# Bit-level Locking for Concurrency Control

Jad F. Abbass and Ramzi A. Haraty
Department of Computer Science and Mathematics
Lebanese American University
Beirut, Lebanon
Email: {rharaty, jad.abbass@lau.edu.lb}

*Abstract*--**Multitasking in both uniprocessor (multithreading) and multiprocessor (multiprocessing) systems have been attracted by many applications. Database systems are somewhat the most important in this regard, especially in centralized and humongous ones. Sometimes thousands, and maybe hundred of thousands of operations are sent to the transaction processing system per second. To handle this bottleneck some queries/updaters are executed concurrently. However, parallelism in such cases is extremely accurate based on the well-know restriction - locks. In this paper, we implement a lock approach based on a Boolean array (1D and 2D) and on the logical OR operation to specify which transactions can be executed in parallel.**

**Keywords**: Bit-level locking, concurrency control, and database management systems.

## I. INTRODUCTION

The concept of transaction provides a mechanism for describing logical units of database processing. Transaction processing systems are systems with large databases and hundreds of concurrent users that are executing database transactions. Examples of such systems include systems for reservations, banking, credit card processing, stock markets, supermarket checkout, and other similar systems. They require high availability and fast response time for hundreds of concurrent users.

Multiple users can access databases and use computer systems simultaneously because of the concept of multiprogramming, which allows the computer to execute multiple programs or processes at the same time. If only a single CPU exits, it can actually execute some commands for one process. However, multiprogramming operating systems execute some commands for one process, then suspend that process and execute some commands for the next process, and so on. Nevertheless, the process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually interleaved. Interleaving keeps the CPU busy when a process requires an input or output operation, such as reading a block from a disk. If the computer system has multiple hardware processors, parallel processing of multiple processes will become possible.

Controlling concurrency is a critical issue in database management systems - building the sets of transactions that can be executed in parallel is not an easy duty to the system. The system should ensure that there is no overlapping in the locks associated with every query/updater; otherwise, several problems can occur. For instance the two well-known problems: The Lost Update Problem and the Dirty Read Problem both produce eventually incorrect results and/or leave the system in an inconsistent status.

The oldest and maybe the most traditional way to control the access to an item is the mutual exclusion; that is, when a transaction wants to access an item, it would be granted a permission to use this item, and other operations should wait in a queue.

This paper is based on interleaved concurrency – using the multithreading approach. The main purpose of this paper is to implement the array-based locks to BUILD the sets of the operations that can be sent to the DBMS at the same time. In this work, first we let the transaction execute serially, then we build sets of transactions that can be executed in parallel, and execute these sets serially based on one dimensional array for both read and write operations, the last phase is to build again the sets just mentioned, and execute them serially but based on a two dimensional array, one for read and the other for write operations and taking account that read operations do not conflict, and at the end compare the response time, as well the number of sets. The remainder of this paper is organized as follows: section 2 provides background and motivation. Section 3 presents the proposed scheme. Section 4 discusses the experimental results and section 5 provides the conclusion.

## II. BACKGROUND AND MOTIVATION

Concurrency control is considered one of the oldest topics in the field of computer science. It has been in research for more than thirty years [3][4][5][6]. Many papers had proposed correct techniques and solutions for concurrency control in database systems, yet these solutions showed many drawbacks and disadvantages when there was an increase in the number of parallel operations [2][8][9][10][12][13].

In order to enhance performance in database environment, database applications must be allowed to interleave their execution. However, concurrent execution gone unsupervised can lead to erroneous results and inconsistent database state. A database management system prevents such inconsistencies by enforcing a concurrency control strategy, which enhances performance and maintains correctness. In other words, the concurrency control strategy will only produce serializable schedulers. Such schedulers will produce executions which are equivalent to some serial order. A number of methods such as time stamp ordering, two phase locking, serialization graph testing, tree locking, optimistic certifiers, and request ordered linked list (ROLL) [11] have been proposed. Many of the existing relational and object-oriented database systems use these methods with varied levels of performance. The focus of this paper is to implement a "binary lock" approach and develop a high performance concurrency control algorithm which is both, efficient and correct.

