School of Engineering and Computer Science

SWEN 432
Advanced Database Design and Implementation

Assignment 4

Due date: Friday 16 May at 23:59 pm

The objective of this assignment is to test your understanding of Object-Relational (Structure-Based) XML to Relational mapping, and Relational to XML mapping. The Assignment is worth 4.0% of your final grade. The Assignment is marked out of 100. Submit your assignment answers electronically and as a hard copy.

Background
To accomplish this assignment you will use PostgreSQL Database Management System and eXist Database Management System. PostgreSQL Manual is in the Technical web pages and is also referenced from the SWEN 432 home page. Do not print out all of the PostgreSQL Manual, use it on-line!

Note: There is a short instruction on using PostgreSQL given at the end of this handout.

Question	Topic	Marks
1.	Structure-Based Mapping of XML to Relational	**[55 marks]**
2.	Mapping a Relational Database into an XML DTD	**[30 marks]**
3.	Querying Relational and Native XML Databases	**[25 marks]**

Question 1. Structure-Based Mapping [55 marks]

There are the XML Schema `Boat_Hire_A4_14.xsd`, and the XML document `Boat_Hire_A4_14.xml` given on the Assignments course web page. Use them to answer the question. The XML scheme is a slightly simplified version of the schema `Boat_Hire_A2_14.xsd` you have used

Assume that, apart from functional dependencies implied by defined identity constraint keys, the set of functional dependencies Σ of the schema (dancingCompetition.xsd, Σ) also contains the following functional dependencies of Arenas&Libkin:

a) **[10 marks]** Define key generating functional dependencies of Arenas&Libkin that will represent a mapping of the following key and unique identity constraints contained in the `dancingCompetition.xsd`.

- **[2 marks]** `Sailor_Key`.
- **[2 marks]** `Boat_Key`.
- **[2 marks]** `Boat_Name`.
- **[2 marks]** `Sailor-Grade_Key`.
- **[2 marks]** `One_Reserve_Per_Day`.

A note on Mapping Identity Constraint Keys into Functional Dependencies

Let e_c be the context element of a key identity constraint (kic) and p_c be the absolute path to e_c. The path p_c has the form $e_1. \ldots e_c$, where e_1 is the root element. Let further p_s and p_{fi} ($i = 1, \ldots, n$) be the relative paths of the selector and field$_i$ elements of kic, repectively. The path p_s is relative with regard to e_c and paths p_{fi} are relative to last($p_c.p_s$). Finally, let e be the element whose key has been defined by kic, and let p be the absolute path having $last(p) = e$. A key identity constraint states that the sequence of values returned by an instance of the relative path $p_s.p_{fi}$ ($i = 1, \ldots, n$) uniquely identifies a node n with $lab(n) = e$ within the scope of a node n_c, where $lab(n_c) = e_c$. Note, each $last(p_s.p_{fi})$ has to be either an attribute, or a text child of a single simple element.

Example 1: Consider

```
<xsd:element name="Competition" …>
        <xsd:key name = "Dance_key">
                <xsd:selector xpath = "Dance"/>
                <xsd:field xpath = "@name"/>
        </xsd:key>
</xsd:element>
```

Here, e_c = Competition, p_c = dancingCompetitions.Competition, p_s = Dance, p_f = @name, ($n = 1$), and e = Dance.

A key defining functional dependency is an expression of the form $S \to p$, where S is a set of DTD paths (that are absolute paths) and $p \in EPath$, where $last(p) \epsilon E$. The set of paths S contains a set of n path ($n \geq 1$) ending with a @a or S (#PCDATA), and may contain a context defining DTD path.

Example 2: Here is the key defining functional dependency of the `Dance` element:

f: {dancingCompetitions.Competition, dancingCompetitions.Competition.Dance.@name} → dancingCompetitions.Competition.Dance

The path dancingCompetitions.Competition defines the context, the path dancingCompetitions.Competition.Dance.@name has an identifying attribute, and the path dancingCompetitions.Competition.Dance defines the element whose nodes are identified by @name values within the scope of a Competition node.

