OPTIMIZING DATABASE ACCESS USING CONNECTION POOLING IN MYSQL SERVER AND APACHE TOMCAT

Andrei MĂCIUCĂ[1], Ioana BRĂNESCU RASPOP[2], Simona DUMITRESCU[3], Gelu IONESCU[4], Dan POPESCU[5]

Articolul descrie modul în care metoda "connection pooling" optimizează accesul la baza de date. Pentru a demonstra faptul că metoda într-adevăr optimizează modul în care cerinţele sunt prelucrate, vom realiza un experiment care va conţine două scenarii, unul în care se foloseşte "connection pooling" şi un altul în care nu se foloseşte. La finalul experimentului, valorile obţinute vor demonstra că accesul la baza de date este îmbunătăţit semnificativ.Un studiu de caz este prezentat intr-o aplicatie cu scop medical.

This paper describes how connection pooling optimizes database access. In order to prove that connection pooling really optimizes how requests are handled, we will make an experiment with two scenarios, one with connection pooling and one without this method. At the end, the values obtained will show that database access is greatly increased. A case study is presented for a medical specific application.

Keywords: connection pooling, server, optimization, WAPT client, immune system, biological database

1. Introduction

Connection pooling represents a technique of initiating and managing a pool of connections. These connections are always ready for use by any thread that needs them. The connection pooling is used in enterprise and web-based applications [1]. Given the huge amount of user interactions, about millions for customer facing applications, the finite server side resources need to be optimally managed. These resources are represented by databases, message queues,

[1] PhD student, Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: andrei.maciuca@gmail.com
[2] PhD student, Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: ioana.branescu@gmail.com
[3] PhD student, Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: simona_roxana_robeci@yahoo.com
[4] Prof., GIPSA-lab, 11 rue des Mathématiques, Grenoble Campus BP46, F-38402 SAINT MARTIN D'HERES CEDEX, email: gelu.ionescu@gipsa-lab.grenoble-inp.fr
[5] Prof., Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: dan_popescu_2002@yahoo.com

enterprise systems, each of these being accessed by an application using a connection object that represents the resource entry point. The way access is managed to these shared resources is essential for meeting high performance requirements in J2EE applications [2], [3].

At any given time, a connection object is involved in one of the following major steps of its lifecycle: creation, initialization, ready for use, destruction and garbage collection. Al of these steps, excepting ready for use, require a lot of memory and processing time. As a result, if creating and holding an object in memory requires fewer resources than creating and destroying it, holding the object in memory and using it when needed is the optimal solution [4], [5].

There are several benefits when using connection pooling. There is a reduced connection creation time, a simplified programming model (each individual thread can act as it has created its own connection, allowing to use straight-forward programming techniques. Also, the resource usage is controlled [6]. So, connection pooling should be used to maximize the performance, while keeping the resource utilization below the point where the application will start to fail rather than just run slower [7].

Being normally used in web-based and enterprise applications, connection pooling is generally handled by an application server. Any dynamic web page can open and close it normally but in the background when a new connection is requested, one is received from the connection pool maintained by the application server. Similarly, when a connection is closed it is actually sent back to the connection pool [8]. Connection pooling also eliminates java database connectivity overhead. Further, object pooling also helps to reduce the garbage connection load. With database connection pooling, applications can be scaled in order to handle increased load and deliver high performance benefits. Using recycled database connection objects cuts the time taken to re-instantiate and load frequently used objects, reducing unnecessary overheads [9], [10].Connection pooling is not limited to using application servers. Usual applications that need frequent access to a database can benefit from connection pooling as well. This was traditionally handled by manually maintaining database connections, but as everybody expected that meant very good programming techniques as the framework for pooling is highly complex [11].

