Laboratory Exercises, Database Technology

Notice:

- The course has four compulsory laboratory exercises.
- You are to work in groups of two people. Sign up for the labs at `http://sam.cs.lth.se/Labs` (see the course plan for instructions).
- The labs are mostly homework. Before each lab session, you must have done all the assignments in the lab, written and tested the programs, and so on. Contact a teacher if you have problems solving the assignments.
- Smaller problems with the assignments, e.g., details that do not function correctly, can be solved with the help of the lab assistant during the lab session.
- Extra labs are organized only for students who cannot attend a lab because of illness. Notify Per Holm (`Per.Holm@cs.lth.se`) if you fall ill, *before* the lab.

The labs are about:

1. SQL usage.
2. Design and implementation of a database.
3. Development of a Java interface to the database in lab 2.
4. Lab 4 comes in two variants. You may choose one of:

   4a) For most of you: Development of a web interface (PHP) to the database in lab 2.

   4b) For those of you who know about graphs, are adventurous and used to solving problems on your own: Using a graph database (Neo4j). This lab was new last year (2013/14) and is still not well tested.

Also note:

- You need a MySQL account to do the labs (one account per group). See the course plan for information about where and when you can pick up your group's username and password.

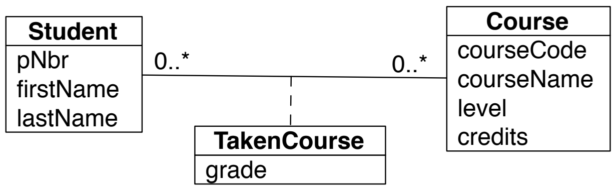
Lab 1 — SQL

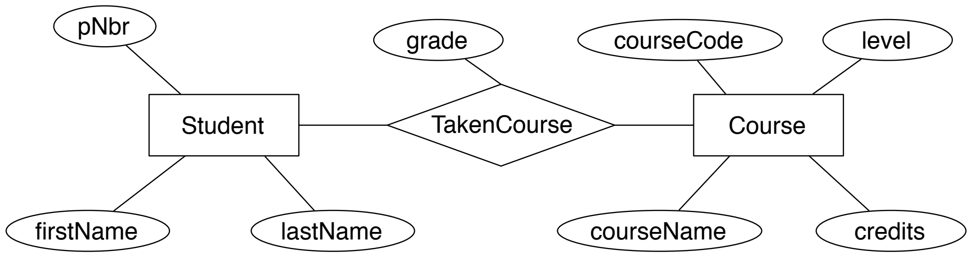
Objective: you will learn to write SQL queries and practice using the MySQL client `mysql`.

Background

A database for registration of courses, students, and course results has been developed. The purpose was not to develop a new Ladok database (Swedish university student database), so everything has been simplified as much as possible. A course has a course code (e.g., EDA216), a name (e.g., Database Technology), a level (G1, G2 or A) and a number of credits (e.g., 7.5). Students have a person number (Swedish civic registration number) and a name. When a student has passed a course his/her grade (3, 4 or 5) is registered in the database.

We started by developing an E/R model of the system (E/R stands for Entity-Relationship). This model is developed in the same way as when you develop the static model in object-oriented modeling, and you draw the same kind of UML diagrams. You may instead use traditional E/R notation, as in the course book. However, diagrams in the traditional notation take more paper space, and the notation is not fully standardized, so we will only use the UML notation. The model looks like this in the different notations (the UML diagram is at the top):





The E/R model is then converted into a database schema in the relational model. We will later show how the conversion is performed; for now we just show the final results. The entity sets and the relationship have been converted into the following relations (the primary key of each relation is underlined):

Students(pNbr, firstName, lastName)
Courses(courseCode, courseName, level, credits)
TakenCourses(pNbr, courseCode, grade)

Examples of instances of the relations:

pNbr	firstName	lastName
861103–2438	Bo	Ek
911212–1746	Eva	Alm
950829–1848	Anna	Nyström
...	...	...

courseCode	courseName	level	credits
EDA016	Programmeringsteknik	G1	7.5
EDAA01	Programmeringsteknik - fördjupningskurs	G1	7.5
EDA230	Optimerande kompilatorer	A	7.5
...	...	...	

pNbr	courseCode	grade
861103–2438	EDA016	4
861103–2438	EDAA01	3
911212–1746	EDA016	3
...	...	...

The tables have been created with the following SQL statements:

```
create table Students (
    pNbr      char(11),
    firstName varchar(20) not null,
    lastName  varchar(20) not null,
    primary key (pNbr)
);

create table Courses (
    courseCode char(6),
    courseName varchar(70) not null,
    level      char(2),
    credits    integer not null check (credits > 0),
    primary key (courseCode)
);

create table TakenCourses (
    pNbr       char(11),
    courseCode char(6),
    grade      integer not null check (grade >= 3 and grade <= 5),
    primary key (pNbr, courseCode),
    foreign key (pNbr) references Students(pNbr),
    foreign key (courseCode) references Courses(courseCode)
);
```

All courses that were offered at the Computer Science and Engineering program at LTH during the academic year 2013/14 are in the table `Courses`. Also, the database has been filled with invented data about students and their taken courses. SQL statements like the following have been used to insert the data:

```
insert into Students values('861103-2438', 'Bo', 'Ek');
insert into Courses values('EDA016', 'Programmeringsteknik', 'G1', 7.5);
insert into TakenCourses values('861103-2438', 'EDA016', 4);
```

Assignments

1. Study the relevant sections about SQL in the textbook (6.1–6.4 and 8.1). These sections contain more than you will use during this lab, so it can be a good idea to study assignment 3 in parallel.

2. Read the introduction to MySQL (see separate instructions).

3. Write SQL queries for the following tasks and store them in a text file. Format the SQL code according to the rules (`select` on one line, `from` on one line, `where` on one line, . . .). Don't use tabs to indent the SQL code — MySQL uses tabs for auto-completion of table names and attribute names.