Mapping:
1. The context element e_c of a key identity constraint maps into the path p_c.
2. The xpath attributes of the xsd:selector and xsd:field elements map into a set of DTD paths of the form $p_c.p_s.p_{fi}$ ($i = 1, \ldots, n$)
3. The key defining functional dependency is f: $\{p_c, p_c.p_s.p_{f1}, \ldots, p_c.p_s.p_{fn}\} \to p$
4. If the path p_c contains:
   a) The root element only, or
   b) For each i in $\{2, \ldots, c\}$ it is true that e_i or e_i? is in the content model of e_{i-1},

   then the path p_c may be omitted from S, since it returns the same value for each tree tuple.

Exmaple 3:
The key identity constraint in Example 1 maps into the key defining functional dependency in Example 2.

b) **[6 marks]** There was the following question in Assignment 2: "All `Boat` elements have a unique (and not null) `@Number` attribute. All `Reserves` elements of a boat have unique (and not null) `@date` attribute. Can you make an (identity) constraint that will act as a key of `Reserves` within the scope of all `Reserves` elements using `@Number` and `@date` attributes? If you think you can, show how. If you think you cannot, explain why not." In Assignment 2, the answer to the question was negative. Now, can you use the same reasoning about `@Number` and `@date` to define a key defining functional dependency of Arenas&Libkin?

c) **[4 marks]** Consider (`Boat_Hire_A4_14.xsd`, ∑) where ∑ contains:

- All functional dependencies that can be inferred from the key and unique identity constraint in `Boat_Hire_A4_14.xsd`,
- The functional dependency you defined in question b) above, and
- Any other non trivial functional dependencies you may infer from `Boat_Hire_A4_14.xsd` and `Boat_Hire_A4_14.xml`.

Is (`Boat_Hire_A4_14.xsd`, ∑) in the XML normal form? Justify your answer.

d) **[25 marks]** Map the `Boat_Hire_A4_14.xsd` schema into a relational database schema (with the same name) using the Object-Relational mapping with the following exception:

- Do not map the elements `Boat_Hire` and `Sailors` into a relational tables, since these are just structuring elements with no properties of their own.

If you perform a correct Object-Relational mapping by defining singleton primary keys for each relational table, you will get [18 marks].

If you perform a correct Object-Relational mapping by defining composite hierarchical keys that represent a mapping of identity constraints contained in the `Boat_Hire_A4_14.xsd` schema and their representation in the form of key defining functional dependencies of Arenas&Libkin, you will get [25 marks]. The composite hierarchical keys should carry information about the hierarchical structure of the `Boat_Hire_A4_14.xml` document (for example: a `Res_Sailor` element instance is a child of a `Reserves` element instance that is a child of a `Boat` element instance that is a child of a `Marina` element instance).

Hint: Mapping of an XML functional dependency into a relational one is performed by replacing each path ending by an attribute (@) or #PCDATA (S) by the corresponding table column name. If a path p_l in the lhs of a functional dependency ends by an element ($last(p_l) \in E$) then that path p_l has to be replaced by the lhs of the key defining functional dependency for $last(p_l)$. If a path p_r in the rhs of a functional dependency ends by an element ($last(p_r) \in E$) then that path p_r has to be transformed into a set of path ending by @ or S using trivial functional dependencies of the form $p_r \rightarrow p_r.@a$ and $p_r \rightarrow p_r.e'.S$, where e' is a simple single child element of $last(p_r)$.

