International Journal of Software Engineering and Its Applications
Vol.12, No.2 (2018), pp.31-46
http://dx.doi.org/10.21742/ijseia.2018.12.2.03

Mathematical Model and Algorithms for some Type of Concurrency Control Problems in Database Applications

Moukouop Nguena Ibrahim[1*] and Ngassa Kaam Cedric[2]

[1]*The National Advanced High School of engineering, Yaoundé Cameroon, CO 8390*
[2]*Megasoft Cameroon, Yaoundé, CO 7136*
[1]*imoukouo@gmail.com*, [2]*cngassamegasoft@gmail.com*

Abstract

Despite the extensive literature describing mechanisms of transactions management and concurrencies, it seems difficult for developers to apply these concepts. We analysed many open source application and we observed that the algorithms used to tackle some of these problems do not apply to concurrencies. These applications are used to treat business processes in many organisations, consequently, aforementioned issues lead to disastrous results and huge losses. It is difficult to find existing literature on algorithm addressing some classical problems of data management related applications, and mathematical models formalizing these problems at high level of abstraction. This article presents a general mathematical model to solve a group of database application related problems, which are designated as account management problems thereafter. It also presents general solutions to aforementioned problems, based on an appropriate combination of transaction isolation levels and update modes used for resolving. The proof of correctness of the algorithm and a performance analysis are done under some reasonable assumptions. The proposed solution is validated through some simulations and real implementation on number of projects.

Keywords: *Mathematical model, database application concurrency control, account management, enterprise resource planning*

1. Introduction

One of the major problems facing by financial or accounting systems (and raised by Weikum [1], see also (Hidouci [2]) is that of concurrent update on an account without facing bugs or poor performance.

In a bank, every customer holds an account which can be credited (money deposit) or debited (money withdraw). A movement of a given customer i, therefore consists of either crediting or debiting his account. At any time his account balance s can be obtained by subtracting the overall debit (outputs) to the aggregate credits (inputs). If we assume that the maximal debt allowed to him is d, then at any time the condition under which a transaction or movement is possible should be $s+d \geq 0$. A suitable procedure to ease the presentation of banking operations consists of designing a table, called movement table, with (at least) five entries, namely id, account, debit, credit, and date, where 'id' is the movement identifier, 'account' is the account identifier of the movement, 'debit' is the "value" of the debit component of the movement (whose worth is the amount of the debit if the transaction is a debit and 0 if the transaction is a credit), and 'credit' is the "value" of the credit component of the movement (which worth 0 if the transaction is a debit and worth the amount of the credit if the transaction is a credit) and 'date' is the date of the

Received (August 18, 2017), Review Result (November 10, 2017), Accepted (February 10, 2018)
* Corresponding Author

ISSN: 1738-9984 IJSEIA

movement. For a customer, debiting his account meanwhile the movement condition is not satisfied implies a loss to the bank. It is therefore very important to any bank to ensure that the movement condition is always satisfied before allowing any movement (namely any debit transaction).

A somehow different but similar problem, encountered in many structures is that of instance count where the purpose is to count, in a given table the exact number of rows verifying a given condition, in order to build a unique key. Assume for instance that each record of a given table is assigned a unique number such that two different records are assigned to different numbers. Trying to give sequence numbers by using the count (.) aggregation will lead to duplicates. When the sequence number is determined as count (.) +1, if two processes compute simultaneously the sequence number, they might create different records with the same sequence number. One can solve this problem by detecting collisions and restarting operations, but there is still a poor performance problem for the count (.) aggregation.

Many similar problems can be found in various institutions or companies. These problems include, but are not limited to those raised above, budgets, inventory, telephone subscriber account management and many others. In the sequel, these problems will be referred to as Account Management Problems (AMP). AMP deal with efficient concurrent update of accounts under certain constraints while avoiding concurrency bugs. As we will see in the sequel, addressing this issue in an efficient way is thus a huge challenge.

An attempt to address the problem of concurrent update on an account consists in the following. Before allowing any debit m in an account, we compute the quantity s1=sum(credit on account)-sum(debit on account)-m in order to make sure that the movement condition will be satisfied after the operation, and the operation is proceeded if and only if s1 is greater than or equal to zero.

This approach is not optimal because two processes might try to do a debit at the same time, such that s1 calculated by each process is ≥ 0, and $s+d<0$ after the execution of the two processes. For example, suppose that s=10, d=0, and two processes are simultaneously executing a debit of 10 on the same account. The condition $s1 \geq 0$ will be satisfied for the two processes, but after their execution, one will have $s+d<0$ which is violating the movement condition. This is very dangerous, because it allows a customer to withdraw more money than the amount available in his account. Moreover, the permanent sum aggregation is a costly operation and may lead to poor performance. A way to address this issue is to create another table called account table, and write codes in order to maintain account conditions. This solution leads to a transactional problem, because, we need to update the two tables (account and movement) in every operation.