   The tables `Students`, `Courses` and `TakenCourses` already exist in your database. If you change the contents of the tables, you can always recreate the tables with the following command (at the `mysql` prompt):

   ```
   mysql> source /usr/local/cs/dbt/makeLab1-utf8.sql
   ```

   After most of the questions there is a number in brackets. This is the number of rows generated by the question. For instance, [72] after question a) means that there are 72 students in the database.

   a) What are the names (first name, last name) of all the students? [72]
   b) Same as question a) but produce a sorted listing. Sort first by last name and then by first name.
   c) Which students were born in 1985? [4]
   d) What are the names of the female students, and which are their person numbers? The next-to-last digit in the person number is even for females. The MySQL function `substr(str,m,n)` returns `n` characters from the string `str`, starting at character `m`, the function `mod(m,n)` returns the remainder when `m` is divided by `n`. [26]
   e) How many students are registered in the database?
   f) Which courses are offered by the department of Mathematics (course codes FMAxxx)? [22]
   g) Which courses give more than 7.5 credits? [16]
   h) How may courses are there on each level G1, G2 and A?
   i) Which courses (course codes only) have been taken by the student with person number 910101–1234? [35]
   j) What are the names of these courses, and how many credits do they give?
   k) How many credits has the student taken?
   l) Which is the student's grade average (arithmetic mean, not weighted) on the courses?
   m) Same questions as in questions i)–l), but for the student Eva Alm. [26]
   n) Which students have taken 0 credits? [11]
   o) Which students have the highest grade average? Advice: define and use a view that gives the person number and grade average for each student.
   p) List the person number and total number of credits for all students. Students with no credits should be included with 0 credits, not `null`. If you do this with an outer join you might want to use the function `coalesce(v1, v2, ...)`; it returns the first value which is not null. [72]
   q) Is there more than one student with the same name? If so, who are these students and what are their person numbers? [7]

4. If you haven't picked up your MySQL account before the lab, the lab assistant will give you your username and password when you sign for it.

5. Log in to MySQL (see separate instructions). Change your MySQL password immediately.

6. Use `mysql` to execute the SQL queries that you wrote in assignment 3 and check that the queries give the expected results. Advice: open the file containing the queries in a text editor and copy and paste one query at a time, instead of writing the queries directly in `mysql`.

Lab 2 — Database Design

Objective: you will learn to design and to implement a database. This involves creating an E/R model for the application and converting the model into a relational model. You will also learn to create SQL tables and to insert data into the tables. During lab 3 you will develop a Java interface to the database, during lab 4 a web interface to the database.

Background

A database contains information about ticket reservations for movie performances. To make a reservation you must be registered as a user of the system. In order to register you choose a unique username and enter your name, address, and telephone number (the address is optional). When you use the system later, you just have to enter your username.

In the system, a number of theaters show movies. Each theater has a name and a number of (unnumbered) seats. A movie is described by its name only. (In a real system you would, naturally, store more information: actor biographies, poster images, video clips, etc.)

A movie may be shown several times, but then during different days. This means that each movie is shown at most once on any day.

You can only reserve one ticket at a time to a performance[1] and cannot reserve more tickets than are available at a performance. When you make a reservation you receive a reservation number that you will use when you pick up the ticket.

Assignments

1. Study the sections on E/R modeling and conversion of the E/R model to relations in the textbook (4.1–4.8).

2. Develop an E/R model for the database that is described above. Start by finding suitable entity sets in the system. For this, you may use any method that you wish, e.g., start by finding nouns in the requirements specification and after that determine which of the nouns that are suitable entity sets.

3. Find relationships between the entity sets. Indicate the multiplicities of the relationships.

4. Find attributes of the entity sets and (possibly) of the relationships. Consider which of the attributes that may be used as keys for the entity sets. Draw a UML diagram of your model.

5. Convert the E/R model to a relational model. Use the method that has been described during the lectures and in the textbook.

   Describe your model on the form Relation1(attribute, . . .), Relation2(attribute, . . .). Identify primary keys and foreign keys. About primary keys: a movie name and a date together suffice to identify a movie performance, since each movie is shown at most once on one day. This means that {movie name, date} is a key of the relation that describes performances. If you convert the E/R model according to the rules, it may happen that the key also contains the name of the theater. (That the name of the theater should not be a part of the key is indicated by the *functional dependency* movieName date $\rightarrow$ theaterName, which means that you can deduce the theater name if you know the movie name and the date. We will discuss functional dependencies later in the course.)

[1] If you want several tickets for the same performance you must make several separate reservations.

Additionally, relations must be *normalized* to avoid redundancy and anomalies in the database. We omit normalization for now, since we haven't discussed it yet. (Also, relations usually are reasonably normalized if you start with a good E/R model.)

6. Study the sections in the textbook (6.5–6.6) about SQL statements to modify and create tables. Also study sections 7.1–7.2, about key constraints.

7. Write SQL statements for the following tasks, and execute the statements in `mysql`:

   a) Create the tables. Don't forget primary keys and foreign keys. Insert data into the tables. Invent your own data with real-world movie names and theater names. Use the data type `date` for dates. Dates are entered and displayed on the form '2014–12–24'. Advice: write the SQL statements in a text file with the extension *.sql*. Execute the statements with the `mysql` command `source filename`. The file should have the following structure:

   ```
   -- Delete the tables if they exist. Set foreign_key_checks = 0 to
   -- disable foreign key checks, so the tables may be dropped in
   -- arbitrary order.
   set foreign_key_checks = 0;
   drop table if exists Users;
   ...
   set foreign_key_checks = 1;
   -- Create the tables.
   create table Users (
       ...
   );
   ...
   -- Insert data into the tables.
   insert into Users values(...);
   ...
   ```

   Note about MySQL: you may specify `check` constraints in the table definitions, but these are not enforced by MySQL.

   b) List all movies that are shown, list dates when a movie is shown, list all data concerning a movie performance.

   c) Create a reservation. Advice: unique reservation numbers can be created automatically by specifying the number column as an auto-increment column, like this:

   ```
   create table Reservations (
       nbr integer auto_increment,
       ...
       primary key (nbr)
   );
   ```

   When you insert rows into the table and don't give a value for the auto-increment column `nbr` (or specify it as `0` or `null`), it will be assigned the values 1, 2, ... The function `last_insert_id()` returns the last automatically generated value that was inserted into an auto-increment column.
   When a ticket is reserved a new row must be inserted into the reservation table, and the number of available seats for the performance must be updated. Before you do this you have to check that there are seats available for the performance. It is not easy to check this in pure SQL, so we save this for lab 3 when we write a graphical user interface to the database. Then, we code in Java and can use `if` statements.

8. Check that the key constraints that you have stated work as intended. Try to:
   - insert two movie theaters with the same name,
   - insert two performances of the same movie on the same date,
   - insert a performance where the theater doesn't exist in the database,
   - insert a ticket reservation where either the user or the performance doesn't exist,
   - . . .

9. Consider the following problem: when you make a ticket reservation you first check that seats are available for the performance, then you create a reservation, then update the number of available seats. Which problems can arise if several users do this simultaneously?