Various parameters can be set in order to make connection pooling work very good regarding the environment used for deployment, presented below:

- initialSize, the initial number of connections that are created when the pool is started,
- maxActive, the maximum number of active connections that can be allocated from the pool at the same time (negative for no limit),
- maxIdle, the maximum number of connections that remain idle in the pool, without extra ones being destroyed (negative for no limit),

- maxOpenPreparedStatements, the maximum number of open statements that can be allocated from the statement pool at the same time (negative for no limit),
- maxWait, the maximum time in milliseconds that the pool will wait when there are no available connections for a connection to be returned before throwing an exception (negative to wait indefinitely),
- minIdle, the minimum number of active connections that can remain idle in the pool, without extra ones being created, or 0 to create none,
- minpool, minimum number of connections that should be held in the pool,
- maxpool, maximum number of connections that may be held in the pool,
- maxsize, maximum number of connections that can be created for use,
- idleTimeout, the idle timeout for connections (seconds).

A single pool maintains multiple open connections, where each connection connects to the same database source using the same authentication. The pool also manages how those connections are handed out to be used, and what happens to them when they are closed. Both the size of the pool and the number of available connections are changed based on user-specified properties [12].

A pool therefore has two general types of behaviour: expiring, and non-expiring. An expiring pool is one for which any connection that is idle/unused for a specified time (idle timeout) is "expired" (removed) from the pool. In both situations the pool can hand out up to maxsize connections, and pool up to maxpool connections. The difference is that a non-expiring pool will not expire unused connections, so will generally retain a larger number of connections for reuse as they only get removed from the pool if they become invalid. The two pool types also differ in how they initially become populated. Immediately after creation both types start out with no connections in the pool.

Pooling of connections establishes automatically as items are checked in and out. Because of the additional checks that need to be done, an expiring pool can self-populate very quickly, but a non-expiring pool will populate gradually [13]. Picking appropriate values for the pooling properties is not always easy. Various factors may affect decisions, not least of which could be licence restrictions, system resources, etc. Many databases will close any unused connections once a certain time has elapsed (e.g. MySQL) [14].

2. Connection pooling method

When creating/using a connection pool instance it is assumed that access to the relevant java database connectivity drivers (JDBC) has already been established. To ensure this is done the JDBC driver(s) should be appropriately registered with *java.sql.DriverManager* before creation/use of any pools. Once the appropriate JDBC drivers have been registered, a connection pool may be

created. Once the pool is created it is ready to hand out connections. By default the pool doesn't open any connections until the first time one is requested, even if minpool > 0. If there is a need for populating the pool of connections at the initialization level, a call to the pool's init method should be done [15].

Once the application no longer requires the pool its resources should be released. Releasing the pool when finished is an important step and should not be omitted. Failure to release the pool can cause an application to hold on to resources, which often leads to unexpected results such as unexpected memory usage, failure of applications to terminate, etc. To help with this it's possible to automate the release with a shut-down hook, which releases the pool when the Java Virtual Machine exits [16].

In this article we are going to observe the impact of connection pooling on a web application which is deployed on Apache Tomcat Server [17]. We are going to use Web Application Performance Testing (WAPT) to realize two scenarios. WAPT is used as software for testing web application. Its main purpose is to show database access information in different scenarios that a regular user does in the application. In the first scenario, the connection to the database has no connection pooling, and in the second one we enable connection pooling. This is the chart returned by the WAPT client for the first case, no connection pooling (Fig. 1).

