CONSISTENCY OF DATA BASES AND SECURITY OF TRANSACTIONS IN E-COMMERCE APPLICATIONS

Costin Radu BOLDEA
PhD, Associate Professor, University of Craiova, ROMANIA
e-mail: cboldea@inf.ucv.ro

Bogdan Ion BOLDEA
PhD, Associate Professor, West University of Timisoara, ROMANIA
e-mail: bogdan.boldea@e-uvt.ro

Abstract
This paper compares various anomalies of updating transaction in distributed databases for online business applications, and examines the access constraints needed to maintain consistent data in terms of concurrent access, as well as some methods to ensure data integrity and security, especially the appearance of Phantom or Dead-lock phenomena. The paper also presents the result of an experiment to identify the occurrence of these undesirable phenomena.

Keywords: *Concurrent transactions, Database management systems, Phantoms, Locking, Data Consistency.*

JEL Classification: *C80*

INTRODUCTION

Data is a company's most valuable asset today, not its offices or production facilities. Regardless of the nature of the company - be it a private enterprise, an educational institution or a state-owned company - its history, current status of assets and liabilities, as well as all information related to the production or distribution process are recorded in internal databases or public. If this data is out of date with the status of the company or if the data is inconsistent with themselves, their usefulness is questionable and there are likely to be problems.

Application programs update data according to changes in enterprise status. Application system components share the same dataset and usually, but not necessarily, share the same computing environment. Today, application systems use many platforms: desktop microcomputers, file servers, and mainframes. With the advent of web-based cooperative computing, sometimes we don't know or don't even care where the app is running. Although each platform brings its unique benefits and challenges to data management science, the fundamental concepts of database management software have remained unchanged for more than three decades [3], [8].

Database Management Systems (DBMS) were created to provide consistency and consistency to application systems that process large data files. The organization and management of data in a database must meet five essential conditions [4], [6]:

- *A good representation of the surrounding reality*, ie the database must always provide a true picture of reality through reliable and up-to-date information;
- A *non-redundancy of information*, the information contained in the database must be unique from a semantic and physical point of view;
- An *independence* of the data from the processing; the data is a true picture of the real world, the application programs must be designed in relation to this data structure;
- *Data security* and confidentiality; data security must be ensured through physical procedures, and confidentiality through procedures that prevent access by unauthorized users;
- *Performance in operation*, any processing request must be met in a time convenient to the user, which involves the use of optimization techniques to reduce processing time.

However, the widespread use of the Internet as a means of communication and access to a

company's data has led to various security or consistency issues due to the massive use of competition, representing the simultaneous access of several users or programmers to a database, and of the distribution of data over several storage areas, possibly at great distances from each other. Ensuring the integrity of the database, its consistency and the security of access to it thus becomes a priority for any company that operates massive data and information, whether they are commercial or personal.

This paper presents various deficient aspects of competitive access to distributed databases, focusing in the end on combating phenomena such as Phantom (false data) or Dead Lock (inter-blockages) that may occur during the use of large databases. The organization of the article is as follows: Section 2 describes the various transaction control anomalies in distributed database databases for business applications; Section 3 presents concrete examples of the occurrence of these anomalies, followed by an Experimental Section, analyzing cases where ghosts or blockages occur during the execution of competing transactions. Last Section concludes the paper.

1. DISTRIBUTED DATABASE TRANSACTIONS AND CONTROL ANOMALIES

Commercial databases were originally created as a form of storage of information related to marketed products and potential customers or suppliers of the company. However, the explosive development of e-commerce has generated the need to store a very large amount of information, possibly shared and updated from separate locations, which has led to the emergence of Distributed Databases (DDB). The primary rationale for a distributed database is as a decentralized enterprise that has to share information between different distributed parties. Distributed operations can lead to the following situations:

• Data can be generated in several parts, requiring fast local access and extracurricular data summaries.

• Data can be generated centrally, requiring fast remote access. Both the exchange and remote users must update the data.

• Data can be generated in many places, requiring quick access to both local and remotely stored data.

In all these situations, Distributed Database can lead to better performance than centralizing it in one location. This is because distribution reduces the volume of data communications, with most accesses being local, and reduces the amount of data stored on any given machine, reducing capacity requirements and improving local responsibilities.

There is only one definition of data elements in a given database management system (DBMS). This definition is given by the database schema of a DBMS. In general, a distinction is made between the programmer's view of the database, which defines its logical schema, and the computer's "vision" of the database, called the physical schema. The database management system connects the two schemes.