Lab 3 — Java Interface

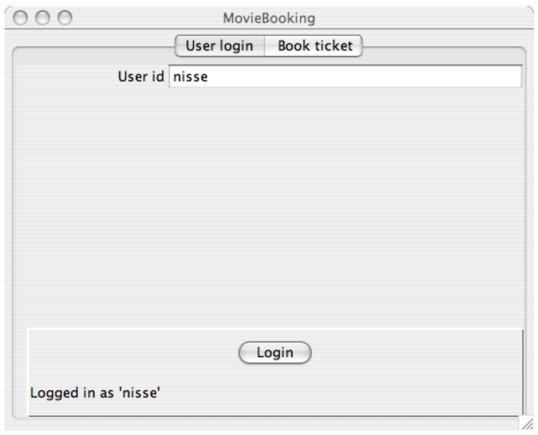
Objective: you will learn to use JDBC to communicate with a database from a Java program. You will also learn something about designing a graphical user interface with the Java Swing classes.

Background

A program which makes it possible to interactively make ticket reservations[2] for movie performances uses the database which you developed during lab 2. The program has a graphical user interface.

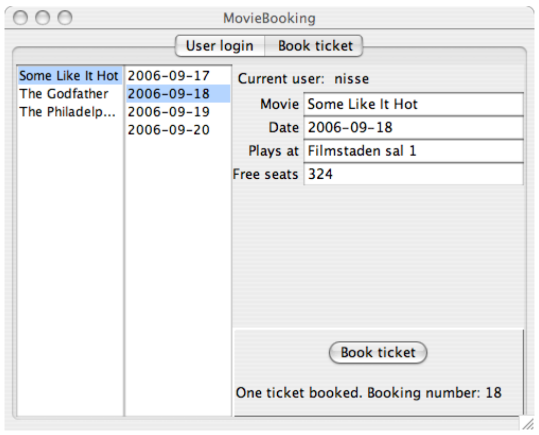
The program is a stand-alone application, and users must have the application installed on their own computers. It would be preferable if users could make ticket reservations over the web. In lab 4, you will develop a PHP application which makes this possible.

The user interface for the program looks like this:



User id nisse

Logged in as 'nisse'



Some Like It Hot
The Godfather
The Philadelp...

2006-09-17
2006-09-18
2006-09-19
2006-09-20

Current user: nisse
Movie Some Like It Hot
Date 2006-09-18
Plays at Filmstaden sal 1
Free seats 324

One ticket booked. Booking number: 18

The window has two tabs: User login and Book ticket. The first tab is used when a user[3] logs in to the system with his or her username, the second tab is used to make ticket reservations.

The reservation tab has two lists. In the left list are the names of movies currently showing. When you select a movie, the performance dates are shown in the right list. When you select a date, information about the selected performance is displayed in the text fields. When you click the Book ticket button a reservation for the performance is made, if there are available seats. You receive an error message if there are no available seats.

When you make a reservation you receive a reservation number. The number of available seats is updated on each reservation.

Assignments

1. Read about JDBC in the textbook (section 9.6), and in the overhead slides. Links to further information about JDBC are on the course homepage.

2. Large parts of the programs (classes) needed in the system are already written: a main program (*MovieBooking.java*), and the user interface (*MovieGUI.java* and other files). The classes are in the file */usr/local/cs/dbt/lab3.tar.gz*, also available on the course web. This file is an archived Eclipse project.

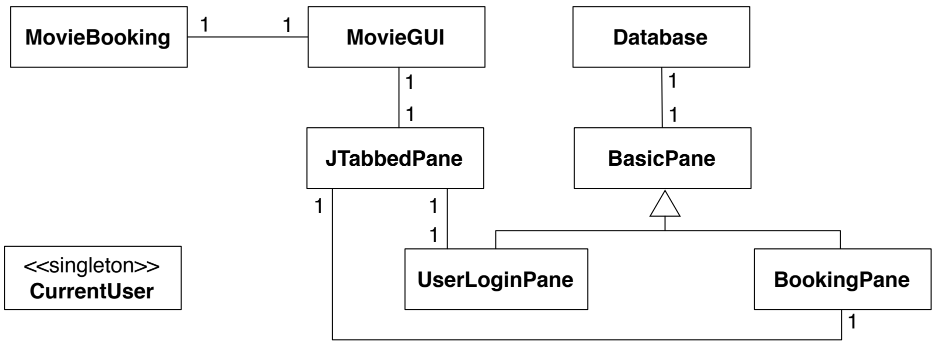
[2] The program only handles new reservations. All other tasks concerning the database, e.g., creation of new performances and creation of new users, are performed by other programs. In your case, you will use the command line client `mysql` to perform such tasks.
[3] Note: the "user" here is the user of the ticket reservation system, who chose a username when he or she registered in the system. Do not confuse this user with the database user, who must log in to the database system with the MySQL username and password.

- If you use Eclipse: import the project file (Import > General > Existing Projects into Workspace).
- If you don't use Eclipse: change to an appropriate directory[4] and unpack the file:

```
tar xzf /usr/local/cs/dbt/lab3.tar.gz
```

3. Your task is to complete the program by filling in the empty action-handling methods in the listener classes.

   The classes in the program are shown below, in a UML diagram. The diagram is schematic and only shows the most important classes and associations. Study the pictures of the user interface (under Background, above), and compare them with the following description:



`MovieBooking`	is the main program.
`MovieGUI`	is the main class for the user interface.
`JTabbedPane`	is one of the Swing standard classes. It describes a Swing component that may contain several tabs.
`BasicPane`	is a general description of a tab in the program.
`UserLoginPane`	is the "left tab", which is used when a user logs in to the system.
`BookingPane`	is the other tab, which is used when the user makes a reservation of a ticket for a movie performance.
`Database`	handles all communication with the database system.
`CurrentUser`	is a singleton class, which keeps track of the user that has logged in to the system. It is used by the classes `UserLoginPane` and `BookingPane`.

An object of the class `BasicPane` divides the available window area into two areas: left and right. The right-part is further divided into three areas: top, middle, and bottom.

In the login tab the left area is empty. The top area contains the text "Username" and the text field where the user enters the username. The bottom area contains the login button and a message line.

In the reservation tab the left area contains two lists: one for movie names and one for performance dates. The top area contains labels and text fields used to display information about a movie performance. The bottom area contains the reservation button and a message line.

4. The program shall perform the following tasks:

   a) When you click the Login button: log in with the specified username.

   b) When you enter the reservation panel: show movie names in the movie name list.

[4] `tar` creates a new directory (here *lab3*) with the lab files in the current directory.

c) When you select a movie name: show performance dates for the movie in the date list.

d) When you select a performance date: show all data concerning the performance in the text fields.

e) When you click Book ticket: make a ticket reservation. Two users that make reservations simultaneously must not interfere with each other (the code must be transaction safe).

