Brno University of Technology

Faculty of Information Technology

Database Systems

Documentation describing the final database schema

Database system for Hospital

Zubrik Tomáš, (xzubri00)
Stupinský Šimon, (xstupi00)

April 30, 2018

Contents

Chapter 1

Basic database schema objects

1.1 Creating tables

Entity Relationship Diagram was transformed into relational database schema. All tables were then created by using command `CREATE TABLE`. In this part of script there was necessary to specify primary keys and add specific constraints. Every table is linked with each other by foreign keys referencing primary keys from another table.

1.2 Inserting data

In next step we have filled created tables with data by using command `INSERT`. Inserted values are checked or modified by later implemented triggers.

1.3 Constraints

Constraints to tables were added by command `ALTER TABLE [name] ADD CONSTRAINT`. Constraints we have declared mostly check if number is positive integer. There is also constraint to check number of numbers of telephone number and constraint to check if set visiting hours are in acceptable range. Other constraints like correctness of ID number of patient and IBAN of patient are checked by triggers.

1.4 Generalization/specialization

Relationship generalization/specialization is present and used in our database. Entity `PERSON` has 3 specializations, `DOCTOR, NURSE` and `PATIENT`. After considering that also doctor or nurse can be patients in future, we decided to implement method, where exists 4 tables individually. Main table `PERSON` keep common data about all figuring entities to avoid redundancy. Remained entities `DOCTOR, NURSE` and `PATIENT`, every has its own data, that differ from each other.

Chapter 2

Advanced database schema objects

2.1 Database triggers

A database trigger is procedural code that is automatically executed in response to events on a particular table or view in a database. Our SQL script contains 4 database triggers that show different functionality and are called by different circumstances. Triggers may change or even update values in table columns BEFORE or AFTER some database event for example as `UPDATE` and so.

2.1.1 Trigger `AUTO_INCREMENTATION_ID`

Trigger is used for generating unique ID numbers for `DEPARTMENT` as `DEPARTMENT_ID` (Primary Key). Trigger uses created `SEQUENCE` of numbers starting with 1000. For every new added department there will be generated new ID number incremented by 100. It is called with attribute `BEFORE INSERT`, that means before the value in column is inserted.

Usage Demonstration

#	DEPARTMENT_ID	NAME
1	**1000**	NEUROLOGY
2	**1100**	UROLOGY
3	**1200**	RADIOLOGY
4	**1300**	CARDIOLOGY
5	**1400**	ORTHOPAEDICS
6	**1500**	GENERAL SURGERY
7	**1600**	GYNECOLOGY
8	**1700**	PEDIATRICS
9	**1800**	PSYCHIATRY

Table 2.1: Table DEPARTMENT

2.1.2 Trigger CHECK_ID_NUMBER

Trigger checks if newly inserted or updated value of column ID_NUMBER that represent identification number of person is in syntactically correct format. Correct format for our purpose means format for Slovakia and Czech Republic ID numbers. If ID number is incorrect, trigger raises an application error with relevant error code. Defined error codes are FORMAT_ERROR (-20100), DAY_ERROR (-20200), MONTH_ERROR (-20300), YEAR_ERROR (-20400) and DIVIDE_ERROR (-20500).

Usage Demonstration

Invalid LENGTH of the ID_NUMBER. ⇒ 990104757 40

```
UPDATE PERSON SET ID_NUMBER = 99010475740 WHERE PERSON_ID = 1
ORA-20100: The bad format of the identification number (bad count of numbers) !
```

Invalid DAY of the ID_NUMBER. ⇒ 9901327574

```
UPDATE PERSON SET ID_NUMBER = 9901327574 WHERE PERSON_ID = 10
ORA-20200: The given DAY is not in the correct format, check it!
```

Invalid MONTH of the ID_NUMBER. ⇒ 9913047574

```
UPDATE PERSON SET ID_NUMBER = 9913047574 WHERE PERSON_ID = 10
ORA-20300: The given MONTH is not in the correct format, check it!
```

Invalid YEAR of the ID_NUMBER. ⇒ -990104757

```
UPDATE PERSON SET ID_NUMBER = -990104757 WHERE PERSON_ID = 10
ORA-20400: The given YEAR is not in the correct format, check it!
```