The purpose of database management systems is to provide quick and easy access to large amounts of data, but to do so in ways that ensure that the integrity of the database is always maintained ([3], [7]). This means that a database management system must allow users to define and manage rules or constraints applied to certain critical data elements. Defining and enforcing data security and integrity constraints is essential to the usefulness of any database management system.

An important component of a database management system is its transaction manager. A *transaction manager* monitors data object updates to ensure that the database is always in a consistent state.

Transactional mechanisms are not new in the world of databases. In the 1970s, just a few hundred miles away, IBM Laboratories in San Jose and the University of Berkeley developed the first

relational systems, System R and Ingres, in parallel, both of which included some form of transactional processing. Since then, all major DBMSs have incorporated trading mechanisms.

The purpose of transaction processing is to maintain the integrity of the database. The integrity (or *consistency*) of the database is a notion that is quite difficult to define rigorously. For the purposes of this presentation, it is sufficient to admit that the database is consistent if it complies with all formal *rules of integrity* that have been imposed on it and, in addition, the information it contains is assumed to be correct.

However, it should be noted that transactional mechanisms are not the only ones that deal with maintaining integrity. Specifically, transaction processing relates to only two aspects:

- *Database recovery* after an incident - is based on the inclusion of a certain level of redundancy by storing the history of transactions in a so-called "log". Although this is not the subject of this article, some technical elements on journaling will continue to be used.
- *Concurrency control* - This is a critical issue in OLTP (On-Line Transaction Processing) application systems. This is about "controlling" (and not necessarily "avoiding") interference because, although always undesirable, such interference may be allowed - in some well-defined forms - to increase system performance. We will see how this is done later.

All issues discussed below relate to large, multi-user databases. We will consider especially the case of relational systems (the vast majority of DBMSs are of this kind) although - except for the aspects related to SQL - all the notions presented are applicable to any type of DBMS. Moreover, most notions generally go beyond the scope of databases, being applicable to competing systems in general.

Transactions generally consist of sequences of read and write operations to the database, along with the necessary computational steps. However, the transaction reflects an *atomic*, indivisible operation, which means that the sequence of operations must be executed in full. A component of the DBMS called the transaction manager ensures compliance with this principle based on the two fundamental operations: COMMIT - signals the successful conclusion of the transaction and causes its consequences to continue; ROLLBACK - signals the failure of the transaction and causes the cancellation of its consequences by restoring the status of the database before the start of the transaction.

A transaction transforms the database from a *consistent* state to a consistent state. Because the operations that make it up are performed sequentially, the database may not be in a consistent state between the start of the transaction and the end of the transaction. For example, a transaction that transfers the amount of 1000 lei from account A to account B, can be executed in two stages: first the amount from account A is subtracted and then the amount is added to account B. between the two stages the database does not reflects the reality, because the amount is neither in account A nor in account B. As such, the access of another application to the two accounts between the two moments of the transfer time must be restricted. As the database may be inconsistent during the execution of a transaction, the result is that a transaction is not allowed to act on the intermediate results of another transaction or, in other words, the transactions must be executed as if they were *isolated*.

Once a transaction has been made, the system assures the user that the updates that the transaction has produced become persistent. This means that they will "survive" even if an accident occurs immediately after committing, before the new values are physically written to disk. In practice, this mechanism is implemented by logging data operations as well as transactions.

In the absence of competition control mechanisms, undesirable phenomena can occur, which can lead to erroneous results or corruption of the consistency of the database. The literature ([1], [5], [8]) generally mentions three typical anomalies that may occur: lost updating, improper reading, and inconsistent analysis, all three of which result in the appearance of the **Phantom** data phenomenon-

erroneous data delivered to the end user, or inter-access lock (**Dead Lock**), in which case users lose access to data.

2. PHANTOMS AND DEADLOCKS

To illustrate the appearance of "Phantoms" we will consider a seat reservation application for an airline.

Consider two “Reservation” transactions, *R1* and *R2*, which are performed concurrently and which happen to refer to the same *Z* flight. We will disregard the insertion operation (which is not important in this context) and use an alternative simplified notation as *Write*; because the selection is - essentially - a reading, we will mark it with *Read*. The updeting, being a writing, we will also note it with *Write*. A possible temporary sequence of events could be the following:

t1 – R1: Read Z
t2 – R2: Read Z
t3 – R1: Write z
t4 – R2: Write Z
...