One of the main types of database systems is the transaction processing system. This latter has spread in the last few years due to the dramatic improvement in the speed of the computers, known as servers. Since the powerful computers and servers, like banks' servers are becoming fast and can serve multiple users at a time, much work and interest are being directed toward the development of transaction processing systems

The main characteristic of any transaction is abbreviated as ACID, which stands for Atomicity, Consistency, Isolation, and Durability. Atomicity means the whole transaction must be done or not; it is not allowed to execute a part of it and leave the remaining unexecuted. Consistency assures that when the transaction terminates, it should leave the systems in a consistent state. Not that during the execution, the system may pass through an inconsistent state; however, what does matter is the state where the transaction finishes executing. Isolation states that every transaction must behave as it is isolated from other transaction, although there may exist an interleaving among many transaction as we will see later, but the effects of the transaction must be done as it was executed in isolation. Durability states that when the transaction commits, all its effects are permanent and cannot be undone.

The basic four operations in a transaction processing system are read, write, commit and abort. Read(x) returns to the system the value of x, however, Write(y, 1) means write into y, the value 1. A transaction may include any number of read and write for different data items, but eventually either commit or abort must end the execution of the transaction. Commit states, implicitly, that the transaction is finished and all its effects have to take place and it returns an acknowledgment to the user that the transaction has been terminated successfully and all its effects have been done. The system uses abort whenever an error occurs and the transaction should terminate; however, all effects done until the moment the error occurred, should be undone, and of course acknowledge the user that the system is in a state as if the transactions had not started at all.

When we say two or more transactions interleave their execution, it means that their operations do not execute in a serial way, that is, the system may execute an operation from T1 and then an operation from T2 and so on. For this reason we do need concurrency control model to handle this problem, because interleaving execution can leave the database in an inconsistent state. However, not all operations conflict with each other for instance the most common known conflict is between read and write.

## III. THE PROPOSED TECHNIQUE

In general, concurrency control techniques can be classified into two categories: Locking and Timestamp. The former relies on locking a data item (it depends on the granularity of this data item, it can be a single attribute value, a record, a block of a disk) while it is in use by a transaction, when this transaction finishes using this data item, it releases the lock to be used by other transactions.

Actually, we can see the lock as a variable or a value associated with each data item. The status of this variable/value determines the status of the data item whether it is busy or not. The types of locks are:

- Binary Lock: this technique is quite simple and efficient, when the binary value is set to 0, it means that the associated data item is free to use, and vice versa (This paper's phase II is based on this technique).

- Shared/Exclusive (or Read/Write) Locks: this approach is quite similar to the previous, except that it relies on the fact that read and read operations do not conflict, so it allows two or more transactions to access the same data item for read purpose (This paper's phase III is based on this technique).

- Conversion of locks: it allows a transaction to "upgrade" its lock from read to write.

As mentioned before, too many approaches and techniques have been proposed to control concurrency problems in multiuser database systems environment. The oldest and simplest is based on Boolean variables that is usually corresponds to a data item for a special transaction/operation. If this bit is assigned to 0 it is free to be used by other transactions/ operations; otherwise no other tasks can grant the access to that data item.

The closet approach used to the one applied in this paper is ROLL (Request Ordered Linked List) [7]. However, the latter is based on Boolean variables embedded in a linked list. Our paper is based on the same techniques and the same operation (logical OR) to know whether the data item is busy or not, but on a different data structure; array of two dimensions first and then of three dimensions. This method is more efficient, since the static data structures are more efficient than dynamic in memory and CPU speed terms, and of terms of implementation.