Indivisibility by number ELEVEN ⇒ 0123456789

```
UPDATE PERSON SET ID_NUMBER = 0123456789 WHERE PERSON_ID = 1
ORA-20500: The identification number must be divisible by 11 !
```

2.1.3 Trigger CHECK_IBAN

Trigger checks if newly inserted or updated value of column ACCOUNT_NUMBER that represent IBAN number of patient is in syntactically correct format. We consider only Slovak and Czech IBAN numbers as correct starting with prefix SK or CZ. If IBAN number is incorrect, trigger raises an application error with relevant error code. Defined error codes are LENGTH_ERROR (-20900), COUNTRY_ERROR (-20900) and FORMAT_ERROR (-20700).

Usage Demonstration

Invalid LENGTH of the ACCOUNT_NUMBER. ⇒ CZ690710178124000000415 90

```
UPDATE PATIENT SET ACCOUNT_NUMBER = 'CZ6907101781240000004159 0'
WHERE PATIENT_ID = 27
ORA-20900: The length of the account number must be 24!
```

Invalid COUNTRY of the ACCOUNT_NUMBER. ⇒ UK690710178124000000415 90

```
UPDATE PATIENT SET ACCOUNT_NUMBER = 'UK690710178124000000415 9'
WHERE PATIENT_ID = 27
ORA-20800: Uncorrect format of the account number!
```

Invalid CHECK SUMMARY of the ACCOUNT_NUMBER. ⇒ CZ690710178124000000415 8

```
UPDATE PATIENT SET ACCOUNT_NUMBER = 'CZ690710178124000000415 8'
WHERE PATIENT_ID = 27
ORA-20700: The check sum of the account number is not valid!
```

2.1.4 Trigger SET_TURNDATE

Trigger is used with attribute BEFORE UPDATE, that means before updating the value of column STATE in table DOCTOR. If value in STATE is updated to 'INACTIVE' trigger gets actual date using SYSDATE and update column TURNDATE that represents turn-out date of certain doctor.

Usage Demonstration

DOCTOR_ID	SPECIALIZATION	DEGREE	POSITION	ABREVIATION	STATE	TURNDATE
12	Anesthesiology	MUDr., prof.	Head doctor	MD., DA. FFA	ACTIVE	-

Table 2.2: Table before the execution of the trigger

DOCTOR_ID	SPECIALIZATION	DEGREE	POSITION	ABREVIATION	STATE	TURNDATE
12	Anesthesiology	MUDr., prof.	Head doctor	MD., DA. FFA	UNACTIVE	29-APR-18

Table 2.3: Table after the execution of the trigger

2.2 Database procedures

PL/SQL procedures behave very much like procedures in other programming language. They act like functions, but they do not return any value and are called by its identifier name. Our SQL script contains 3 nontrivial procedures that have been implemented with their specific functionality. All procedures have 1 compulsory parameter. After execution of main body of procedures, output information is printed in client's terminal window. To achieve this printing there is necessary to execute command `SET serveroutput ON.`

2.2.1 Procedure `DETAILS_EXAMINATION`

Procedure is used to get information about certain examination. Compulsory parameter of procedure is integer number that represents `EXAMINATION_ID.` Procedure consists of 3 individual selects. In a case of non-existing input examination id number `EXCEPTION` will be raised with printing info message.

Usage Demonstration

```
Listing of details about the specific procedure
```

```
EXAMINATION with number ID 42 has these details:
TYPE OF EXAMINATION: UltraSound
HOSPITALIZATION: 23
DOCTOR : John Friedrich
PATIENT: Maria Curry
DATE: 22-MAR-17 04.22.00.00 AM
```

2.2.2 Procedure `MEDICAMENTS_COUNT`

Procedure counts number of all medicaments in hospital database system and number of medicament with specified side effect given as parameter of called procedure. In the procedure there is used explicit `CURSOR` that is programmer-defined cursor for gaining more control over the context area. When `EXCEPTION` is raised procedure checks exception type and prints error message to client's terminal window. There is also used variable with data type %ROWTYPE.