Using open source software, Abbaspour *et al.*, [3] recently showed that concurrency bugs are common and are more severe than other bugs. They categorized concurrency bugs into seven disjoint classes: Deadlock, Livelock, Starvation, Suspension, Data race, Order violation and Atomicity violation [4]. Deadlock, Livelock and Starvation bugs lead to deny of service or poor performance, while Data race, Order violation and Atomicity violation bugs lead to incoherence. Incoherence arises when two or more concurrent transactions result in the loss of information of at least one of the transactions. For instance, one recharge card is used to credit two different subscribers' accounts by reloading simultaneously from two telephone sets. This case has been observed and led to huge losses for telecommunications operators.

The trivial solution to avoid incoherence leads to poor performance. The trivial solution serializes transactions, which on one hand leads to the poor use of server resources and increases time for many transactions, and on the other hand can lead to other concurrency bugs such as deadlocks and starvation.

These challenges remain viable for developers up-to-date. To the best of our knowledge, there is no general solution to this problem, with a performance analysis of the solution and a proof that it avoids common concurrency bugs.

Several studies focus on transaction management in general [1],[2],[5[,[6],[7],[8],[9] but do not provide any specific solution to account management problems. Some studies are related to the definition and the implementation of transaction isolation levels [10],[11]. Many authors provide tools to solve concurrency control problems at a more general level [12], [13]. For example, Weikum and Vossen [1] present some families of solutions used at a global level for solving concurrent access problems. These families include two-phase locking, timestamp ordering, multiversion timestamp ordering and optimistic concurrency control. However, because of their constraints, timestamp ordering techniques are usually implemented at the dbms level and not at the application level. Two phase locking can be implemented at the application level, but we don't have a description and a formal analysis of a two phase locking algorithm to solve AMP problems. Generally, solutions to concurrency control problems are not presented with algorithms which can be directly applied on AMP, and therefore, there is no rigorous performance analysis of the proposed solutions. Consequently, software developers usually write incorrect or serialized codes for AMP, despite their knowledge on transactions. For instance, we studied many open source software (Enterprise resource planning, inventory management…) and observed that codes for AMP allowing concurrent access to data are incorrect.

In this article we introduce a general mathematical formulation for account management problems and provide an algorithm relying on optimistic concurrency control to solve these problems. The proof of correctness of the algorithm and a performance analysis are done under some reasonable assumptions. We prove that the solution avoids common concurrency bugs. In order to prevent deadlocks, we introduce and formalize a constant order access rule and a notion of consistent order for resolving. The proposed solution is validated through some simulations and real implementation on number of projects.

Section 2 presents concepts which are used in subsequent sections: preventable phenomena, transaction isolation levels, the deadlock rule and the update mode of a dataset. At the end of this section, we introduce a formal definition of account management problem. Section 3 focuses on the presentation of solutions. We introduce therein the constant access order rule, the consistent order for resolving, and a solution for AMP. We also present a proof of the correctness of this solution, and its ability to avoid the common concurrency bugs presented by Abbaspour [4]. We further provide an analysis of the performances of the proposed solution as well as an example case with simulations. We carry out simulations using MySQL DBMS, and study the impact on the overall performances of a variation at the transaction isolation level. At the end of this section we present experimental tests on real cases. Section 4 and 5 are devoted to discussion and conclusion respectively.

2. Materials and Methods

2.1. Concepts

Preventable phenomena

According to Oracle [14] the ANSI/ISO SQL standard (SQL92) defines four levels of transaction isolation. These isolation levels are defined in terms of three phenomena that must be prevented between concurrently executing transactions [14].
The three preventable phenomena are:

- **Dirty reads**: A transaction T2 reads data that has been written by another transaction T1 that has not been committed yet. The problem here come from the possible rollback of the transaction T1, implying false data for T2.
- **Non repeatable (fuzzy) reads**: A transaction T1 re-reads data it has previously read and finds that another committed transaction T2 has modified or deleted the data.
- **Phantom reads (or phantoms)**: A transaction T1 re-runs a query returning a set of rows that satisfies a search condition and finds that another committed transaction T2 has inserted additional rows that satisfy the condition.

Transaction Isolation Levels

Little known and often misused, isolation levels of transactions enable transactions executing simultaneously to be more or less sealed [15], [16], [17].

SQL92 defines four levels of isolation in terms of the phenomena a transaction running at a particular isolation level is permitted to be experienced. They are shown in table 1.