The readings made by the two transactions will find the line in the same state. Assume that the value of the " Seats_sold " field is 25. Both transactions will set their local variables Seats_sold to 25. The update performed by transaction R1 at time t3 will change the value of the field to 26. At time t4, transaction R2 also updates its own computer line Z, but placing the same value on the Seats_sold field. Two reservations are made, but the number of seats sold increases by one unit. If the number of requests is high, it is very likely that extra seats will be sold for that flight when the central database is updated (COMMIT function).

The anomaly called "*lost update*" occurred due to the fact that transaction R2 based its update from time t4 on a value read at time t2, which was meanwhile (at time t3) modified by transaction R1 . Basically, the effect of the update made by transaction R1 at time t3 was lost. If transaction R2 read line Z after it was changed by R1, then R2 would have placed 27 in the Seats_sold field, which would have been correct.

Another possible situation is when one of the transactions is canceled. Consider the following scenario:

t1 – R2: Read Z
t2 – R2: Write Z
t3 – R1: Read Z
t4 – R2: ROLLBACK
t5 – R1: Write Z
...

In this situation, transaction R1 will have, in the local memory buffer, the result written by transaction R2 at time t2. The cancellation of the transaction can occur for various reasons (assuming that the addition of the line corresponding to the reservation in the RES table fails due to lack of disk space) and will cause the Z line to return to the values before time t2. In this case, the R1 transaction will base its future actions on values that practically never existed in the database! Suppose that at time t1 the Seats_sold field of line Z contained the value 25. At time t2 this value is incremented so that transaction R1 will read at time t3 the value 26. At time t4, transaction R2 is canceled and consequently the value of field SOLD of line Z will be brought to the value it had before the start of transaction R2, so 25. At time t5, the transaction will write in line Z the value 27. Which is obviously wrong. The ”*improper reading anomaly”* (also called *uncomitted dependency*) stems from the fact that transaction R1 read intermediate results of transaction R2.

To exemplify a third anomaly, called "*inconsistent analysis*" we will consider the CLIENT table, on which two transactions are performed concurrently. Transaction T1 calculates the total amount received from Z-passengers, while transaction T2 transfers a certain amount x from a C3 customer's account to a C1 customer's account (which happened to fly together). Normally the total amount should be the same no matter which account x is in. Suppose that the balances of customers C1, C2, and C3 are 200, 100, and 300, respectively, and x is 20.

But here's what can happen:

t1-T1: Read C1 (sold=200, total=200)
t2-T1: Read C2 (sold=100,total=300)
t3-T2: Read C3 (sold=300)
t4-T2: Write C3 (sold=280)
t5-T2: Read C1 (sold=200)
t6-T2: Write C1 (sold=220)
t7-T2: Commit
t8-T1: Read C3 (sold=280, total=580)
...

For customers C1, C2 and C3 the total should, of course, have been 600. It is obvious that the total amount will be wrong. Unlike the other two cases presented, it is no longer a question of reading updated values of a transaction before committing it. T1 does not read anything from the time the T2 transaction is started until it is completed. The problem is that T1 read one of the accounts before the transfer and the other after the transfer.

In all three cases, the anomalies are caused by competitively executed interfaces between transactions. A simple way to avoid these interfaces would be to fully execute the transactions in the order they started. In this case, the performance of the data server would be unacceptably affected. It's as if in a large self-service store, customers are left inside one by one, each after the previous customer has finished all their shopping.

In order to best balance performance and security, database management system builders have developed several ways in which interfaces between competing transactions can be controlled. The most common is the mechanism based on ***locking*** portions of the database to prevent other transactions from accessing that data during critical operations performed by a transaction. Another method is based on applying "*timestamps*" to transactions and objects involved in transactions.

The idea behind the blocking technique is very simple: a transaction that has started operating on some data must prohibit other transactions from accessing that data until the transaction is completed. During this time, the data is "kept under lock and key". Data lock control is provided by a component of the DBMS called the Lock Manager (LM). When a transaction T wants to access a specific object in the database (reservation table for example), it will ask the LM component to block the object. If the object is blocked by another transaction, transaction T will be put on hold until the object is released.

Let's reanalyze the second anomaly presented above, inconsistent reading. At time t1 transaction R2 blocks line Z. At time t3, transaction t3, transaction R1 in turn requests the blocking of line Z but, since the line is already blocked by R2, it is put on hold. At time t4, transaction R2 ends (via ROLLBACK) and releases the Z line. Only now does R1 get the Z line lock and can access it. The data on which R1 is now acting is "clean" (the effects of transaction R2 have been canceled), so the anomaly has been avoided.