Consider which tasks that have to be performed in the database for each of the tasks a–e above.

JDBC calls are used for the communication between the program and the database system. You should not have a tight coupling between the user interface and the database communication, so you must collect all JDBC calls in a class `Database`. Parts of this class are already written.

Specify further methods in the class `Database` to perform the tasks a–e. Do not implement the methods yet. Advice: when you are to show information concerning a performance (task d), you have to fetch the information from the database. Do this by creating and returning an object of a class `Performance`, which has the same attributes as the corresponding table in the database.

5. When you select a movie name in the name list, the method `valueChanged` in the class `NameSelectionListener` (in the class `BookingPane`) is called (task c in assignment 4):

```
public void valueChanged(ListSelectionEvent e) {
    if (nameList.isSelectionEmpty()) {
        return;
    }
    String movieName = nameList.getSelectedValue();
    /* --- insert own code here --- */
}
```

Replace the comment with the appropriate Java code. The code will call one of the `Database` methods; implement that method.

6. Other methods, similar to `valueChanged`, must also be written.[5] Do this, and implement the corresponding methods in the class `Database`.

To fetch a date column from a result set, use the method `getString()` (all you want to do with the date is to display it).

7. Compile and test the program. Don't forget to test the case when you try to reserve a ticket for a fully-booked performance; in that case you should receive an error message.

When the program is executed it needs access to the MySQL JDBC driver (Connector/J, the class `com.mysql.jdbc.Driver`), which is in the file *mysql-connector-java-5.1.27-bin.jar* in *lab3*. If you use Eclipse, the build path for the project *lab3* is set to include this file. If you don't use Eclipse, you must set `CLASSPATH` as shown below. Then, the program may be executed with `java MovieBooking`.

```
export CLASSPATH=.:/path-to-mysql-connector-java-5.1.27-bin.jar
```

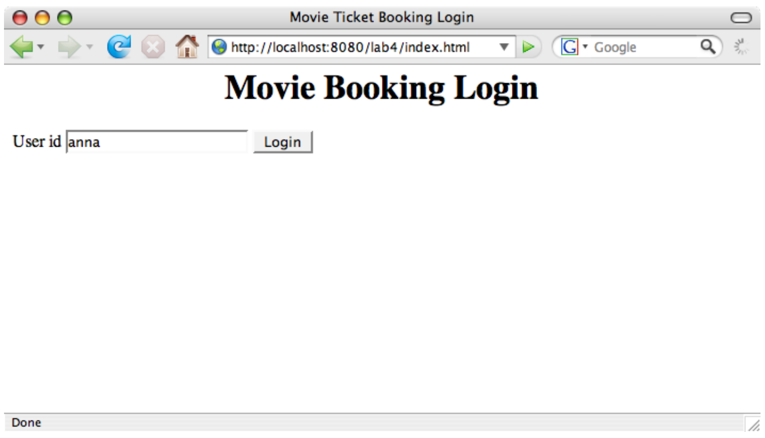
[5] The files *MovieGUI.java*, *UserLoginPane.java*, *BookingPane.java*, and *Database.java* must be modified. The places where you have to make changes are marked with comments starting with `/* ---`.

Lab 4a — PHP

Objective: you will get an introduction to PHP. You will also learn to use forms in HTML.

Background

Look at the series of screenshots of a web browser window below. As you see, a ticket for a movie performance is reserved, just as in the application that you developed during lab 3. Unlike lab 3, tickets are reserved over the web, which makes it possible for users at any location to reserve tickets, as long as they have access to a web browser.

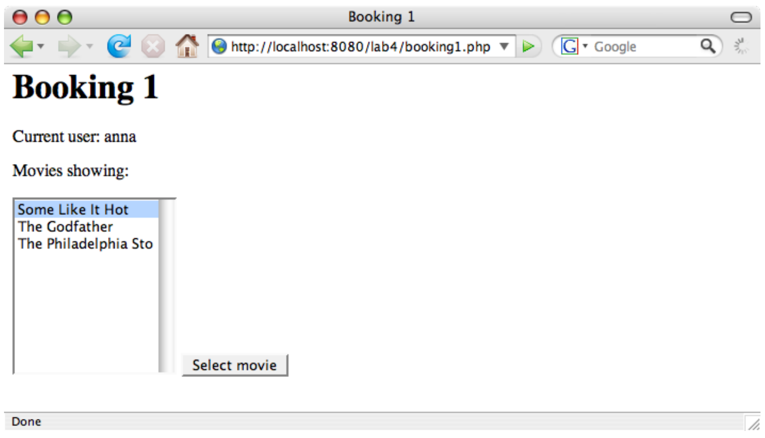


Movie Ticket Booking Login

http://localhost:8080/lab4/index.html

Movie Booking Login

User id anna



Booking 1

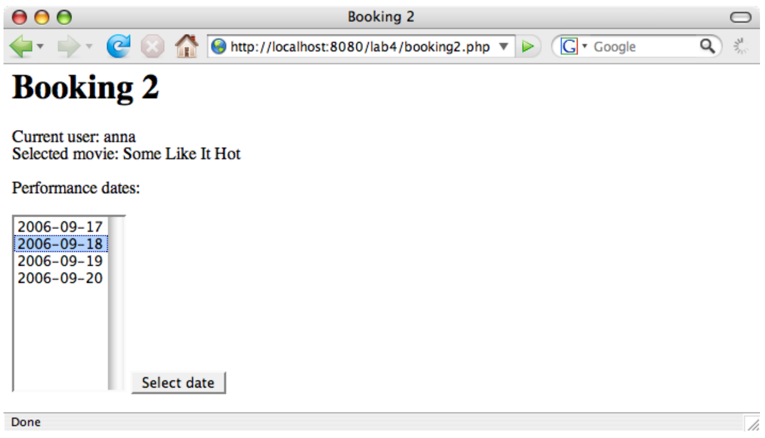
http://localhost:8080/lab4/booking1.php

Booking 1

Current user: anna

Movies showing:

Some Like It Hot
The Godfather
The Philadelphia Sto



Booking 2

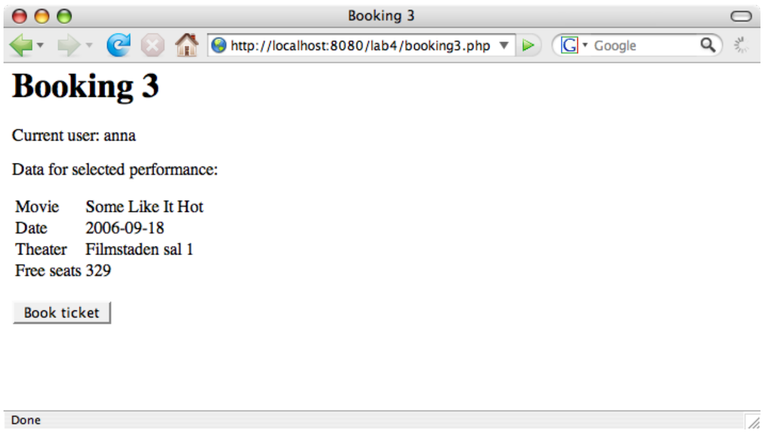
http://localhost:8080/lab4/booking2.php

Booking 2

Current user: anna
Selected movie: Some Like It Hot

Performance dates:

2006-09-17
2006-09-18
2006-09-19
2006-09-20

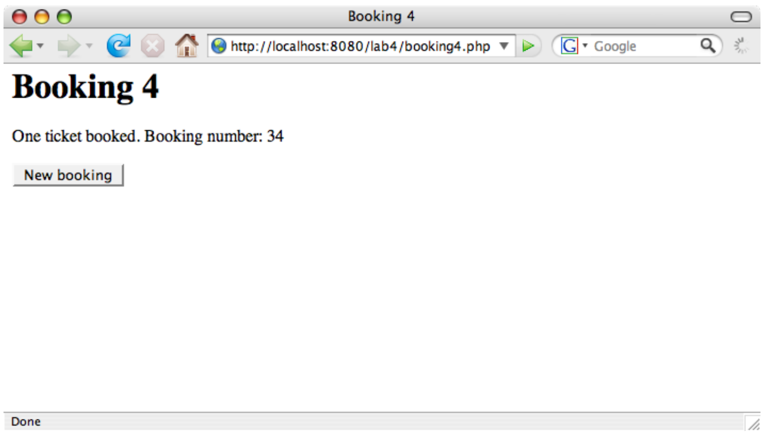


Booking 3

Current user: anna

Data for selected performance:

Movie	Some Like It Hot
Date	2006-09-18
Theater	Filmstaden sal 1
Free seats	329



Booking 4

One ticket booked. Booking number: 34

The dialog with the user could be much more user-friendly and elegant. You are free to design a better interface, if you wish.

Assignments

Your task is to write the PHP programs that are necessary to implement a web-based ticket reservation system, as shown in the screenshots starting on the facing page. You will use PHP's built-in web server and the MySQL installation on `puccini.cs.lth.se`.

To be able to do this lab you have to master, in varying degree, the following:

- "Ordinary" HTML.
- Forms in HTML.
- Basic PHP programming, session handling in PHP, using MySQL from PHP.

Ordinary HTML and forms in HTML

See the overhead slides for the web server lecture. You can also find HTML tutorials on the web, for example:

```
http://www.htmlcodetutorial.com/
http://www.htmlcodetutorial.com/forms/
```

Web servers

To use PHP you need a "php-enabled" web server. Apache (`http://www.apache.org`) is a common choice in Linux/Unix environments. To correctly configure an Apache server is not an easy task, but there are many pre-configured "LAMP" packages that include Apache, MySQL and PHP (WAMP for Windows and MAMP for Macintosh).

If all you wish to do is to test your PHP programs it is easier to use the built-in web server in PHP (available from PHP version 5.4). This server is explicitly for development and testing; it should *not* be used in a production environment.

Starting and Testing the Web Server

1. A web server has a "document root", a directory that contains the top-level files and directories that the server sees. In this lab, the root directory is *phproot*. It contains the files *index.html*, *phpinfo.php* and *connect.php* that are used to check the web server configuration (see below). `demo1`, `demo2` and `demo3` are directories containing PHP examples (see below).

   Create the *phproot* directory (the *.gz* file is also available on the course web):

   ```
   tar xzf /usr/local/cs/dbt/phproot.tar.gz
   ```

2. Start the built-in web server:

   ```
   cd phproot
   php -S 127.0.0.1:8080
   ```

   It should be possible to use `localhost` instead of `127.0.0.1`, but for some reason this doesn't work on the student computers.

3. Open a web browser and go to:

   ```
   http://localhost:8080/index.html
   ```

   You will reach the start page (the same as the standard Apache start page), which only contains the text `It works!`. Also check the pages `http://localhost:8080/phpinfo.php` (gives information about the PHP module) and `http://localhost:8080/connect.php` (checks that the connection to the MySQL server on Puccini works).

PHP programming, examples

1. Study the overhead slides for the web server lecture and section 9.7 in the textbook. More information about PHP is all over the web. Examples (`http://www.php.net` is the official PHP site):

   ```
   http://en.wikibooks.org/wiki/Programming:PHP
   http://www.w3schools.com/php/
   http://www.php.net/docs.php
   ```

   As a first example, we will study a PHP program which computes square roots. The first page of the application is a static HTML page. When you enter a number and press the submit button, the PHP program *roots.php* is called. The PHP program computes the square root of the number and returns a dynamic HTML page containing the result.

The static HTML start page (*demo1/index.html*) looks like this:

```
<html>
<head><title>Square Roots</title></head>
<body>
<h1 align = "center">Fill in some data</h1>

This program works out square roots.
<p>

<form method = "get" action = "roots.php">
    <input type = "text" name = "number">
    <input type = "submit" value = "Compute root">
</form>
</body>
</html>
```

The PHP program is in *demo1/roots.php*. It looks like this:

```
<html>
<head><title>Square Root Results</title><head>
<body>

<?php
    $number = $_REQUEST['number'];
    if (is_numeric($number)) {
        if ($number >= 0) {
            print "The square root of $number is ";
            print sqrt($number);
        } else {
            print "The number must be >= 0.";
        }
    } else {
        print "$number isn't a number.";
    }
?>

<p>
Try another:
<p>

<form method = "get" action = "roots.php">
    <input type = "text" name = "number">
    <input type = "submit" value = "Compute root">
</form>
</body>
</html>
```

`$_REQUEST` is an associative array which contains the parameters to the HTTP request.

2. Visit `http://localhost:8080/demo1/index.html` and check that the application works properly.

3. Change something in the PHP program, check that your changes have taken effect.

4. Many web applications need to store data on the server between accesses to different web pages. The data must be kept separate for each user. PHP uses *sessions* for this purpose. Sessions are implemented with cookies containing a "session id", so you must allow cookies in your browser.