Table 1. Preventable Read Phenomena by Isolation Level [14]

Isolation level	Dirty read	Non repeatable read	Phantom read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

The transition to a higher isolation level causes locks of which granularity, or retention increases. Increasing the isolation level will therefore increase the duration of the lock and the probability of obtaining deadlocks [14].

Transaction Isolation Levels Implementation and Deadlocks

At the application level, the developer must use transaction isolation levels in order to avoid data inconsistencies resulting from preventable phenomena. At the database level, transaction isolation levels are usually implemented using locks. Locks are mechanisms that prevent destructive interactions between transactions accessing the same resource [13]. Bearing in mind that, locks usually cause all further transactions to pause, waiting for the completion of the transaction which is the source of the lock, they can have a dramatic impact on performance, and even leading to deadlocks. A deadlock is when two transactions are each holding a lock on a row that the other requires to continue. This brings in a situation where the two transactions are waiting on each other to release the lock, which will obviously never happen since they are both waiting on each other. A higher transaction isolation level is more prone to deadlock than a lower isolation level [13].

The deadlock rule

The following conditions are required to cause a deadlock [16]:

1) Condition of mutual exclusion: each resource is either assigned to only one process or available

2) Detention and waiting condition: processes with resources can request for new ones.

3) No requisition: resources assigned to a process cannot be removed by force. They have to be free explicitly by the process holding them.

4) Circular waiting condition: there have to be at least two processes, each waiting a resource held by the process.

Generally, DBMS are able to detect a deadlock and break it by stopping one of the transactions involved in the deadlock and making a rollback on this transaction. Therefore, developers must be able to detect a transaction failure due to a deadlock and restart the transaction.

The Update Mode of a Dataset

We admit that data of the proposed algorithm are stored and processed into a dataset in memory. A dataset is a memory object that can contain rows of a table, while maintaining their structure or object form. The dataset is associated to a resolver which sends updates made in the dataset to the database. The resolver builds an update query depending on the update mode determined by the dataset.

The update mode allows DBMS to make a version control on data it updated in the database. It is determined by a selection criterion of what should be updated. In case this criterion is not verified, an error will be generated. The developer has the opportunity to indicate on which column the DBMS has to perform version control during the online update using the update mode. At least four options can be used to define the selection criterion:

KEY_COLUMNS: during the update of a record, search for it using solely its identifier

CHANGED_COLUMNS: during the update of a record, search for it using its identifier and the values of columns modified by the executing transaction. Every value of the modified column should be that which has been read.

ALL_COLUMNS: during the update of a record, search for it using all its columns (every value of each column must be that which had been read).

TIMESTAMP: during the update of a record, search for it basedon its identifier and on the value of its TIMESTAMP column which contains the time of the recent update, which has to be similar to that which had been noted during the record reading.

When none of these options is chosen, the default selection criterion used is KEY_COLUMNS

The use of datasets permit all additional processing which are expected to occur due to a change or any other reason without affecting the proposed algorithm. It also eases the computation of digital signatures. Developers who do not use datasets should build queries using consistent update criterion with the chosen update mode.

Mathematical Formulation of the Account Management Problem

As we have observed, the literature abounds with tools and not solutions in the form of algorithms studied for the resolution of application problems relating to transactions. This can come from the fact that these problems can be grouped into several classes with distinct solutions. In order to propose usable solutions, it is necessary to focus each time

on a specific class of problems, for which a general solution can be provided. In this article, we are interested in the class of AMP.

In order to characterize this class in a general way and to perform a rigorous analysis of the proposed solution, we propose a mathematical formulation for the AMP. To illustrate this formulation, we will use the case of the bank account. In this case, we have an account table that contains the customer account information, and a transaction table, which contains the movements of the accounts. The simplified schema of these tables is as follows:

Compte (numero_compte, credit, debit)

Movement (number_movement, number_count, credit, debit)

We can then formalize the problem as follows:

1) Let Db a database, and T(Db) the set of the tables in the database Db.