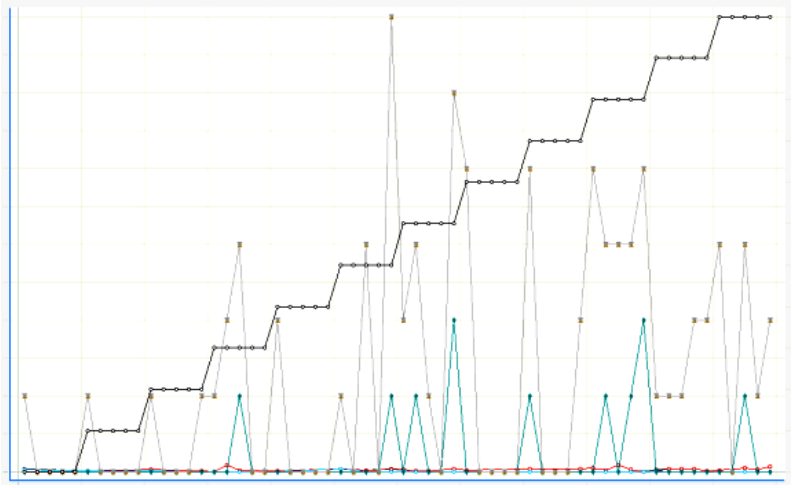


Fig.1. No connection pooling chart

Using the same web application, the same application server (Apache Tomcat) and the same scenario for WAPT, but enabling connection pooling with the following parameters [18]:

```
<beanid="mysqlDataSource"
class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
```

```
<property name="driverClassName" value="com.mysql.jdbc.Driver"/>
<property name="url" value="jdbc:mysql://localhost:3306/bwsn"/>
<property name="maxActive" value="20"></property>
<property name="maxIdle" value="20"></property>
<property name="username" value="root"/>
<property name="password" value="admin"/>
</bean>
```

the following chart is obtained (Fig.2).

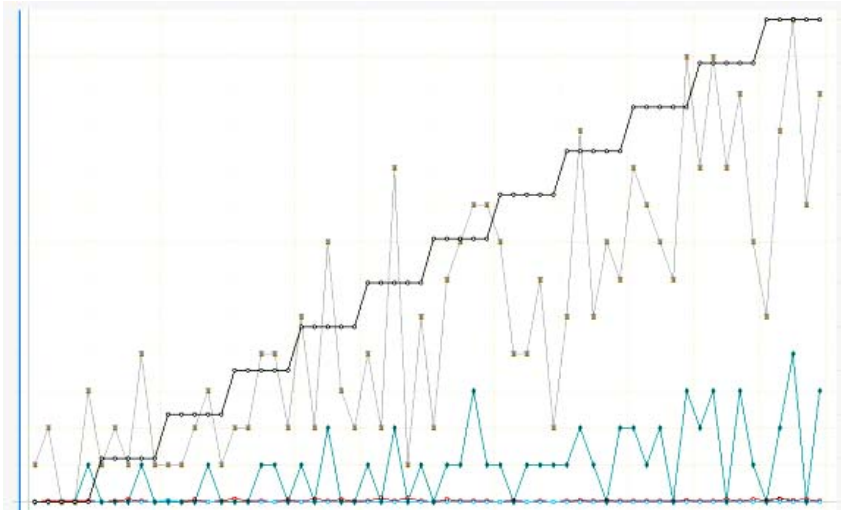
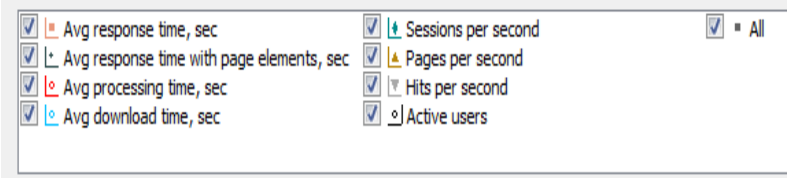


Fig.2. With connection pooling chart

The following legend is associated with figures 1 and 2. Therefore, we can see the average response time, the average processing time, average download time, hits per second and so on:



The grey line from the chart represents the number of hits. It can be noticed that the second scenario, with connection pooling, has a higher number of hits, therefore resulting higher performance. The red line shows the average processing time (in seconds). Again, in the case of using connection pooling, it is smaller than in the first scenario, so application efficiency is improved.