Basically, our work is built on two parts, the first one uses a two dimensional array, and the second uses a three dimensional array. In both cases, the array is composed of Boolean arrays corresponding in essence to a special transaction for a special data item, the difference lies in merging read and write locks in one Boolean element in that array in the first part, and splitting this element into two elements: one for read locks, and the other for write locks in order to improve parallelism among transactions. In other words, to increase the number of transactions that can be executed in parallel or simply interleaved.

The granularity of the data item used in the project is the record. So, whenever a record is locked, all fields within this record are locked as well, even if some of them are in use. When a transaction wants to access at least one attribute in a record, the system sets the corresponding element in the array to 1 to prevent any other transaction to access this record. Also, recall the four properties of any transaction ACID, atomicity indeed here is valuable. For instance, if the transaction wants to use 248 records and one of them is locked the whole transaction waits until this one is free.

Every transaction can contain one or more operations. In order to generalize the experimental results, three versions were created based on the number of operations per transaction. The first creates randomly, from 1 to 5 operations per transaction. (Low) The second creates 6 to 10, (Medium) and the third creates 11 to 15 operations per transaction (High). In the three versions, we use twenty transactions.

The set of operations from which the system generates a random number of operations from a specified range, depending on the version, is composed of 200 queries and updaters - 20% of the operations are usually updaters and the remaining are queries, that is, write and read operations respectively. Moreover, the operations within this list are evenly distributed in terms of the number of records, they access; for instance, 10% of them access 1 record, 10% access 3 records, 10 % access 10 records, and so on.

In general the whole list accesses roughly 70 to 80% of the whole database, which constitutes 2180 records and 12 fields. The latter, in their turn, are distributed evenly among all data types (text, number, auto number, date, time, currency).

The paper is composed of three phases (or version). The first is serial, that is, the 20 transactions are executed serially, one after the other, without parallelism. The purpose of this execution is to record the running time, and compare it with the next two phases, to show that, even in a uniprocessor system, multithreading is more efficient. The next phase is the application mentioned before about using a two dimensional array. In other words, using one bit for both read and write locks; however, the third and last phase uses two bits, one for read and the other for write locks. For more information, interested readers are referred to [1]

## IV. EXPERIMENTAL RESULTS

This section shows 30 experiments or executions, distributed evenly among the three versions; 10 for the first, Low, 10 for the second, Medium, and 10 for High. All these experiments were conducted on Intel Mobile Centrino 1.78 GHz, with 512 MB of RAM.

### *A. Version 1*

In version one, the number of transactions does not change; however, the number of operations within each transaction changes. In this version, we randomly assign for each transaction form 1 to 5 operations. These operations could be queries or updaters, depending on the randomization action (see table 1).

### *B. Version 2*

In version two, the number of operations within each transaction varies from 6 to 10. It is clear that we still have a decrease in both -the number of sets and the running time. However, comparing with version 1, there is relatively a high difference in that decrease. Accordingly, we can say that, when the number of operations increases within the transaction, the efficiency of the adopted concurrency control model decreases (see table 3).

### *C. Version 3*

It is obvious now that the restriction that is implemented within this work, the atomicity of the transaction; one thread per transaction, has led to the "bad" results when a transaction contains between 11 and 15 operations, and if one record required per one operation conflicts with another operation from a different transaction, these two transactions cannot work in parallel. It is worth mentioning also, that without phase III, version 3 has almost no efficiency at all in terms of parallelism (see table 5).

### *D. Comparison of the Results*

In this section, we conduct a study that compares the results among the three versions especially in terms of number of sets that run sequentially, and try to see the de facto causes that have affected these results.

Between phase I and phase II, the efficiency is improved by introducing the ability to run more than one transaction at a time, and to see that instead of having 20 sets of transactions that run serially, this set will decrease, such that every set can contain one or more transactions (see tables 2 and 4).

Between phase II and III, the running time is not a critical issue like the previous one; however, the number of sets of transactions that run serially is the moral of the story here. So, improving the parallelism based on the fact that read and read operations do not conflict, to see by how many sets the third phase can decrease (see tables 6 and 7).