2) Let A be the account table, and B be the movement table

3) Let f(A) the set of columns of table A, and r(A) the set of rows of table A

4) Let P(B) the set of subsets of r(B)

5) Let a be the application associating to a row of table B the corresponding account in r (A). $a : r(B) \Rightarrow r(A)$

In the case of the bank account, for a movement x, a (x) gives the account whose field "numero_compte" has the same value as the field "numero_compte" of the movement x. Let denote $h = a^{-1}$, the reciprocal application of a, which associates to a given account all the movements relating to this account. We have:

$h : r(A) = >P(B)$

$\forall x \in r(A)$, $h(x)= \{y \in r(B)/ a(y)=x\}$

We admit the following constraint corresponding to the uniqueness of the account, meaning that a movement corresponds to only one account:

$$\forall x,y \in r(A),\ x \neq y \Rightarrow h(x) \cap h(y) = \varnothing \quad \text{(uniqueness of the account)}$$

6) Let $n \geq 1$ be a natural integer. We suppose that A contains n fields c1, c2,...,cn whose values are obtained by aggregating the values of the fields of the corresponding movements in B. In the case of our example, the mentioned fields above are the debit and credit fields. Let g1,...,gn be the applications that calculate these aggregations. We have.

$$\forall i=1,\ldots,n,\ \forall x \in r(A),\ x.ci - gi(h(x)) = 0.$$

We will call these constraints the aggregate invariants.

7) In the normal operation of a bank account say ac, the balance s = credit-debit must not be less than an authorized threshold s(ac), which depends on the account concerned. This translates into the equation

$$credit - debit - s(ac) \geq 0$$

Any movement resulting in the violation of this condition is prohibited. To preserve the generality, let define the application w related to the movement condition, such that:

$w : r(A) = >\Re$

$\forall x \in r(A)$, $W(x) \geq 0$ (movement condition)

In the sequel, we designate by account constraints the following constraints: the uniqueness of the account, the aggregate invariants and the movement condition.

Given a transaction O on table B (O can lead to a record creation, modification or deletion) how can O be realized while allowing concurrent access in order to maintain A and B consistent with account constraints after transaction O? What is the performance of O? How do we avoid concurrency bugs during the operation O?

3. Results and Discussion

3.1. The Constant Access Order Rule, Basic Version

In order to avoid deadlocks or reduce them, some rules must be observed. One of the most important rules is the "constant access order rule". Let us introduce a formal definition of this rule.

Giving two tables A and B, all transactions using A and B make their modifications (update, insert or delete) on these tables in the same order (always A before B or B before A). More generally, there is a total order relation ≤ between tables of the database, such that for any transaction updating A and B, if (A≤B), modifications on A must be done by this transaction before modifications on B.

Let Db a database and A, B ∈ T(Db). For x,y ∈T(Db) Let Tr(x) the set of the transactions using x and Tr(x,y) the set of the transactions using x and y. Let Sr(Db) the set of the total order relations defined on T(Db).Let Srr(x) the set of the total order relations defined on r(x).

The two following conditions are enough in order to avoid deadlocks:

1) Table ordering: $\exists r1 \in Sr(Db)$ /$\forall x,y \in T(Db)$, $\forall tr \in Tr(x,y)$, $x \leq_{r1} y \Rightarrow$ tr modifications on x are done before tr modifications on y.

2) Row ordering: $\forall x \in T(Db)$, $\exists r2 \in Srr(x)$, $\forall tr \in Tr(x)$, $\forall a,b \in r(x)$, $a \leq_{r2} b \Rightarrow$ tr modifications on a are done before tr modifications on b.

Proof:

If conditions 1 and 2 are satisfied, it is not possible to have a "circular waiting".

If conditions 1 and 2 are satisfied one can define a total order relation between two rows of the databases, even if they are not in the same table.

Let $x,y \in T(Db)$, $a \in r(x)$, $b \in r(y)$,

Let's define the relation ≤ between all rows of the database such that $a \leq b \Leftrightarrow x < y$ or $(x=y)$ and $a \leq_{r2} b$,.

It is obvious that the relation ≤ is a total order relation, and that resources are locked by all transactions according to the relation ≤.

Let tr1 and tr2, two transactions. Suppose that tr1 locks the resources a and b, with $a \leq_{r2} b$. Suppose that tr2 locks the resource b. It is not possible, due to the relation ≤, for tr2 to request the resource a in the future, because it must always requests a before b.

3.2. Definition: consistent order relation for resolving

Given a transaction T and an order relation r between datasets or tables involved in the transaction T, r is consistent for resolving if changes made in the datasets can be resolved to the database in the order defined by r, without facing problems due to respect of the integrity constraints of the database. For example, when there is a master-detail relation between two tables A and B, deleting rows on the master table before the detail table is not consistent for resolving. Master row deletion will automatically delete details, and consequently, detail deletion during resolving will fail.

3.3. A solution when there is no sequence problem

We are working in the case of a transaction O, and we want to resolve changes to the databases, after user manipulations on the datasets.
We make the following assumptions:

3) O does not create or delete records in the account table A (it is usually the case in an AMP).

4) For every table x involved in O, there is a total order relation $\leq x$ defined on r(x)

5) There is a total order relation rr consistent with O for resolving between datasets involved in O.

6) Just after a load from the database using a dataset, this dataset contains only loaded rows.