Usage Demonstration

```
Listing of informations about the medicaments with specific SIDE_EFFECT
```

```
The database of the Hospital contains: 20 MEDICAMENTS.
The count of medicaments, which have the side effect dizziness: 4
The percentage rate of medicaments with side effect dizziness
to all medicaments in the database of hospital is 20 %
```

2.2.3 Procedure REMOVE_OLD_EXAMINATION

Procedure deletes examination entries older than the given date, which figures as input parameter. Procedure uses explicit CURSOR and variable with data type %ROWTYPE.

Usage Demonstration

The example of the functionality of the procedure

```
SELECT * FROM EXAMINATION
WHERE DATE_AND_TIME < TO_TIMESTAMP('01-01-2016', 'DD-MM-YYYY');

BEGIN
  REMOVE_OLD_EXAMINATION(TO_TIMESTAMP('01-01-2016', 'DD-MM-YYYY'));
END;/

SELECT * FROM EXAMINATION
WHERE DATE_AND_TIME < TO_TIMESTAMP('01-01-2016', 'DD-MM-YYYY');
```

EXAM_ID	DOCTOR_ID	DEPAR_ID	HOSP_ID	SPECIALIZATION	DATE_AND_TIME
3	6	1000	2	MRI	04-AUG-14 09.36.01.00 PM
4	10	1000	2	Done Density	14-AUG-14 11.11.11.00 AM
23	10	1500	13	Ultra Sound	17-DEC-15 09.44.11.00 AM
24	12	1500	13	Operation	25-DEC-15 12.53.24.00 AM

Table 2.4: Result table of the first SELECT (before the calling of the procedure)

EXAM_ID	DOCTOR_ID	DEPAR_ID	HOSP_ID	SPECIALIZATION	DATE_AND_TIME

Table 2.5: Result table of the second SELECT (after the execution of the procedure)

2.3 Materialized View

SQL script contains two variants of MATERIALIZED VIEWS. The first of them has name NURSE_VIEW. It represents view for NURSE that contain information about DOCTOR. Command CREATE MATERIALIZED VIEW was used to create it. We have used 2 important clauses, BUILD IMMEDIATE for immediate creating of view and REFRESH COMPLETE for refreshing all content by calling the command used to refresh the view. To create materialized view by other user we have used the command CONNECT login/password. For deleting and updating this view there is necessary for other user to connect. When second user is accessing the first user's tables there is necessary to access the table by the owner's login.

Usage Demonstration

SURNAME	NAME	SPECIALIZATION	POSITION
Cissell	Charlyn	Neurology	ordinary doctor
Dlneen	Del	Orthopedic Surgery	Head doctor
...	...	...	...
Janz	Junior	Pediatrics	Head doctor
Pegues	Paz	Surgery	ordinary doctor
...	...	...	....
Zarrella	Zachary	Anesthesiology	Head doctor

Table 2.6: Tables from the Original Materialized View.

Insert new data to the table DOCTOR AND PERSON (for the simplicity, were used the simplified commands)

```
INSERT INTO PERSON
VALUES(33, 'Peter', 'Panelak', '25-04-1977', 'Zvolen', 'Listova', 3, 'Slovakia', 'M');
INSERT INTO DOCTOR
VALUES(33, 'Anestesiology', 'MD, FFA', 'prof.', 'Doctor');
```

The TABLES in the Materialized View are not changed after the execution command INSERT. To update the data in the tables, it is need to use command:
`EXECUTE DBMS_SNAPSHOT.REFRESH('NURSE_VIEW', 'COMPLETE')`

SURNAME	NAME	SPECIALIZATION	POSITION
Cissell	Charlyn	Neurology	ordinary doctor
Dlneen	Del	Orthopaedic Surgery	Head doctor
...	...	...	...
Janz	Junior	Pediatrics	Head doctor
PANELAK	**PETER**	**ANESTESIOLOGY**	**DOCTOR**
Pegues	Paz	Surgery	ordinary doctor
...	...	...	....
Zarrella	Zachary	Anetesiology	Head doctor

Table 2.7: Tables after actualization in the relevant MATERIALIZED VIEW.