In order to justify the optimized performance, we are going to show the tables that contain values resulted from the two scenarios. These values are also generated by WAPT client, for the following scenario: five pages are accessed

from the application, the first one is the main menu, the second one presents a list of recordings (a request is done to the database for a select), the third one brings the user to an input form, where values are introduced and saved (so, another request is done to the database for an insert). After insert the forth page is showed with the new list of recordings (as the list has a new element, a request to the database must be done again for a select so the list will have all the actual elements), and the last page redirects to the main menu. There are twenty users logging in the same application, with a step of five seconds, and the length of the scenarios is sixty seconds.

3. Case study for a clinical medicine database

The purpose of this database is to obtain a better understanding of the diseases through its main components (mechanisms and elements) and to find as many relations between these components, which will lead to new investigation threads [19]. The access to this database is done using the connection pooling method.

In order to get this result, from the design point of view, the database has many tables and relations.

Therefore, the main results of the interrogations will be:

•the mechanisms and elements of each disease;

•the human body systems affected by a specific disease;

•the connections between a disease and it's mechanisms, elements, systems affected and the specific organs [20].

The structure of the database is represented in Fig. 3. Further explanations about the tables, fields and the relations between the tables will be given in the following pages.

3.1 Conventions

The fields marked with "*" are primary keys. Each table has a field named *Id* (of type *Integer*) which is primary key.The name of the *Id* field will have the following structure *IdTableName.* The foreign keys are marked using the *Id* of the referenced table (the foreign one) in the referencing table (the current one) with the character "_" ahead the *Id* name. Example: in the *MechanismElements* table, the *_IdMechanism* field makes the connection with the *Mechanism* table through a foreign key (*_IdMechanism* is the foreign key).

In the designing of this database there have been used the following types of relations:1:N, N:N and auto-reference relations [21]. Further on, it will be described how the last two relations were implemented.

3.2 Tables' Classification

The tables are divided into the following categories, depending of the information stored:

• **Main tables**: *Disease, DiseaseCategory, Mechanism, MechanismType, Elements, ElementsType, System, Organ.*

These tables are the one which store the most important information of the database.

• **Main connection tables**: *DiseaseElements, DiseaseMechanism, MechanismElements, SystemDisease, DiseaseCategoryDisease, MappingTable.*

The purpose of these tables is to implement the N:N relationships between two main tables. The name of a table follows the format Name_of_the_Table1Name_of_the_Table2, meaning a N:N relation between Table 1 and Table 2. These tables have only 3 fields: Name, _IdTable1, _IdTable2. The field Name is the primary key and this is why it must store an unique value. Because of this rule, the name will follow the format: _IdTable1_IdTable2.

• **The tables for the lists**: *ElementsLink, MechanismLink, SystemLink, GeneratedDisease.*

Each element/mechanism/system/disease will have a corresponding list of elements/mechanisms/systems/diseases with which it may have a relation.

From the point of view of the implementation of the database, this means an auto-reference relationship, while from the point of view of the programming code this means it will be implemented using the list type [22]. These tables are similar (from the point of view of the structure of the database) with the previous ones, except the fact that they have two fields which points out to the same table and will be implemented in a different way in the application.

• **Secondary tables**: *Publication, InfluenceFactors*.

4. Experimental results

Table 1 presents a summary of the experiment. It can be easily observed the difference between the successful hits and the slight improvement in average response time, therefore a reason for saying that optimization of database access took place. The summary also presents the number of successful and failed sessions, successful and failed pages, successful and failed hits and the total kilobytes sent and received.