7) There is a total order relation defined on r(B), called the sequence order of r(B).

8) There is no sequence constraints, ie there is no field in B which is modified by O and which value for a row x

    a. depends on the values of the records created before x,

    b. cannot be found using only the row x and the row of A representing the account of x

9) There is no other lock than locks due to database isolation levels, concurrency and synchronization.

Therefore, we can use the following algorithm as a solution to the AMP

1) set autocommit to false;

2) int MAX_ATTEMPTS=10;

3) int nbe=0;

4) set transaction isolation level to one of the levels of the set (READ_COMMITTED, SERIALIZABLE, REPEATABLE_READ)

5) while(true){

    a. try{

    b. Load in the dataset Da all records of the table A which will be modified by O

    c. Sort the records of the dataset Da according to the relation $\leq A$

    d. set the update mode of Da to CHANGED_COLUMNS

    e. Load in the dataset Db all existing records of the table B which will be modified by O

    f. Create in the dataset Db all new records of the table B, created by O

    g. set the update mode of Db to CHANGED_COLUMNS

    h. Sort all records of the dataset Db using the derived order from A as first sorting criteria and their sequence order as second sorting criteria.

i. Create other datasets necessary for O, load them with appropriate data, set their update mode to CHANGED_COLUMNS, and sort each dataset records.

j. For every row x ∈ Db

a. If x is modified by O, modify x according to O

b. Find y ∈Da, such that x∈h(y) (find the account of the movement x)

c. If y is found,

i. For int i=1 to n, set y.ci=gi(h(y)∪x)

ii. check the movement condition for y. If it is not satisfied, throw a Movement Condition Exception

d. If x was loaded from the database and modified, let xp the version of x before modification by O

e. Find yp∈Da, such that xp∈h(yp) (the previous account of the movement)

f. If yp is found and yp≠y,

i. For int i=1 to n, set yp.ci=gi(h(yp)\xp)

ii. check the movement condition for yp. If it is not satisfied, throw a MovementConditionException

g. If applicable, update other datasets according to O.

6) Resolve all datasets to database, sorted according to rr, with respect to row order relation for each table, in one transaction.

7) Break;

8) }

9) Catch(SqlException exp){ //we assume that failure of the resolving throws a Sqlexception

10) Make a rollback;

11) nbe++;

If(nbe>MAX_ATTEMPTS) throw new Exception("impossible operation at this time. Please try later");

}

} (end while).

3.4. Proof of the solution

Theorem 1: using only transactions of type O, it is not possible to have a deadlock

Proof: due to the ordering of rows in the datasets and of tables before resolving, all objects are accessed in the same order in the database by all processes executing the transactions of type O. Therefore, circular waiting is not possible.

Theorem 2: concurrent modification will lead to failure or to a consistent database

Proof: Let P_1 and P_2, two processes executing transactions of type O, and working with a common account, denoted CA. Let's assume that the process P_1 is the first process to start the resolving. Let's denote P_{is} the start time of the process P_i and P_{if} the end time of the process P_i(commit or rollback).

The following scenarios are possible:

S1) $P_{1s}P_{2s}P_{1f}$ P_{2f}

S2) P_{1s} $P_{2s}P_{2f}$ P_{1f}

S3) P_{1s} P_{1f} P_{2S} P_{2f}

For the first scenario, the process P_1 is committed or rolled back before the end of the process P_2. If P_1 is rolled back, P_2 will not be impacted by P_1, due to the fact that the translation isolation level is at least the read committed level. If P_1 is committed, the account field af of the account CA is modified. Because the translation isolation level is at least the read committed level, the process P_2 have read the committed value of af prior to the update by P_1, or is locked, waiting for the end of P_1 before the reading of the field af. If the process P_2has read the last committed value, the resolving of the transaction P_2 will fail, due to the change of the field af and the fact that the update mode of all datasets is changed_column. Let's recall that with the update mode changed_column, the update fails if one of the modified columns was changed in the database. If the process P_2 were locked waiting for the end of P_1 before reading its values, it will read its values after the commit of P_1, and work is the sequel without concurrency with P_1.

For the second scenario, P_2 is either committed or rolled back. If P_2 is committed, P_1 will fail. If P_2 is rolled back, P_1 is not be impacted by P_2 and will succeed or rolled back.

The third scenario is similar to a serialization. There is no problem of concurrent access with this scenario.

This theorem proves also that using this algorithm, there is no Data race, Order violation or Atomicity violation bugs.