It is easy to see that this way of locking is too restrictive. The anomalies appear only in the case of data update, which suggests that refining the technique involves the use of two types of locks:

Share lock (Slock) - Allows you to read the object but forbids changing it, which is why it is also called "read lock". Multiple transactions can block a particular object from being read at the same time.

Exclusive lock or Xlock - Prevents other transactions from accessing the locked object, either for reading or editing. Exclusive locking is more "strong" than shared locking, and is only used for updates, which is why it is also called "write lock".

Let's review the "lost update" issue now, using shared and exclusive locks. At time t1 transaction R1 requests a shared block on the Z line and (assuming the line was not blocked for writing by another transaction) obtains it. At time t2, transaction R2 also requires a shared Z-line lock to read and update. It does not get blocked because the line is blocked for reading by transaction R2, so it is put on hold. The R2 transaction also requires exclusive blocking and obviously does not obtain it for similar reasons. None of the transactions can continue, as each is waiting for the other to release the Z line.

This situation is called **DeadLock**. It is easy to verify that even in the situation of "inconsistent analysis" an interlock will be reached. The result is that we solved the problem of anomalies but we got another problem, that of interlocks. Solving the new problem involves two aspects: preventing interlocks and detecting them.

The simplest method of detecting a deadlock situation is based on a time-out mechanism: if the duration of the execution of a transaction exceeds a predetermined value, the system deduces that an interlock has occurred. In practice, a mixture of techniques is most often used: an access protocol is required to reduce the possibility of interlocking (without completely preventing it, but also without significantly inhibiting competition), to implement a mechanism to detect the most interlocking usual, leaving the others to a time-out mechanism. The actual settlement of an interlock is the determination of a "victim" between the deadlock transaction and its cancellation (ROLLBACK). Once the interlock has been removed, the transaction can be restarted.

The question is: how do we know if the concurrent execution of a group of transactions is correct or not?

We call the planning (schedule) of a set of transactions an order of execution of the elementary steps (LOCK, READ, WRITE, etc.) of all transactions. Ordering refers to steps in different transactions, the relative order of the steps of the same transaction not being affected. A schedule is called a serial if all the steps of any transaction appear in consecutive planning positions. Such planning results in a serial execution, without interference of transactions. A schedule is called **serializable** if and only if its effect is equivalent to that of a serial schedule. The implementation of transactions through serializable planning prevents the occurrence of interlocking phenomena and phantom data, ensuring the consistency and integrity of data.

A step-by-step schedule of many transactions containing SLOCK (read lock) and XLOCK (read-only lock) primitives can be serialized if the following rules are followed:

1. Any access unit blocked by a transaction must be unlocked by the same transaction before it is completed.

2. No transaction attempts to unlock an access unit that it does not block in read or write-write.

3. No transaction attempts to block an access unit that it blocks from reading (either read or read-write).

4. No transaction attempts to block a read-write access unit that it blocks from read-write.

5. A transaction may request and obtain a read-write lock on an access unit that it is already blocking from reading. This is possible because the read-write lock is more restrictive than the read-only lock.

Example:

Let two transactions T1 and T2 be defined by the sequences

T1: READ A; A=A-10; WRITE A; READ B; B=B+10; WRITE B;

T2: READ B; B=B-20; WRITE B; READ C; C=C+20; WRITE C;

Any serial planning of T1 and T2 transactions has the property that the amount A + B + C remains unchanged. Figures 1 and 2 show two different T1 and T2 transaction schedules: one serializable and one non-serializable.