In the second variant of the **Materialized View**, there were created the materialized records, basically materialized logs, which contain the changes of the Tables, that are part of this View. They make it possible to use a quick recovery after confirming the changes instead of the complete recovery that would require the full query of the materialized view, which would take longer time.

Subsequently, there was created the **Materialized View** individually, that contains the `SELECT` query with `INNER JOIN` two tables. Specifically it is the listing of the IDs of doctors who have registration in the database that they have done certain examinations, which results from the join tables `DOCTOR` and `EXAMINATION`.

The Materialized View contains the following specific options. For optimization of reading data from database tables there was used clause `CACHE` to work with cache memory.

For immediate fulfillment of view after its creating, there was used clause `BUILD IMMEDIATE`. As has been mentioned, were used the materialized logs for quickly actualization the views after the confirming changes in the tables. This feature ensures the clause `REFRESH FAST ON COMMIT`.

As the last clause there was enabled the `QUERY REWRITE` for the optimizer using `ALTER SESSION SET query_rewrite_enabled = TRUE`, and was used the EXPLAIN PLAN to query. In the report below we can see that materialized view has been used. The optimizer will use a materialized view when evaluating the relevant query.

ID	OPERATION	NAME
0	SELECT STATEMENT	
1	NESTED LOOPS	
2	NESTED LOOPS	
3	TABLE ACCESS FULL	EXAMINATION
4	INDEX UNIQUE SCAN	PK_DOCTOR_ID
5	TABLE ACCESS BY INDEX ROWID	DOCTOR

ID	OPERATION	NAME
0	SELECT STATEMENT	
1	MAT_VIEW REWRITE ACCESS FULL	NURSE_VIEW_1

Table 2.8: Compare the explain plan at the Materialized View. The first table is without a query permission `QUERY REWRITE` and the second table with this option enabled.

2.4 Explain Plan

`EXPLAIN PLAN` clause is used to get knowledge about query performance. For optimization we have chosen `SELECT`, that shows how many obligations has certain doctor. This select uses tables `PERSON`, `DOCTOR` and `OBLIGATION`. After execution of explain plan on above select, there is used select, that works with data in system tables. Then the specific explain plan is shown by optimizer.

Explain plan of above select looks like this:

ID	OPERATION	NAME	ROWS	BYTES	COST (%CPU)	TIME
0	SELECT STATEMENT		22	1584	4 (25)	00:00
1	HASH GROUP BY		22	1584	4 (25)	00:00
2	NESTED LOOPS		22	1584	3 (0)	00:00
3	NESTED LOOPS		22	1584	3 (0)	00:00
4	TABLE ACCESS FULL	OBLIGATION	22	286	3 (0)	00:00
5	INDEX UNIQUE SCAN	PK_PERSON_ID	1		0 (0)	00:00
6	TABLE ACCESS BY INDEX ROWID	PERSON	1	59	0 (0)	00:00

Table 2.9: EXPLAIN PLAN of the specific SELECT without the index.

In zero row there is executed operation `SELECT STATEMENT`, that represents execution of command `SELECT`. Operation `HASH BY GROUP` in the first row represents using this operation to group table elements by hash key using `DOCTOR_ID`, `NAME` and `SURNAME`. In the second and third row there is operation `NESTED LOOPS`, that represents joining of tables, where every entry of the first table is compared with every entry of second one. Operation `TABLE ACCESS FULL` in the fourth row means transition of whole table without using indexes. Operation `INDEX UNIQUE SCAN` reads the index nodes down to the leaf move level

and then returns `ROWID` for appropriate single row.

Next we have created 2 indexes by using command `CREATE INDEX` and then execute explain plan with them to see optimized results.

ID	OPERATION	NAME	ROWS	BYTES	COST (%CPU)	TIME
0	SELECT STATEMENT		22	1584	2 (50)	00:00
1	HASH GROUP BY		22	1584	2 (50)	00:00
2	NESTED LOOPS		22	1584	1 (0)	00:00
3	NESTED LOOPS		22	1584	1 (0)	00:00
4	INDEX FULL SCAN	ID_INDEX	22	286	1 (0)	00:00
5	INDEX UNIQUE SCAN	PK_PERSON_ID	1		0 (0)	00:00
6	TABLE ACCESS BY INDEX ROWID	PERSON	1	59	0 (0)	00:00

Table 2.10: EXPLAIN PLAN of the specific SELECT with an established index.