## V. CONCLUSION AND FURTHER WORK

In this paper, we have attempted to present a relatively new technique for concurrency control model for the real-time database systems known as, transaction processing systems, in order to solve the locking mechanism by using an array (2D and 3D) of Boolean elements.

To improve efficiency of transaction processing systems, we have tried to allow more than one transaction to run concurrently or simply, interleaved with others, as long as, no common data items - records in this work -are in use mutually.

Experimental results conducted on the three versions, depending on the number of operations per transaction, has proved, that despite the additional computational operations, the proposed approaches are efficient in both terms, decreasing the number of sets of transactions that run sequentially, and the running time needed to complete the whole sets of transactions.

Experimental results have shown also, that using two bits instead of one for locking mechanisms, is more efficient in all versions, especially the last one, when we have from 11 to 15 operations per transaction. In conclusion both the fact of atomicity of transactions and assigning a thread for each transaction and not for each operation within transaction, can simply explain the results.

Future work includes, assigning a thread for each operation within a transaction. Future work also includes conducting experiments on a large scale database independent of the number of operations or the percentage of accessing the database.

## REFERENCES

[1] Abbass, Jad. (2008) *Array-based Locks for a Concurrency Control Model.* Lebanese American University. Master Project.

[2] Bernstein P.A. and Goodman N. (1984). *An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases*. ACM Transactions on Database Systems, 9(4):596–615.

[3] Bernstein P., Brodie M., Ceri S., DeWitt D., Franklin M., Garcia-Molina H., J. Gray, J. Held, J. Hellerstein, H. V Jagadish, M. Lesk,D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman. (1998). *The Asilomar Report on Database Research*. Technical Report MSRTR-98-57, The Microsoft Corporation.

[4] Elmasri R. and Navathe S. (2003). *Fundamentals of Database Systems*. Addison Wesley.

[5] Gançarski S., Naacke H., Pacitti E., Valduriez P. (2002). *Load Balancing of Autonomous Applications and Databases in a Cluster System, Parallel Processing with Autonomous Databases in a Cluster System*. International Conference of Cooperative Information Systems (CoopIS).

[6] Gray J. and Reuter A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

[7] Hakimzadeh, H. (1992), *ROLL Concurrency Control*, Computer Science Department, technical report Number: NDSU-CSOR-TR-1992-20). North Dakota State University, Fargo, ND.

[8] Herlihy, M.P. and Wing, J.M. (1990). "Linearizability: A Correctness Condition for Concurrent Objects." ACM Transactions on Programming Languages and Systems. 12, 3, 463-492.

[9] M Tamer Ozsu and Patrick Valduriez. (1999). *Principles of Distributed Database Systems*. Prentice Hall.

[10] P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing: For the Systems Professional.* Morgan Kaufmann.

[11] Perrizo W., Hakimzadeh H., Haraty R., and Panda B. (1992). *A Concurrency Control Model for Multilevel Secure Object-Oriented Databases*. AAAI-IEEE-ACM-ISMM-International Conference of Information and Knowledge Management. Baltimore, MD

[12] Stonebraker M. (1979). *Concurrency Control and Consistency of Multiple Copies of Data in Distributed Ingres*. IEEE Transactions on Software Engineering, 5(3):188–194.

[13] Thomas R. H. (1979). *A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases*. ACM Transactions on Database Systems, 4(2):180–209.

*Acknowledgements*: This work was sponsored by the Lebanese American University.