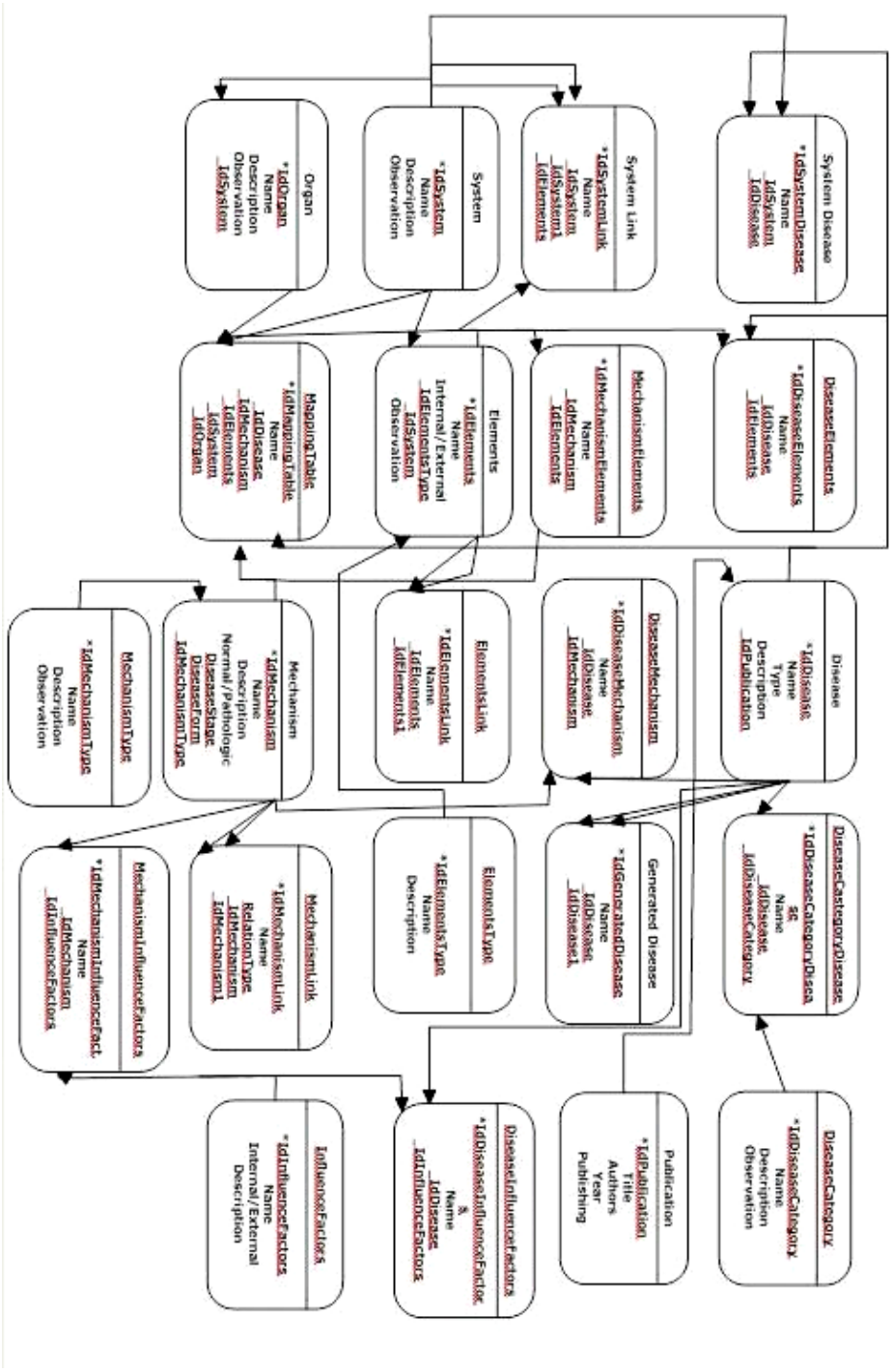


Fig 3. The Database Structure

Again, the profile with connection pooling provides great improvements in comparison with the scenario where connection pooling was not used.