Theorem 3: if the account conditions were satisfied before a transaction, they are also satisfied after this transaction

Proof:

a. The uniqueness of the account is always verified, by definition

b. The aggregate values are maintained by the conditions of the step j of the algorithm

c. The movement condition is also checked and maintained at the step j

3.5. Performance Analysis of the algorithm

General analysis. The main questions here are:

1) Isn't it possible for a transaction to always fail?

2) What is the mean number of failures before the success of a transaction?

3) Isn't it possible to have a situation in which all transactions will always fail?

If every transaction implies only one account, it is obvious that case 3) is impossible.

Let's define the solicitation rate of an account as the probability of a transaction to use this account. For a transaction, let's define its collision appetite Ca as the probability of another transaction to use one of the accounts involved in this transaction. It is natural to

think that between two transactions, the transaction with the higher collision appetite has a greater probability to fail many times before succeeding. Let's note that a collision appetite of a transaction can be increased by involving a great number of accounts in this transaction.

It seems preferable to update the account dataset before the movement dataset, because in practical problems, accounts are more prone to concurrency than movements. If possible, in the case of high concurrency, it is also a good idea to sort accounts according to their solicitation rate, starting with the account with the highest solicitation rate. If this tip is used, solicitations rates must be kept constant during defined periods of time, in order to have the same order for all processes.

Performance analysis in the case of finite set of submitters: Let's define a submitter as the man who initiates a transaction. Let's assume that the maximum number of "submitters" has a constant value c. It is a reasonable assumption in many cases where transactions are not initiated from Internet (stores, SME…). Even if the transactions are initiated from Internet, it is possible to configure servers in order to deny more concurrent transactions than a defined maximum.

Let's also assume that every transaction involves a limited number of accounts lmax, such that one can define a time t1 as the maximal duration of a transaction processed alone, *i.e.*, without concurrency. Knowing that in real problems it is rare to have a transaction involving more than 20 accounts, this assumption is also reasonable. We assume that with a limited number of transactions, the system can scale linearly, *i.e.*, the maximum time to achieve c processes is less than

$$\mathrm{T} = (1 + \mathrm{c})\mathrm{t1} \tag{1}$$

Let's divide the time into smaller intervals of size T. Let's denote p the probability for a submitter to start a transaction during the period T. We want to find the mean number of failures of a process P1 before succeeding, assuming that the movement condition is always verified for all accounts involved in P1.

Let's denote Pe the probability of failure of the process P1. We admit that for P1 failure during the period T, at least one of the other submitters must start a transaction during this period, on one of the accounts involved in P1. Knowing that the probability for no submissions during the period T by all other submitters is $(1\text{-p}\,\mathrm{Ca(P1)})^{c-1}$, one can write

$$Pe \leq (1 - (1 - \mathrm{pCa(P1)})^{\mathrm{c}-1})$$

Let k an integer. The probability for P1 to fail exactly k times, assuming that each failure takes T seconds is

$$Pfk = Pe^{\mathrm{k}}(1 - Pe)$$

The mean number of failure is

$$Mnf = \sum_{k=0}^{\infty} kPe^{\mathrm{k}}(1 - Pe)$$

After some few calculations, one obtains

$$Mnf = Pe/(1 - Pe)$$

If we suppose that every submitter submits an average of s operations per second, knowing that the probability to submit an operation during the time T is p, one can write

$$s = p \times \frac{1}{T}$$

Which allow us to find p

$$p = s \times T = s \times (1 + c)t1 = S \times t1(1 + \frac{1}{c})$$

With S=s×c, the global number of operations submitted per second by all the submitters.

In most cases concerning SME, $Ca(P1) \leq \frac{1}{4}$ (there is no a dominant account involved in more than a half of all operations), S≤50/60 (it corresponds to 24000 submissions per day of 8 hours), t1≤10ms. Using c=99, we have p=0.0084, pe≤0.188 and Mnf≤0.231.

Note: t1 is small only in the case where the submitter is close to the server. In other cases, network delays can lead to a high value of t1. In a tree tier model, the submitter is the application server, and in a two tier model, the submitter is the client. Due to the fact that the application server can be very close to the database server, in order to have a small value for t1, the tree tier model is preferable in wide networks.

This analysis of the mean number of failures also allows us to conclude with the hypothesis of this solution, that guarded suspension, live lock and starvation are not possible using this algorithm in a context allowing low values of t1 (tree tier model or local network).

3.6. Example Case and Experimental Tests

Example case: Let's consider the inventory management problem in the context of a unique warehouse, described according to the following rules:

a) For each product we should be able to know the quantity of inventory, the total input and output. An inventory movement is prohibited if it leads to a negative stock.

b) Product movements are created by deliveries. One can have input delivery (purchase), or output delivery (sales).

Each delivery may involve one or many products. When a delivery is validated, inventory movements are automatically created in the system for the related product. Validating a delivery is a different operation from its registration.