In a PHP program, session data is kept in the associative array `$_SESSION`. The program *demo2/roots.php* is almost the same as the program in *demo1*, except that it remembers and prints the number of root computations. The beginning of the program looks like this:

```
<?php
    session_start();
    $_SESSION['computationNbr']++;
?>

<html>
<head><title>Square Root Results</title></head>
<body>
<?php
    $computationNbr = $_SESSION['computationNbr'];
    print "Root computation number $computationNbr<p>";
    $number = $_REQUEST['number'];
    ... same as before
```

`session_start()` starts or restores a session. It must be called before anything is sent to the client, i.e., before the `<head>` tag. `$_SESSION['computationNbr']` is initialized to 0 in *index.php*.

Go to `http://localhost:8080/demo2/index.php` and check that this program also works as expected.

5. We will use the PDO (PHP Data Objects) package to communication with the MySQL server on Puccini. Start by creating and populating a table PersonPhones in your database on Puccini. Do the following at the `mysql` prompt:

```
create table PersonPhones (
    name varchar(20),
    phone varchar(20),
    primary key (name)
);
insert into PersonPhones values('Alice', '123456');
...
```

6. In *demo3/getpersondata.php* there is a PHP program which fetches all the data from the PersonPhones table:

```
<?php
$host = "puccini.cs.lth.se";
$username = "xxx";
$password = "yyy";
$database = "xxx";

$conn = new PDO("mysql:host=$host;dbname=$database", $username, $password);
$conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

$stmt = $conn->prepare("select * from PersonPhones order by name");
$stmt->execute();
$result = $stmt->fetchAll(PDO::FETCH_ASSOC);
?>

<html>
<head><title>PHP PDO Test</title><head>
<body><h2>Data from the PersonPhones table</h2>

<table border=1>
```

```
<tr><th>Name</th><th>Phone</th></tr>
<?php
$rowcount = 0;
foreach ($result as $row) {
    $rowcount++;
    print "<tr>";
    foreach ($row as $attr) {
        print "<td>";
        print htmlentities($attr);
        print "</td>";
    }
    print "</tr>";
}

?>
</table>

<p>
A total of <?php print "$rowcount"; ?> rows.
</body>
</html>
```

The function `htmlentities` converts characters that have special significance in HTML. For example, it converts `'&'` (ampersand) to `'&amp;'`.

7. Change the username, password and database to your login data. Direct your browser to `http://localhost:8080/demo3/getpersondata.php` and check that the program works.

Write the ticket reservation programs

1. Write the PHP programs necessary to implement the ticket reservation system. The user interface should look like the screenshots on pages 12–13 (or better if you know more PHP and HTML). Some of the necessary programs are already written (they are in *phproot/ lab4*):

**.html*	Finished.
database.inc.php	A class `Database` which contains the calls to the MySQL server. Built along the same lines as the Java class `Database` in lab 3. You must change some of the functions and add new functions.
login.php	Creates the database object, connects to the server, checks that the user is registered in the database, redirects the browser to *booking1.php*.
mysql_connect_data.inc.php	Contains the host name, username, password and database name that are necessary to login. Change these to your data. — This information is sensitive, so it really should be kept outside of the *htdocs* tree.
booking1.php	Shows the "Booking 1" screen.

2. Complete the program by writing *booking{2,3,4}.php*. Test (start at *lab4/ index.html*).

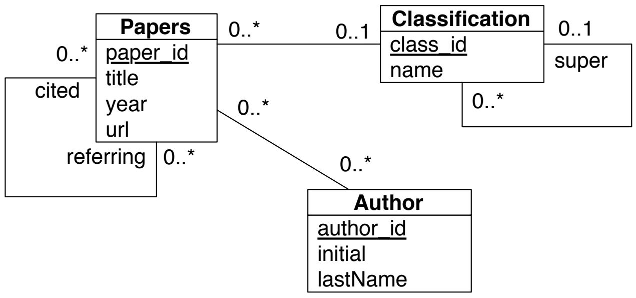
Lab 4b — Graph Databases

Objective: you will get an introduction to graph databases and learn to solve simple problems using a graph database.

Background

Consider the following: scientific papers have authors (often more than one). Most papers have a classification (what the paper is about). The classifications form a hierarchy in several levels (for example, the classification "Databases" has the sub-classifications "Relational" and "Object-Oriented"). A paper usually has a list of references, which are other papers. These are called citations.

This is described in the following E/R diagram:



It is straightforward to translate this diagram into a relational model:

Papers(paper_id, title, year, url, *class_id*)
Classifications(class_id, name, *super_id*)
Citations(*referring_id*, *cited_id*)
Authors(author_id, initial, lastName)
AuthoredBy(*paper_id*, *author_id*)

It is also straightforward to answer many queries about the papers. Examples:

Find all papers by a given author:

```
select title, year
from Authors natural join AuthoredBy natural join Papers
where lastName = 'Anderson' and initial = 'A';
```

Find the ten papers with the largest number of citations:

```
select title, year, count(*)
from Papers join Citations on paper_id = cited_id
group by title, year
order by count(*) desc
limit 10;
```

Most interesting queries will (naturally) involve one or more joins. In a large database, these can be costly. Also, there are queries which cannot be answered in classical SQL. An example is "Does paper A cite paper B? If not directly, does paper A cite a paper which in its turn cites paper B? And so on, in several levels." A query like this requires many joins, and furthermore the number of joins isn't known beforehand.

An even simpler example is "Print the full classification of a paper (for example, Databases / Relational)". Since the number of levels of the classification hierarchy is not known, you cannot write one single SQL query for this.

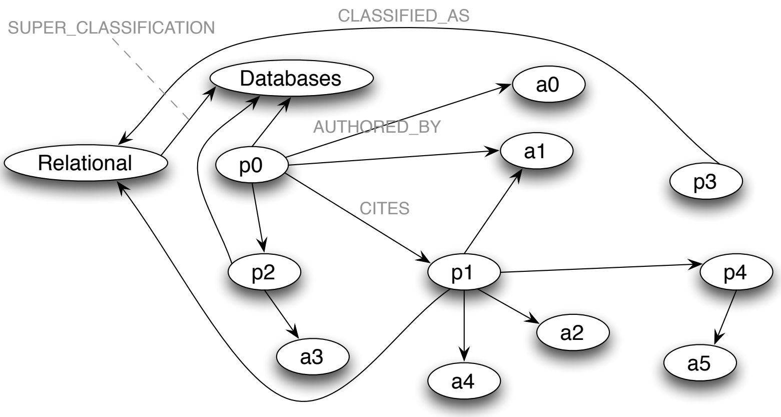


Figure 1: Nanocora, a small paper graph.