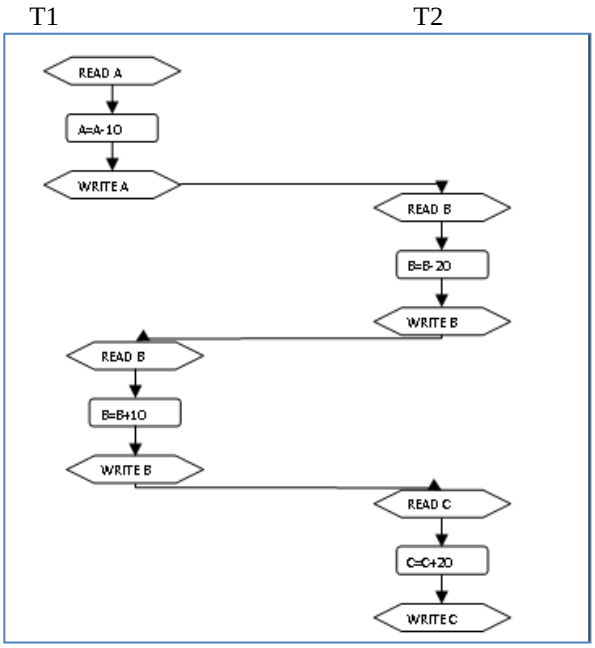

Figure 1. Serializable Transactions

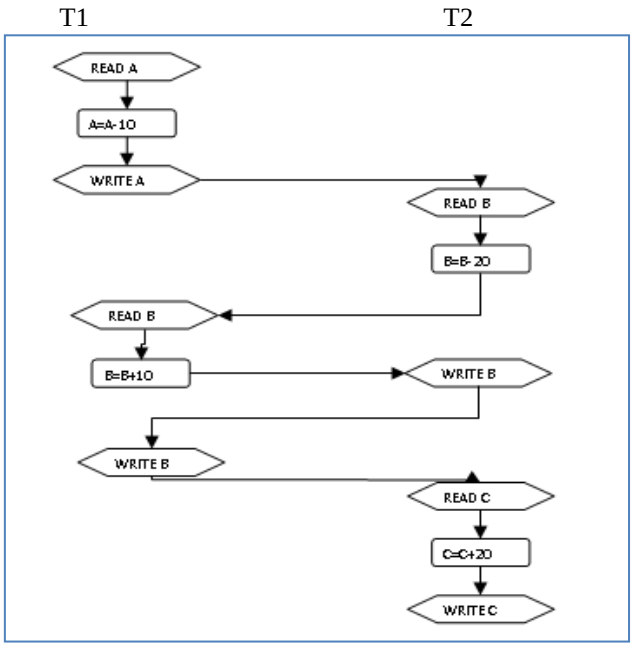

Figure 2. NON-Serializable Transactions

After the execution of the non-serial planning, the amount A + B + C is increased by 20 due to the loss of the update of B (WRITE B) from the transaction T2.

3. EXPERIMENT TO TEST THE APPEARANCE OF DEADLOCK AND DATA-PHANTOM

In order to test the appearance of Phantom or DeadLock phenomena in ORACLE databases, we performed several UPGRADE experiments at the READ COMMITTED and SERIALIZABLE isolation level of PL // SQL [1], [2]. At the first isolation level, the transaction acquires read locks for all retrieved data, but does not acquire interval locks, at the second the ORACLE server forces serialization.

In the experiments we used an Employees table and two PL / SQL scripts that use cursors to change employees' salaries by plus 10% and minus 10%, respectively (Figure 3), operated by two different users.