Table 1

Summary

Profile	NoConnPooling	WithConnPooling
Successful sessions	11	54
Failed sessions	0	0
Successful pages	73	301
Failed pages	0	0
Successful hits	73	301
Failed hits	0	0
Total Kbytes sent	33	13.9
Total Kbytes received	856	227
Avg Response time, sec (with page elements)	0.05(0.05)	0.04(0.04)

Table 2

Successful hits (Failed hits)

Profile	NoConnPooling	WithConnPooling
0-6s	2(0)	7(0)
6-12s	1(0)	10(0)
12-18s	1(0)	14(0)
18-24s	3(0)	23(0)
24-30s	8(0)	23(0)
30-36s	7(0)	38(0)
36-42s	10(0)	31(0)
42-48s	9(0)	42(0)
48-54s	15(0)	59(0)
54-60s	17(0)	54(0)
Total	73(0)	301(0)

Table 3

Successful hits per second/ Network errors/Timeouts

Profile	**Successful hits per second** NoConnPooling	**Successful hits per second** WithConnPooling	**Network errors/ Timeouts** NoConnPooling	**Network errors/ Timeouts** WithConnPooling
0-6s	0.33	1.17	0/0	0/0
6-12s	0.17	1.67	0/0	0/0
12-18s	0.17	2.33	0/0	0/0
18-24s	0.5	3.83	0/0	0/0
24-30s	1.33	3.83	0/0	0/0
30-36s	1.17	6.33	0/0	0/0

36-42s	1.67	5.17	0/0	0/0
42-48s	1.5	7	0/0	0/0
48-54s	2.5	9.83	0/0	0/0
54-60s	2.83	9	0/0	0/0
Total	1.22	5.02	0/0	0/0

In table 3 we present the successful hits per second. As it can be seen, there is a significant difference of 3.8 seconds. Also in the same table we proved that there were no network errors, so the scenario is not influenced by other external parameters. Also, for the same reason, the timeout column shows that there are no timeouts in both scenarios, with and without connection pooling. In conclusion, there were no external factors that could influence the experiment.

In table 4 we present the actions related with every page from the scenario, and their relation with the users think time. The application server is installed on the locale machine and it uses the port 8080, the default port for Apache Tomcat Server. Connection to the internet will not be needed in order to access the application and introduce all the values received from the nodes (called users in the experiment) if the nodes and the application server are on the same network. If this demand is not met, the internet access is imperative in order to receive data. On the other hand, the machine where the application server is installed needs internet access to send various notifications to hospitals, firemen or police stations. These notifications are the main goal of the whole application, offering help to the elderly and all people who need medical homecare.

Table 5 strengthens the fact that it has been used the same scenario, the same pages and the same server and port for the experiment.

Table 4

Profile "NoConnPooling"

Name	Page
NoConnPooling.page_1: http://localhost:8080/disease	/disease
NoConnPooling.page_2: http://localhost:8080/disease/DiseasesList.action	/disease/DiseasesList.action
NoConnPooling.page_3: http://localhost:8080/disease/DiseaseAdd.action	/disease/DiseaseAdd.action
NoConnPooling.page_4: http://localhost:8080/disease/DiseaseInsert.action	/disease/DiseaseInsert.action
NoConnPooling.page_5: http://localhost:8080/disease/admin.action	/disease/admin.action

Table 5

Profile "WithConnPooling"

Name	Page
WithConnPooling.page_1: http://localhost:8080/disease	/disease
WithConnPooling.page_2: http://localhost:8080/disease/DiseasesList.action	/disease/DiseasesList.action
WithConnPooling.page_3: http://localhost:8080/disease/DiseaseAdd.action	/disease/DiseaseAdd.action
WithConnPooling.page_4: http://localhost:8080/disease/DiseaseInsert.action	/disease/DiseaseInsert.action
WithConnPooling.page_5: http://localhost:8080/disease/admin.action	/disease/admin.action

5. Conclusions and future development

Connection pooling is a technique used for sharing a cached set of open database connections among several requesting clients. It doesn't require modifying the code significantly and it provides enhanced performance benefits. Object pooling should be used with care. It does require additional overhead for such tasks as managing the state of the object pool, issuing objects to the application, and recycling used objects. Pooling is best suited for objects that have a short lifetime. When working in a rich Java EE environment it is most likely to use an out-of-box connection pooling facility provided by the application server, therefore making the applications' use of connection pooling transparent.