Alternatively, the data about papers and authors can be stored in a *graph database*. In a graph database, the data about the entities is stored in *nodes* — paper data in paper nodes, author data in author nodes, etc. The connections between the nodes are described by *relationships*. A relationship starts in one node and ends in a node, can be directed or bi-directional, has a type, and can carry data.

Figure 1 shows Nanocora, a small scientific paper database (five papers `p0`–`p4`, six authors `a0`–`a5`, classifications `Databases` and `Databases / Relational`). All relationships are directed, but they can be traversed in both directions. We see that paper `p0` is written by authors `a0` and `a1`, and that `p0` cites papers `p1` and `p2`. Paper `p3` has no author, paper `p4` is unclassified.

You can see that this representation ought to be advantageous in many cases: from a paper node you can immediately reach the nodes describing the authors of the paper, and vice versa.

There must also exist a facility to find nodes in the graph. Normally, this is done with indexes — for example, there could be indexes on paper titles and on author names. Note that the purpose of indexes isn't to speed up traversals, just to find starting points in a graph.

Neo4j

Neo4j, `http://www.neo4j.org`, is a very popular open-source graph database. Examples of use cases are social applications, recommendation engines, fraud detection, resource authorization, network and data center management and much more. A Neo4j database may be distributed among several computers and may contain billions of nodes and relationships.

Neo4j is written in Java and can be used either as an embedded database or as a server. There are API's at different levels: the "core" API to access nodes and relationships, the "traversal" API to traverse graphs, and also a proprietary query language, Cypher.

Clients communicate with Neo4j servers through a REST API (queries and updates are sent via HTTP GET or POST requests, and data is packaged in JSON objects).[6] There are API's for different languages that hides this complexity from the user.

Neo4j Programming

In this section, some simple examples (in Java) of using the core and traversal API's are given. There is a Javadoc description of the API's at the Neo4j API documentation site, `http://api.neo4j.org`. We assume that an Neo4j server is available and that the embedded Nanocora database has been created. (We only cover database queries here, not database updates.)

[6] A good overview of REST services is in `http://www.restapitutorial.com`.

Nodes and Relationships

Nodes in a Neo4j database have *properties*. A property has a name (a key) and a value. A property value must be a primitive (scalar or string) or an array of primitives. For example, a paper node has the properties title (string), year (integer), and url (string). Different nodes of the same kind need not have the same properties and the presence of a property can be queried by an application. Note that there is no need for artificial keys like `paper_id`.

Naturally, nodes can be wrapped in domain classes, but in these examples we work with the nodes directly.

The database has an index on the paper titles. Here's how to find a paper node via the paper index and fetch the publishing year:

```
String paperTitle = "p1";
Index<Node> paperIndex = db.index().forNodes("paperIndex");
Node paper = paperIndex.get("title", paperTitle).getSingle();
int year = paper.getProperty("year");
```

Relationships have a type, a start node and an end node, and can have properties (relationship properties are not used in the paper database). Here's how to follow relationships to find the authors of a paper:

```
Node paper = ...;
for (Relationship r : paper.getRelationships(RelTypes.AUTHORED_BY)) {
    Node author = r.getEndNode();
    System.out.println(author.getProperty("lastName") + ", "
            + author.getProperty("initial"));
}
```

The method `getRelationships` can have a second argument which indicates the direction of the relationship (`Direction.OUTGOING` or `Direction.INCOMING`).

A Complete Program

The following is a complete program that uses an embedded database *nanocora.db*. It prints the data about a paper, including its authors (but not the classification).

```
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.Relationship;
import org.neo4j.graphdb.Transaction;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;
import org.neo4j.graphdb.factory.GraphDatabaseSettings;
import org.neo4j.graphdb.index.Index;
import common.RelTypes;

public class PrintPaperData {
    public static void main(String[] args) {
        // The title of the paper.
        String paperTitle = "p1";

        // Open the database read only.
        String DB_PATH = "nanocora.db";
        GraphDatabaseService db = new GraphDatabaseFactory()
                .newEmbeddedDatabaseBuilder(DB_PATH)
                .setConfig(GraphDatabaseSettings.read_only, "true")
                .newGraphDatabase();
        registerShutdownHook(db);
        // Wrap the database operations in a transaction.
        try (Transaction tx = db.beginTx()) {
            // Database operations.
            Index<Node> paperIndex = db.index().forNodes("paperIndex");
            Node paper = paperIndex.get("title", paperTitle).getSingle();
            System.out.println(paperTitle + ", " + paper.getProperty("year"));
            System.out.print("    Authors: ");
            for (Relationship r : paper.getRelationships(RelTypes.AUTHORED_BY)) {
                Node author = r.getEndNode();
                System.out.print(author.getProperty("lastName") + ","
                        + author.getProperty("initial") + " / ");
            }
            System.out.println();
            // Mark the transaction as successful. The transaction will be
            // committed when it's closed. Use tx.failure() to roll back
            // the transaction.
            tx.success();
        } catch (Exception e) {
            e.printStackTrace();
        }
        db.shutdown();
    }

    private static void registerShutdownHook(final GraphDatabaseService db) {
        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                db.shutdown();
            }
        });
    }
}
```

The class `Transaction` implements the `AutoCloseable` interface so the transaction is automatically closed after the actions in the "try with resources" statement. (This is new in Neo4j version 2; earlier the transactions had to be explicitly closed with `tx.finish()`.)

Indexes

The database has indexes on paper titles ("paperIndex"), author last names ("authorIndex") and classification names ("classificationIndex"). Neo4j uses Apache Lucene Core (`http://lucene.apache.org`) as the search engine. Lucene has very advanced facilities for searching; here we use only searching for exact strings and searching using wildcards (*). When wildcards are used, the search string must not contain any spaces.

Use the paper index to find the papers with titles starting with "p":

```
String searchString = "p*";
...
Index<Node> paperIndex = db.index().forNodes("paperIndex");
for (Node p : paperIndex.query("title", searchString)) {
    System.out.println(p.getProperty("title"));
    ...
}
```

Traversals

The classes in the traversal API aid in traversing a graph. A traversal starts in a node and may be performed breadth-first or depth-first. There are built-in routines for more complex algorithms (such as shortest path).

The details of how a traversal should be performed are collected in an object of the class `TraversalDescription`. During traversal, each *path* is reported to the caller. A path is a series of nodes and relationships.

