualifies, list the one that was released first, as given by the album it appears on. If there is still a tie, list all such tracks.

Block 5 - Relational Algebra **max 6p**

Use the relations for the music site from the previous blocks when answering the following problems.

5A (3p)

(i) (1p)
What does the following relational algebra expression compute (answer in plain text):

$$\tau_x(\gamma_{playlistName,COUNT(*)\rightarrow x}(\sigma_{playlistName=playlist}(Playlists \times InList)))$$

(ii) (2p)
Translate the following relational algebra expression(s) to corresponding SQL:

$$let\ R1 = \gamma_{user,track,COUNT(*)\rightarrow nrTimes}(PlayLog)$$
$$\sigma_{avgNrTimes>=10}(\gamma_{track,AVG(nrTimes)\rightarrow avgNrTimes}(R1))$$

5B (4p)

Translate the following SQL query to relational algebra:

```
SELECT album, MAX(trackNr) AS nrOfTracks, SUM(length) AS totalLength
FROM Albums, TracksOnAlbum, Tracks
WHERE albumId = album
  AND trackId = track
GROUP BY albumId
ORDER BY totalLength DESC;
```

5C (6p)

Write a relational algebra expression that lists the artist(s) appearing in the highest number of distinct playlists. In case of a tie for highest number of different playlists, list all such artists.

Block 6 - Transactions **max 6p**

Use the relations for the music site from the previous blocks when answering the following problems.

6A (3p)

Consider the situation where an administrator adds a new track to the database. Data then needs to be added to the tables *Tracks*, *Participates* and *TracksOnAlbum*. We assume for the sake of simplicity that relevant data already exists in the *Artists* and *Albums* tables, and also that the track being added is recorded by one single artist, and appears on one single album.

Consider the following program (partly in pseudo-code), for handling this situation. In the code I prefix program variables with : just to distingush them from attributes (i.e. you don't need to worry about any connection to PSM or the like).

```
1 ... admin submits :title, :length, :artist, :album and :trackNr ...
2 SELECT MAX(trackId)+1 INTO newTrackId
   FROM Tracks
   WHERE post = :post;
3 INSERT INTO Tracks VALUES (:newTrackId, :title, :length);
4 INSERT INTO Participates VALUES (:newTrackId, :artist);
5 INSERT INTO TracksOnAlbum VALUES (:album, :trackNr, :newTrackId);
```

(i) (1p)
For the program as specified above, what atomicity problems could arise if it was not run as a transaction?

(ii) (2p)
For the program as specified above, what isolation problems could arise if it was not run as a *serializable* transaction?

6B (6p)

Consider the situation where a user asks the system to play the songs on one of her playlists, in order. Each time a new song begins playing, the system should log this fact. Consider the following program (partly in pseudo-code), for handling this situation. In the code I prefix program variables with : just to distingush them from attributes (i.e. you don't need to worry about any connection to PSM or the like).

```
1 ... user (:user) asks to play list (:playlist) ...
2 SELECT MIN(number), MAX(number)
   INTO (currentTrackNr, lastTrackNr)
   FROM InList
   WHERE user = :user AND playlist = :playlist;
3 while (:currentTrackNr <= :lastTrackNr) {
4   SELECT track INTO currentTrack
     FROM InList
     WHERE user = :user AND playlist = :playlist
       AND number = :currentTrackNr;
5   INSERT INTO PlayLog VALUES (:user, NOW(), :track);
6   ... fetch and stream the requested track ...
7   SET currentTrackNr = currentTrackNr+1;
8 }
```

Compare what would happen if the program above was run as a transaction with isolation level SERIALIZABLE, to if it was run with isolation level READ COMMITTED. Point out benefits and drawbacks of the two choices for this particular problem, and suggest a suitable transaction strategy to use.

Block 7 - Semi-structured Data and XML **max 8p**

The following DTD attempts to as faithfully as possible model the same domain and constraints for the music streaming site as the relations used in the previous blocks.

```
<!DOCTYPE MusicStream [
<!ELEMENT MusicStream (Track+,Album+,Artist+,User*)>
<!ELEMENT Track        (Participant+)>
<!ELEMENT Participant  EMPTY >
<!ELEMENT Album        (TrackOnAlbum+)>
<!ELEMENT TrackOnAlbum EMPTY >
<!ELEMENT Artist       EMPTY >
<!ELEMENT User         (Playlist*,PlayedTrack*)>
<!ELEMENT Playlist     (InList*)>
<!ELEMENT InList       EMPTY >
<!ELEMENT PlayedTrack  EMPTY >
<!ATTLIST Track
    trackId   ID    #REQUIRED
    title     CDATA #REQUIRED
    length    CDATA #REQUIRED >
<!ATTLIST Participant
    artist IDREF #REQUIRED>
<!ATTLIST Album
    albumId      ID    #REQUIRED
    title        CDATA #REQUIRED
    yearReleased CDATA #IMPLIED>
<!ATTLIST TrackOnAlbum
    trackNr CDATA #REQUIRED
    track   IDREF #REQUIRED>
<!ATTLIST Artist
    artistId ID    #REQUIRED
    name     CDATA #REQUIRED>
<!ATTLIST User
    username ID    #REQUIRED
    email    CDATA #REQUIRED
    name     CDATA #IMPLIED>
<!ATTLIST Playlist
    name     CDATA #REQUIRED>
<!ATTLIST InList
    number CDATA #REQUIRED
    track  IDREF #REQUIRED>
<!ATTLIST PlayedTrack
    time   CDATA #REQUIRED
    track  IDREF #REQUIRED>
]>
```

7A (5p)

(i) (2p)
Give an example XML document that is valid with respect to the DTD above.

(ii) (3p)
For a document conforming to the schema given above, what would the following XQuery expression compute? Answer in plain text:

```
<Result>
{ for $d in ( doc("musicstream.xml") )
  for $u in $d//User[/Playlist]
  let $c := count (for $x in $u/Playlist[/InList]
                   return $x)
  order by (-$c)
  return <User username="{$u/@username}">{$c}</User> }
</Result>
```

7B (8p)

When answering this question, disregard types and usage (required vs NULL etc).

Compare the DTD schema given above to the relational schema presented in block 3. For each of the four kinds of constraints listed below, give one concrete example of a constraint that is enforced by one of the schemas but not the other. For each, state which of the two schemas that enforces it, and give an example of unwanted data that could be entered into the one that does not enforce it.

(i) (1p)
A value constraint.

(ii) (2p)
A dependency constraint.

(iii) (2p)
A reference constraint.

(iv) (3p)
A multiplicity constraint.