TABLE 1. TEN EXPERIMENTS ON VERSION 1

| | Total Number | | | Average | | | Phase I | | | Phase II | | | Phase III | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Experiment # | Operations | Queries | Updaters | Opers/ Trans | Queries/ Trans | Updaters/ Trans | Number of sets | Transactions/ Set | Running Time | Number of sets | Transactions/ Set | Running Time | Number of sets | Transactions/ Set | Running Time |
| Average | 59.1 | 52.6 | 6.8 | 2.955 | 2.615 | 0.34 | 20 | 1 | 9910.4 | 6.8 | 3.2658 | 4205 | 3.9 | 6.7332 | 3150.2 |

TABLE 2. RESULTS FROM TEN EXPERIMENTS ON VERSION 1

| | From Phase I to II | | From Phase II to III | | From Phase I to III | |
|---|---|---|---|---|---|---|
| | Number Of Sets | Running Time | Number Of Sets | Running Time | Number Of Sets | Running Time |
| Decreasing In Numbers | **20 to 6.8** | **9910.4 to 4205** | **6.8 to 3.9** | **4205 to 3150.2** | **20 to 3.9** | **9910.4 to 3250.2** |
| Decreasing In Percentage | **66%** | **57.569%** | **42%** | **25.084%** | **80.5%** | **67.204%** |

TABLE 3. TEN EXPERIMENTS ON VERSION 2

| | Total Number | | | Average | | | Phase I | | | Phase II | | | Phase III | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Experiment # | Operations | Queries | Updaters | Opers/ Trans | Queries/ Trans | Updaters/ Trans | Number of sets | Transactions/ Set | Running Time | Number of sets | Transactions/ Set | Running Time | Number of sets | Transactions/ Set | Running Time |
| Average | 160.6 | 144 | 16.6 | 8.03 | 7.2 | 0.83 | 20 | 1 | 9005.1 | 17.4 | 1.1601 | 8710 | 12.8 | 1.6074 | 6966.1 |

TABLE 4. RESULTS FROM TEN EXPERIMENTS ON VERSION 2

| | From Phase I to II | | From Phase II to III | | From Phase I to III | |
|---|---|---|---|---|---|---|
| | Number Of Sets | Running Time | Number Of Sets | Running Time | Number Of Sets | Running Time |
| Decreasing In Numbers | 20 to 17.4 | 9005.1 to 8710 | 17.4 to 12.8 | 8710 to 6966 | 20 to 12.8 | 9005.1 to 6966 |
| Decreasing In Percentage | 13% | 3.277% | 26.436% | 20.022% | 36% | 22.643% |

TABLE 5. TEN EXPERIMENTS ON VERSION 3

| | Total Number | | | Average | | | Phase I | | | Phase II | | | Phase III | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Experiment # | Operations | Queries | Updaters | Opers/ Trans | Queries/ Trans | Updaters/ Trans | Number of sets | Transactions/ Set | Running Time | Number of sets | Transactions/ Set | Running Time | Number of sets | Transactions/ Set | Running Time |
| Average | 261 | 232 | 29.1 | 13.05 | 11.595 | 1.455 | 20 | 1 | 9685.5 | 20 | 1 | 8225 | 16.4 | 1.2375 | 8369.7 |

TABLE 6. RESULTS FROM TEN EXPERIMENTS ON VERSION 3

| | From Phase I to II | | From Phase II to III | | From Phase I to III | |
|---|---|---|---|---|---|---|
| | Number Of Sets | Running Time | Number Of Sets | Running Time | Number Of Sets | Running Time |
| Decreasing In Numbers | 20 to 20 | 9685.5 to 8225 | 20 to 16.4 | 8225 to 8369.7 | 20 to 16.4 | 9685.5 to 8369.7 |
| Decreasing In Percentage | 0% | 15.079% | 18% | -1.759% | 18% | 13.585% |

TABLE 7. COMPARISON OF THE THREE VERSIONS

| | | From Phase I to II | | From Phase II to III | | From Phase I to III | |
|---|---|---|---|---|---|---|---|
| | | Number Of Sets | Running Time | Number Of Sets | Running Time | Number Of Sets | Running Time |
| Version I | Decreasing In Percentage | 66% | 57.569% | 42% | 25.084% | 80.5% | 67.204% |
| Version II | | 13% | 3.277% | 26.436% | 20.022% | 36% | 22.643% |
| Version III | | 0% | 15.079% | 18% | -1.759% | 18% | 13.585% |