Example. Consider

f: {dancingCompetitions.Competition, dancingCompetitions.Competition.Dance.@name} → dancingCompetitions.Competition.Dance

Assume the key defining functional dependency of the `Competition` element is:

{dancingCompetitions.Competition.@place, dancingCompetitions.Competition.@date} → dancingCompetitions.Competition

then, according to the pseudo transitivity and union inference rules, f can be transformed into

{dancingCompetitions.Competition.@place, dancingCompetitions.Competition.@date, dancingCompetitions.Competition.Dance.@name} → {dancingCompetitions.Competition.Dance.@name, dancingCompetitions.Competition.Dance.@category, dancingCompetitions.Competition.Dance.@origin, dancingCompetitions.Competition.Dance.@order_no}

e) **[5 marks]** Implement your relational `Boat_Hire` database using PostgreSQL. Name it `user_id_Boat_Hire`. Populate your database by data contained in `Boat_Hire_A4_14.xml.`

f) **[5 marks]** What is the normal form of your relational `user_id_Boat_Hire` database? To answer the question, map all functional dependencies of Arenas&Libkin you defined by answering questions a), b), and c) above into relational functional dependencies and embed them into relation schemes (tables) of your relational schema `user_id_Boat_Hire`.

Note: Imbedding functional dependencies into relation schemes is performed by projecting the set of functional dependencies onto the set of attributes of each relation scheme (table).

Submit your answers to questions a), b), c), and f) as a hard copy. Do not delete your `user_id_Boat_Hire` database (question 1.d) and 1.e)). Pavle will check them using PostgreSQL.

After submitting your answers, execute the following SQL commands against your `user_id_Boat_Hire` relational database:

```
grant connect on database user_id_Boat_Hire to pmogin;

grant select on all tables in schema public to pmogin;
```

This way, you will allow Pavle to view your database, execute your queries against your databases, and mark your answers properly.

Question 2. Relational to XML DTD Mapping [20 marks]

a) **[15 marks]** Use Inclusion-Dependency Mapping to map the `Boat_Hire` relational database schema you produced in question 1 above into an XML DTD. Call it `Boat_Hire.dtd`.

b) **[5 marks]** Compare the structure of your `Boat_Hire.dtd` you have produced in question 2.a) above and the `Boat_Hire_A4_14.dtd` given on the course Assignments page and comment on roundtripping.

Question 3. Querying Relational and Native XML Databases [25 marks]

In this question, you will define and execute queries against relational and XML databases from question 1 above. To execute SQL queries against a relational database, use PostgreSQL and to execute XQuery queries against XML databases, use eXist. You will also need to measure response times of your queries. To accomplish that, use VACUUM and EXPLAIN ANALYZE commands of PostgreSQL and the report of eXist's Query Dialog (query compilation and execution time).

a) **[7.5 marks]** Declare the following query against your `user_id_Boat_Hire` relational database and against the `Boat_Hire_A4_14.xml` document (as a native XML database):

Find those marinas that have boats with no reserves. Your relational query answer should contain the following columns: `marina_name, unused_boat_Number, unused_boat_name`. Apply renaming (operator `AS`) when declaring the SQL query.

Your XML query answer should conform to the following DTD fragment:

```
<!ELEMENT Unused_Boats (Marina)*>
<!ELEMENT Marina (Boat+)>
<!ATTLIST Marina name CDATA #REQUIRED><!ELEMENT Boat EMPTY>
<!ATTLIST Boat Number CDATA #REQUIRED name CDATA #REQUIRED>
```

Note: your answer has to contain Marina elements of only those marinas that have unused boats (i.e. if all boats in a marina have reserves, the tag for that marina should not apeare in your answer).

b) **[5 marks]** Declare the following query against your `user_id_Boat_Hire` relational database and against the `Boat_Hire_A4_14.xml` document (as a native XML database):

Find all skippers. Your relational query answer should contain the following columns: `skipper_SailorId, skipper_name`. Apply renaming (operator `AS`) if needed when declaring the SQL query.