In the following example the part of a graph reachable from a specific node is traversed and each path is printed. (Normally, you wouldn't traverse an entire graph.)

```
Node p = ...;
TraversalDescription td = Traversal.description();
for (Path path : td.traverse(p)) {
    System.out.println(path);
}
```

Traversals become useful only when they are restricted. The following restrictions can be added to a traversal description:

Relationships Specify that only relationships of a specific type (and direction) should be traversed. Example:

```
td = td.relationships(RelTypes.AUTHORED_BY);
```

Evaluators Decide if a node in a path should be included in the result, and if the traversal should continue. You can write your own evaluators, but often you only need the evaluators from the class `Evaluators`. Example:

```
td = td.evaluator(Evaluators.toDepth(5))
        .evaluator(Evaluators.excludeStartPosition());
```

Order Specify the traversal order. You can write your own order specifications, but usually you only need to specify depth first or breadth first order. Example:

```
td = td.depthFirst();
```

As an example, here is how to find the authors of a paper, using the traversal API. Note that the traversal depth is set to exactly 1, which is very uncommon.

```
TraversalDescription td = Traversal.description()
        .relationships(RelTypes.AUTHORED_BY, Direction.OUTGOING)
        .evaluator(Evaluators.atDepth(1));
Node paper = ...;
System.out.println(paper.getProperty("title") + ", "
        + paper.getProperty("year"));
System.out.print("    Authors: ");
for (Path path : td.traverse(paper)) {
    Node author = path.endNode();
    System.out.println(author.getProperty("lastName") + ", "
            + author.getProperty("initial"));
}
System.out.println();
```

Graph Algorithms

You can solve most common graph problems using the traversal API, but there are specialized algorithms for special problems. One such problem is to find a path (most often the shortest path) between two nodes. The shortest path between two authors, using "authored by" relationships in both directions to a maximum level of 5, is found in the following way:

```
Node author1 = ...;
Node author2 = ...;
PathFinder<Path> finder = GraphAlgoFactory.shortestPath(
        Traversal.expanderForTypes(RelTypes.AUTHORED_BY), 5);

Path path = finder.findSinglePath(author1, author2);
for (Node n : path.nodes()) {
    if (n.hasProperty("title")) {
        System.out.print(" --> " + n.getProperty("title") + " <-- ");
    } else {
        System.out.print(n.getProperty("lastName"));
    }
}
System.out.println();
```

Here is the path between authors `a0` and `a2` in the Nanocora database:

```
a0 --> p0 <-- a1 --> p1 <-- a2
```

Using the REST Interface

The same program as in the section A Complete Program, page 20, with some small modifications, can be used to access a database that is managed by a server. The most important difference is how the database service is created (assuming that the server is running on `puccini.cs.lth.se`):

```
String DB_PATH = "http://puccini.cs.lth.se:7474/db/data";
GraphDatabaseService db = new RestGraphDatabase(DB_PATH);
```

Another difference is that the REST API doesn't support transactions. In the Java REST API, transactions are represented by a class `NullTransaction`, in which the transactional methods have no effect. In the current implementation this class does not implement `AutoCloseable`, so it cannot be used in a "try with resources" statement. The easiest workaround is to not use transactions at all in programs that communicate via the REST interface.

Finally, properties of type character somewhere lose their type information — they are treated as integers. In the Cora database this means that author initials must be printed in this complicated fashion:

```
System.out.print((char)((Integer) author.getProperty("initial")).intValue());
```

The Paper Data — Cora

The "real" paper data is from Cora, a research project on "domain-specific search engines over computer science research papers". It is described in "Automating the Construction of Internet Portals with Machine Learning", by Andrew McCallum, et al, Information Retrieval Journal, volume 3, 2000, pp. 127–163. The raw data can be downloaded from `http://people.cs.umass.edu/~mccallum/data.html` (the link Cora Research Paper Classification).

The raw data has been processed to make it more uniform. For example, papers without title have been assigned a title (the last part of the URL), citations of unknown papers have been removed, etc. Some obvious errors have also been corrected, for example in the paper where the 100+ words of the abstract had been taken as the list of authors.

The database contains approximately 25,000 authors, 37,000 papers and 220,000 relationships. It is available as a Neo4j graph database (on Puccini, port 7474) and also as a relational MySQL database (on Puccini, database `cora`, tables as shown on page 18). The MySQL database can be used to find starting points in the graph database and to check results from Neo4j queries.

Assignments

1. Familiarize yourself with the Neo4J manual, `http://neo4j.com/docs/2.0.2/`, read chapter 3 and sections 32.2–32.3, 32.7, and 32.9. Also look at the API documentation at `http://neo4j.com/api_docs/2.0.0/`.

2. The file */usr/local/cs/dbt/neo-projects.tar.gz*, also available on the course web, contains five Eclipse projects. Import it into Eclipse (Import > General > Existing Projects into Workspace). The following projects are created:

`cora`	Example programs.
`jackson`	JSON processor.
`jersey`	RESTful Web Services in Java.
`neo-lib`	The Neo4J library.
`neo-rest-graphdb`	The Neo4j REST framework.

3. The `cora` project contains these packages:

`common`	Definitions of label types and relationship types.
`nanocora`	Programs using the Nanocora embedded database.
`restcora`	Programs communicating with the server managing the Cora database.

   Study the class `nanocora.PopulateNanoCora`, run it. It creates and populates the Nanocora database in the current project (the directory *nanocora.db*; it will not show up in Eclipse until you refresh the workspace).

4. The "Complete Program" (page 20) is in the class `PrintPaperData`, both in embedded (`nanocora`) and server (`restcora`) versions. Study the programs, note the differences. Run the programs.

5. The class `nanocora.PrintPaperAuthors` prints the authors of selected papers (papers with titles starting with "p"). Write a similar program for the Cora database (for papers with titles starting with "Generating").

6. Solve some of the problems in the list below (choose the ones that you find most interesting). Test your programs on the Nanocora database first, then convert them to the Cora database.

   a) Modify `PrintPaperAuthors` to print also the full classification of each paper.

   b) Print the entire classification tree:

```
Artificial Intelligence
    Expert Systems
    ...
    NLP
    Machine Learning
        Probabilistic Methods
        ...
        Case-Based
Data Structures  Algorithms and Theory
    Randomized
   ...
```

c) Donald Knuth is a well-known computer scientist. Print the names of authors with whom he has cooperated (94 of them, from Aho to Zhu).

d) Modify the program from assignment a) to print not only the authors and classification of each paper, but also the titles of papers that are cited by the paper and titles of papers that cites the paper.

e) "Introduction to Algorithms" is a fundamental paper. Print the "citation path" to this paper from the paper "An algebraic semantics of Basic Message Sequence Charts" (6 hops).

f) Find something interesting to do on your own.