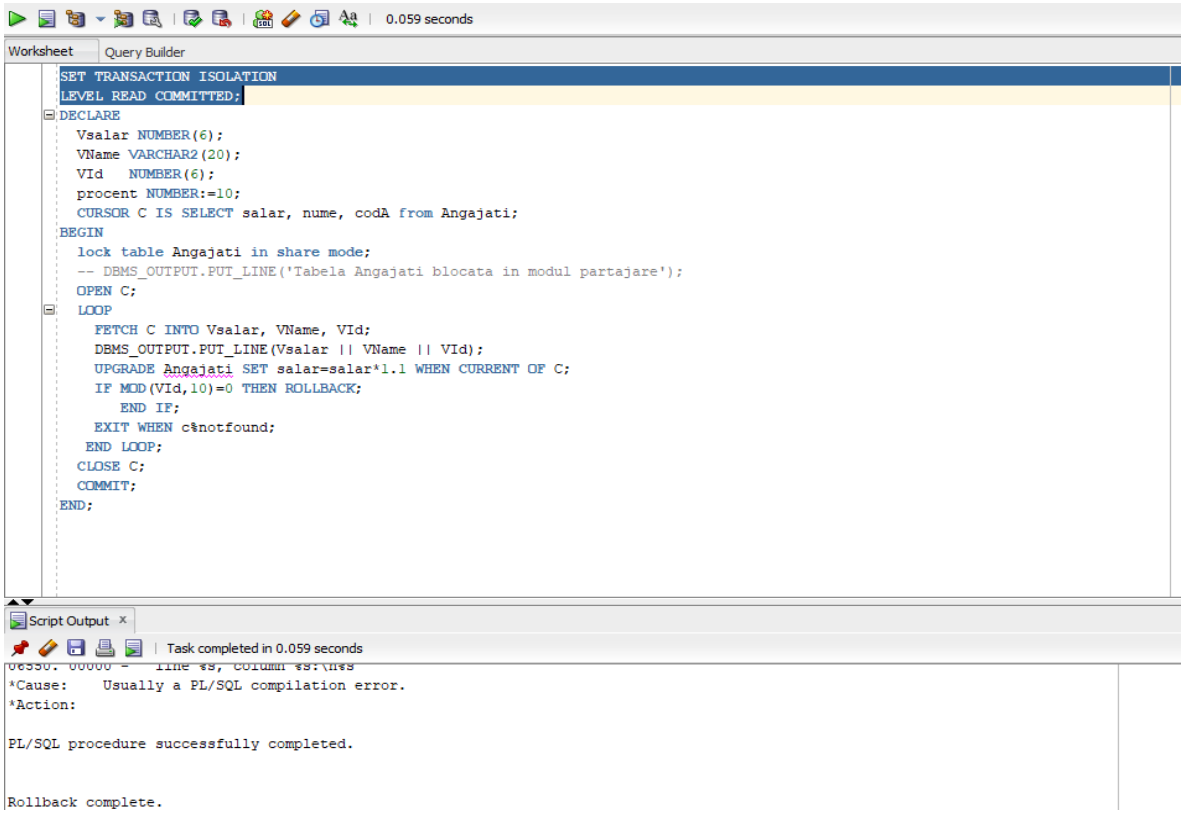

Figure 3. Sample script for T1 first user transactions. The second user has an equivalent block that reduces salaries by 10%

For both users, we locked the Employees table in sharing mode, using 1000 records per table. The completely independent execution of the two scripts should have no effect on the data in the table, only that we executed them in parallel. Since the data is read at the beginning of the transaction by transferring it to the local memory buffer of the cursor, UPGRADE attempts result in phantom effects in the first case at the "READ COMMITED" isolation level, and with execution error (at "SERIALIZABLE" level), by the appearance of an interlock generated by forcing the table to lock in shared mode.

Thus, for the first case of isolation level, the ROLLBACK operation, sporadic, resets the current transaction for each of the two users, but the required update operations produce different results in case of reading the data during the interspersed execution of the two blocks; on average, between 600 and 700 Phantom results are obtained when a sequence of successive data displays is introduced inside the LOOP loop.

At the "SERIALIZABLE" isolation level, the appearance of DeadLock requires the modification of the scripts by introducing an exception, otherwise the execution of the two will be permanently blocked.

CONCLUSIONS

In large databases, the introduction of temporary access blocking clauses to a database component and changing the settings of concurrent update transactions can reduce the occurrence of phantom data phenomena, but induce the occurrence of update interlocks, which require interlocking. important changes to auto-update scripts.

Given that these types of large databases are often used by commercial applications, and not only, maintaining the consistency and integrity of data remains a necessary goal, without hindering access to them, or blocking update applications.

BIBLIOGRAPHY

1. Boldea, C. R. (2020). *Gestiunea Bazelor de date distribuite în ORACLE folosind PL/SQL*- Joshua Print, Vrsac.
2. Boboila, C. ; Boboila, S. & Lupsoiu, C. (2009*). DELP System: Tracking Deadlocks and Phantoms in Databases*, Proceedings of the European Computing Conference, Vol. 2, p.319, Springer-Verlag.
3. Lupsoiu,C. & Boldea, C.R. (2008). *Modelarea şi proiectarea bazelor de date*, Sitech, Craiova
4. Loshin, P. & Vacca, J. R. (2004). *Electronic Commerce, 4th Edition*, Charles River Media, Inc., 10 Downer Avenue, Hingham, Massachusetts 02043.
5. Null, L. & Lobur, J. (2003). *The Essentials of Computer Organization and Architecture*, Jones and Bartlett Publishers.
6. Moréjon, J. (1992). *Principes et conception d'une base de données relationnelle,* Les Editions d'organisation, Paris, p. 20
7. Reynolds,J. (2004). *The Complete E-Commerce Book-Design, Build & Maintain a Successful Web-Based Business*, Second Edition. CMP Books.
8. Silberschatz, A., Korth H. F. & S. Sudarshan, S. (2005). *Database System Concepts, 5th Edition*, Boston, MA: McGraw-Hill.