A same product can be purchased at different unit price. Inventories are valued by the weighted average cost method. Inventory state (quantity and value) should be presented at any time (present or past).

c) Deliveries can be made with concurrencies in the system, and performances should remain adequate, with over 50 users connected and the volume of data corresponding to hundreds of thousands of lines (adequate for most SME).

The purpose here is to propose a conception and algorithms to enable the validation of deliveries in order to ensure consistency and meet specifications stated here.

Presentation of Simulation Tools

To implement algorithms proposed in this paper, we used the JAVA programming language and the MDAL framework.

MDAL is a Framework which eases programming, deployment, operating and maintenance of database applications [18].

Concurrencies were simulated using java threads.

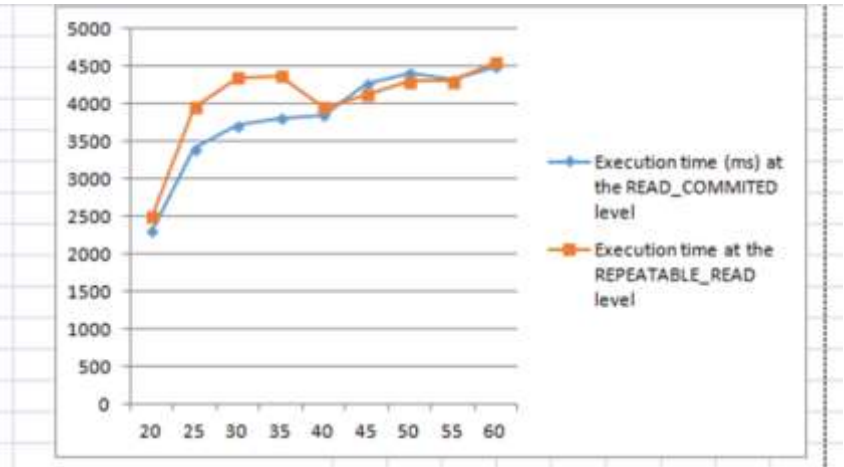
MYSQL is the database management system used.

Processing is made using:

A TOSHIBA computer core i3 with 10 GHz processor speed and 4GB of RAM and Windows Seven operating system.

A Dell computer core i3 with 10 GHz processor speed and 4GB of RAM and Windows Seven operating system.
A server with 8Go of RAM and a linux operating system (Redhat) host database.

Simulations and Results: Having worked in an environment with over twenty thousands (20,000) registered products and more than forty thousands (40,000) inventory movements, using twenty (20) to sixty (60) threads. As one can expect, using KEY_COLUMNS as update mode leads to wrong values of accounts in case of concurrency. Using the CHANGED COLUMNS update mode, all results are consistent, and we obtained the execution times of the following figures:



	20	25	30	35	40	45	50	55	60
Execution time (ms) at the READ_COMMITED level	2304	3401	3720	3816	3852	4265	4419	4321	4493
Execution time at the REPEATABLE_READ level	2497	3963	4354	4372	3958	4123	4302	4302	4556

Figure 1. Results Obtained using the CHANGED COLUMNS Update Mode

Test on Real Cases

These algorithms have been implemented in several real projects, among which we can mention two particular cases: the nexus project and budget management application.

The nexus system is used by the Cameroonian Customs to track goods in transit by geolocalization. Authorized customs brokers have accounts in Nexus. These accounts are credited by their payments and debited by the trips they make. The developers first wrote algorithms of their design for the management of the accounts. This resulted in numerous inconsistencies in the accounts, due to competing access to the creation of trips for operators.

The corresponding module has been rewritten, using the algorithms presented above. Since then, the problems have been solved, and the system has been functioning normally for several years, with very good performance, although the database is unique and accessible from all sites in the country.

The other example concerns a budget management software developed within the company Megasoft. The algorithms presented here have been used to manage all transactions on the accounts. Tests in the laboratory with simulation of competition, and tests in operation were made, and all resulted in positive results, confirming the good behavior of these algorithms.

Discussion

As expected, all results are good with the CHANGED_ COLUMNS update mode, using READ_COMMITTED or REPEATABLE_READ isolation levels. With a small number of threads, READ_COMMITTED takes less time than REPEATABLE_READ isolation level. From 45 threads, we noticed that the execution time frames are almost similar.