As a conclusion, the values presented above demonstrate what has been said in the first part of the article. The optimized database access has been proved with a higher number of hits per second, by a lower average response time and user think time. The values obtained are correct and consistent, as there are no external parameters that could influence the experiment, demonstrated by no network errors and no timeouts. In the same time, this experiment is somehow similar with optimizing database access by using the "join" clause [23].

As a future development, an experiment in which various connection pooling parameters are attributed different values can be done. For example, initialSize and maxActive parameters can receive different values in web applications with many or few requests, in order to decide what values are suitable for each case.

Regarding the medical database case study, the disease system is complex, intricate and interesting. It has a large number of variables. This database model has just a few numbers of elements. It can be expended, by adding new tables with new elements. The model will become more complex and will offer the possibility to store a lot of data.

Acknowledgement
This work was supported by doctoral program *POSDRU/107/1.5/S/76813.*

REFERENCES

[1] *Julia Lerman,* Programming Entity framework, February 2009, O'Reilly Publishing
[2] *Todd Cock,* Mastering JSP, Sybex Publishing.
[3] *Rahul Sharma, Beth Stearns, Tony Ng,* J2EE Connector Architecture and Enterprise Application Integrity, January 2003.
[4] *George Reese,* Database Programming with JDBC and Java, 2nd Edition, August 2000, O'Reilly Publishing.
[5] *George Reese,* Java Database Best Practices, May 2003, O'Reilly Publishing.
[6] *Jason Hunter, William Crawford,* Java Servlet Programming, Second Edition, April 2001, O'Reilly publishing.
[7] *SrinivasGuruzu, Gary Mak,* Hibernate Recipes: A Problem-Solution Approach, Apress publishing.
[8] *Nick Kew, Rich Bowen,* The Apache Module Book: Application Development with Apache, January 2007, Prentice hall Publishing.
[9] *Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy D. Zawodny, Arjen Lentz, Derek J.* Balling: High Performance MySQL, O'Reilly Publishing, June 2008.
[10] *Matthew Moodie, Kunal Mittal:* Pro Apache Tomcat 6, Apres publishing, 2007.
[11] *Maydene Fisher, Jon Ellis, Jonathan Bruce,* JDBC API tutorial and reference, June 2003, Pearson Education.
[12] *Ying Bai:* Practical Database Programming With Java, Wiley publishing, 2011.
[13] *Yuli Vasiliev:* Beginning Database-Driven Application Development in Java EE using GlassFish, Apress publishing, 2008.
[14] *David Parsons:* Dynamic Web Application Development using XML and Java, Cengage Learning publishing, 2008.
[15] *Art Taylor:* J2EE And Beyond: Design, Develop and Deploy World-Class Java Software, Pearson Education publishing, 2003.
[16] *Mahmoud Parsian:* JDBC Recipes: A Problem-Solution Approach, Apress publishing, 2005.
[17] *Fl. Manea, C. Cepisca,* An original way for having remote control over any type of automation, U.P.B. Sci. Bull., Series C, **Vol. 69**, No. 2, 2007, ISSN 1454-234x.
[18] Chad *Michael Davis, Scott Stanlick,* Manning Struts 2 in Action, May 2008, Donald Brown, Manning publishing.
[19] *Denis Cosmatos, Wyeth; Shein-Chung Chow, Duke University School of medicine, Durham NC,* Translational Medicine – Strategies and Statistical Methods*m* 2008.
[20] *Murray Longmore, Ian B.Wilkinson, Supraj Rajagopalan,* Mini Oxford Handbook of Clinical Medicine
[21] *D.Carstoiu*, Baze de date relationale, Printech Publishing, Bucharest, 2000.
[22] Design Patterns: Elements of Reusable Object-Oriented Software - Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides.
[23] *I. Rusu, C. Ariton, Mihaela Căpătoiu, Vlad Grosu,* The optimization of data access using "join" clause, U.P.B. Sci. Bull., Series C, **Vol. 68**, No. 4, 2006.