Your XML query answer should conform to the following DTD fragment:

```
<!ELEMENT Skippers (Sailor)*><!ELEMENT Sailor EMPTY>
<!ATTLIST Sailor SailorId CDATA #REQUIRED name CDATA #REQUIRED>
```

c) **[7.5 marks]** Declare the following query against your `user_id_Boat_Hire` relational database and against the `Boat_Hire_A4_14.xml` document (as a native XML database):

Find the skipper(s) who made the maximum number of reserves. Your relational query answer should contain the following columns: `SailorId, no_of_reserves`.

Your XML answer should conform to the following DTD fragment:

```
<!ELEMENT Most_Active_Sailors (Sailor)*>
<!ELEMENT Sailor EMPTY>
<!ATTLIST Sailor SailorId CDATA #REQUIRED no_of_reserves CDATA
#REQUIRED>
```

d) **[5 marks]** Analyze the time and logical complexity of the six queries you defined and executed when answering question 3.a), 3.b), and 3.c) above. Show your conclusions regarding the causes of noticed differences between semantically the same queries against relational and XML databases. To obtain the information about the response time of a SQL query and to get a help for analyzing its time complexity, use `VACUUM` and `EXPLAIN ANALYZE` commands of PostgreSQL. Use the report of the eXist Query Dialog to obtain information about the compilation and execution times of your XQuery queries. To estimate the logical complexity of a query, use the effort you had to invest to declare the query. Express the estimate of the logical complexity on the scale from 1 to 5, where 1 designates a very low complexity and 5 designates a high complexity.

Summarize your findings in the following table:

Query	Time to compile and execute	Effort invested
Q3.a) relational		
Q3.a) XML		
Q3.b) relational		
Q3.b) XML		
Q3.c) relational		
Q3.c) XML		

Submission Instruction

Submit all your answers both electronically and as a hard copy.

Using PostgreSQL on a Workstation

We have a command line interface to PostgreSQL server, so you need to run it from a Unix prompt in a shell window. To enable the various applications required, first type

```
> need postgresql
```

You may wish to add the "`need postgresql`" command to your `.cshrc` file so that it is run automatically. Add this command after the command need `SYSfirst`, which has to be the first need command in your `.cshrc` file.

There are several commands you can type at the Unix prompt:

```
> createdb 〈db name〉
```

Creates an empty database. The database is stored in the same PostgreSQL default school cluster used by all the students in the class. To ensure security, you must name your database by your **`userid`**. You only need to do this once (unless you get rid of your database to start again).

```
> psql  [ -d 〈db name〉 ]
```

Starts an interactive SQL session with PostgreSQL to create, update, and query tables in the database. The `db name` is optional (unless you have multiple databases)

```
> dropdb 〈db name〉
```

Gets rid of a database. (In order to start again, you will need to create a database again.)

```
> pg_dump --inserts 〈db name〉 > 〈file name〉
```

Dumps your database into a file in a form consisting of a set of SQL commands that would reconstruct the database if you loaded that file.

```
> psql -d <database_name> -f <file_name>
```

Copies the file `<file_name>` into your database `<database_name>`.

Inside an interactive SQL session, you can type SQL commands. You can type the command on multiple lines (note how the prompt changes on a continuation line). End commands with a '`;`'

There are also many single line PostgreSQL commands starting with '`\`' . No '`;`' is required. The most useful are

`\?` to list the commands,

`\i` `〈file name〉`

loads the commands from a file (eg, a file of your table definitions or the file of data we provide).

`\dt` to list your tables.

\d ⟨`table name`⟩ to describe a table.

\q to quit the interpreter

`\`**`copy`**` <table_name> `**`to`**` <file_name>`
Copy your `table_name` data into the file `file_name`.

`\`**`copy`**` <table_name> `**`from`**` <file_name>`
Copy data from the file `file_name` into your table `table_name`.

Note also that the PostgreSQL interpreter has some line editing facilities, including up and down arrow to repeat previous commands.
For longer commands, it is safer (and faster) to type your commands in an editor, then paste them into the interpreter!