The fourth row now contains operation `INDEX FULL SCAN`, that means index is read in tree order and we can see that database has used created index leading to optimization. CPU cost has been reduced.

2.5 Granting Privileges

Granting privileges for database access to other user, that is already defined as user in database system was made by command `GRANT ALL ON [table] TO [user]`. Other user has full access to all database tables and user-defined procedures. It allows other user to insert, update or delete entries from database tables.

Appendix A

Data Diagrams

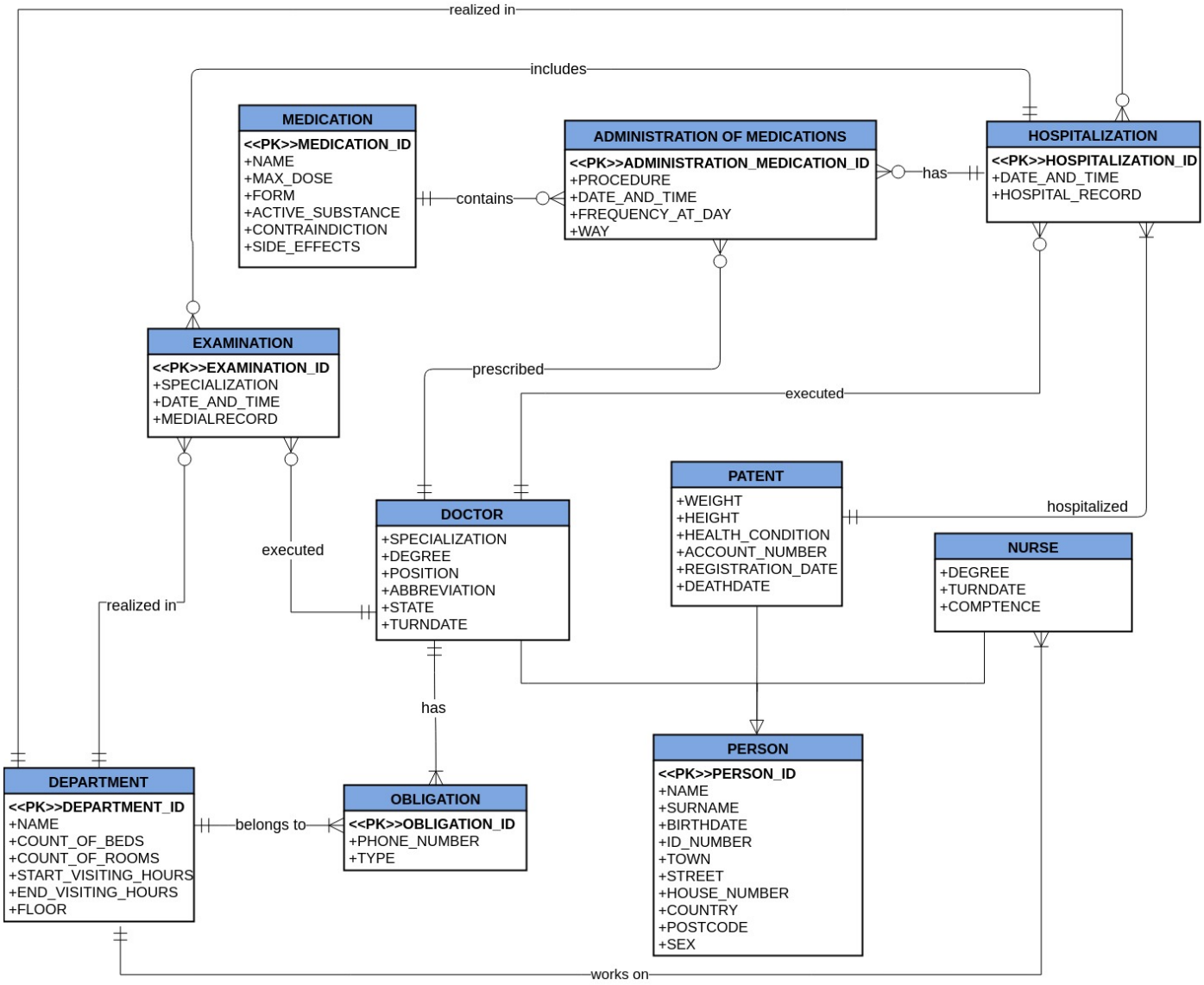

Figure A.1: **Entity Relationship diagram.**

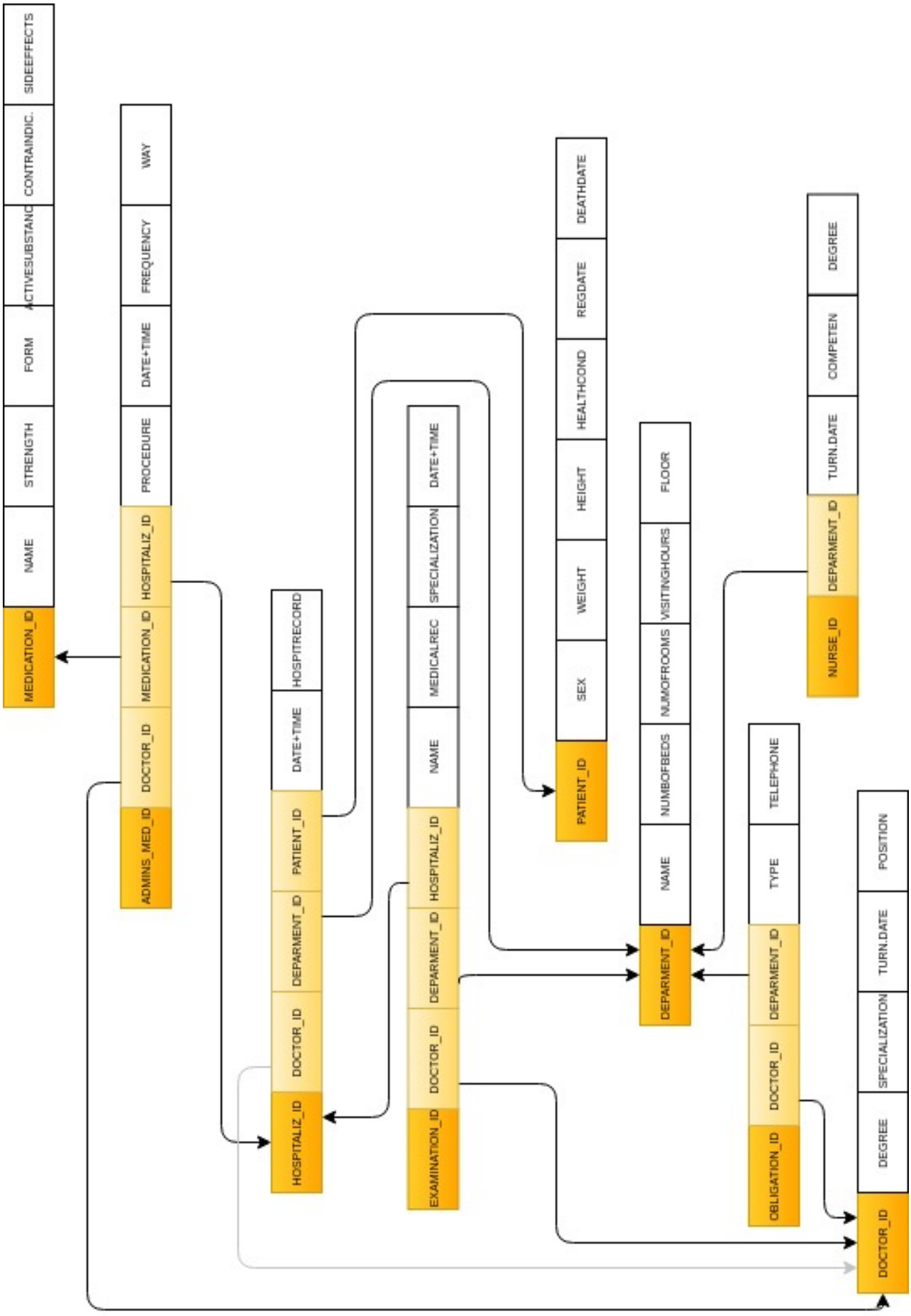

Figure A.2: **Transformation ER Diagram to Tables of Database.**

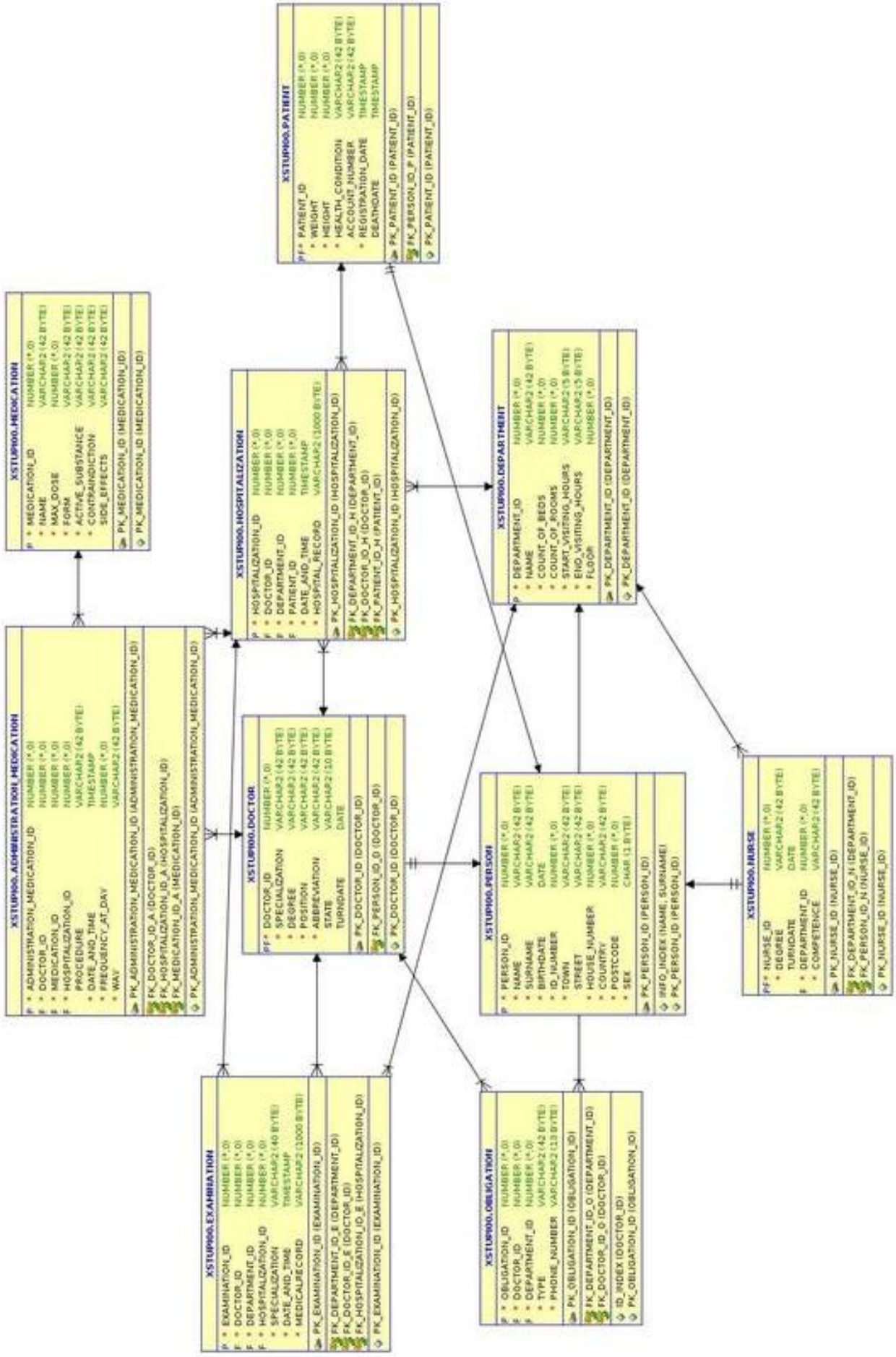

Figure A.3: **Scheme of the Relational database.**