4. Conclusion and Outlook

With the aim of providing developers with a development approach to overcome the main challenges (inconsistency, poor performance, deadlock and starvation) they are faced with the problem of concurrent account update, in the state of art, in a detailed manner, we presented some problem type of account management, transactions and some open source software which manage accounts (inventory management, transfer management); then for each of these problems, we provided solutions based on the adequate combination between the selection criterion defined by the update mode and the isolation level. These solutions enabled us make simulations from which we observed that with an update mode criterion defined in CHANGED_ COLUMNS and the READ COMMITTED and REPEATABLE READ isolation levels, we have a reliable view of data and satisfactory answers with about 60 concurrent threads.

In the future, we will make simulations with the Serializable isolation level. We will also use other DBMS in order to find the impact of the isolation level on different databases. This work does not provide a solution neither for the sequence problem nor when one movement involves more than one account; we will also work on these cases. The case where there are other locks than concurrency locks must be also investigated. These cases are found in real cases, due to firewalls and security considerations.

Acknowledgments

I thank Mr. Njei Check for his help in correcting this document.

This work was supported by MegaSoft Cameroon and by the CETIC project of the National Advanced School of Engineering of Yaounde.

References

[1] G. Weikum, G. Vossen, Morgan Kaufmann publishers, “Transactional information systems, theory, algorithms, and the practice of concurrency control and recovery”, **(2002)**.

[2] W. K. Hidouci, “Systeme de Base de donnees avancees”, https://sites.google.com/a/esi.dz/hidouci/competences-professionnelles/bases-de-donnes-avances, Accessed (2016) November 05.

[3] S. Abbaspour Asadollah, D. Sundmark, S. Eldh, H. Hansson and E. Paul Enoiu, “A Study of Concurrency Bugs in an Open Source Software”, IFIP Advances in Information and Communication Technology, **(2016)** May, pp. 16-31.

[4] S. A. Abbaspour, H. Hansson, D. Sundmark and S. Eldh, “Towards classification of concurrency bugs based on Observable properties”, 1st International Workshop on Complex Faults and Failures in Large Software Systems, Italy, **(2015)**.

[5] S. Mahapatra, “Transaction management under J2EE 1.2.JavaWorld”, 2000, http://www.javaworld.com/article/2076126/java-se/transaction-management-under-j2ee-1-2.html, **(2000)**, 05 November 2016.

[6] J.-M. Doudoux, “Développons en Java, La gestion des transactions avec Spring”, https://www.jmdoudoux.fr/java/dej/chap-spring_transactions.htm, Accessed (2016) November 05.

[7] R. Ramakrishnan and J. Gehrke, “Transaction Management Overview”, https://www.coursehero.com/file/19316862/Ch16/, Accessed (2016) November 05.

[8] B. Schawrtz, P. Zaitsev, V. Tkachecnko, J. D. Zawodny, A. Lentz, E. Derek and J. Balling, Ed O’REILLY, High Performance Mysql, second edition **(2008)**.

[9] A. K. Elmagarmid, “Transaction Models for Advanced Database Applications”, Computer Science Technical Reports. Paper 871. http://docs.lib.purdue.edu/cstech/871, **(1991)**.

[10] H. Berenson, P. Bernstein and J. Gray, “A critique of ANSI SQL isolation levels”, Proceedings of the 1995 ACM SIGMOD international conference on Management of data, vol. 24, no. 2, **(1995)** May, pp. 1-10.

[11] A. J. Bernstein, P. M. Lewis and S. Lu, "Semantic conditions for correctness at different isolation levels", pub Data Engineering, 2000. Proceedings. 16th International Conference, **(2000)**.
[12] R. J. Bamford and K. R. Jacobs, "Oracle Corporation, Method and apparatus for providing isolation levels in a database system", pub. Google Patents, https://www.google.com/patents/US5870758, **(1999)**. Accessed (2016) November 05.
[13] J. D. Richey, S. R. Avadhanam and Z. Chu, "Concurrency control within an enterprise resource planning system", https://www.google.com/patents/US7933881, pub. Google Patents, **(2011)**, Accessed (2016) November 05.
[14] Oracle Corporation http://docs.oracle.com/cd/B28359_01/server.111/b28318/consist.htm#CNCPT1313, Accessed 05 November 2016.
[15] F. Brouard, Developpez.com, Niveau d'isolation et anomalies transactionnelles, http://sqlpro.developpez.com/isolation-transaction **(2008)**, Accessed (2016) November 05.
[16] A.-C. Caron, Les transactions, http://www.fil.univ-lille1.fr/~caronc/SGBD/transPar4.pdf, Accessed (2016) November 05.
[17] A. Tanenbaum, Système d'exploitation, Interblocages, 2ieme Edition, Librairie Eyrolles, **(2001)**, pp 169-196.
[18] Megasoft Sarl, Mdal presentation, http://www.megasoftcm.com/Produits/Logiciels/DAL/dal.html, Accessed (2016) November 05.

