CONCURRENCY CONTROL FOR TEMPORAL DATABASES

Achraf MAKNI and Rafik BOUAZIZ

Faculté des Sciences Economiques et de Gestion de Sfax
BP 1088, 3018 Sfax, Tunisia
Tunisia
{Achraf.makni, Raf.bouaziz}@ fsegs.rnu.tn

ABSTRACT

Several database (DB) applications are temporal of nature and require a special treatment. In particular, in the field of concurrency control (CC) which takes new dimensions when applied to temporal DB (TDB). The CC algorithms proposed for TDB have tried to find solutions for the CC to improve their performances. Especially, they have tried, by benefiting from the characteristics of the TDB, to decrease the degree of conflict, and this by the use of à priori knowledge or the reduction of the granule sizes. But these algorithms have not reached the fixed objectives. To contribute to the edification of a CC component suitable for TDB, we propose in this paper a complete study of access concurrency control in TDB. We have chosen to build our algorithms according to the optimistic methods, which are, in our opinion, more suitable for TDB than the pessimistic methods. Indeed, our optimistic algorithms can exploit the temporal specifications to reduce the granule size and then to minimize the conflict degree. Moreover, they can detect, as soon as possible, all the conflict cases. By using the end of transaction marker technique, they have the merit to reduce to the maximum the period during which resources are locked in the validation phase. To be sure that our algorithms operate correctly, we have carried out a formal verification, based first on the serialization theory and next on the SPIN model checker. Then, we have made a performance evaluation vis-à-vis of other well-known concurrency control algorithms based on optimistic and pessimistic approaches, to show that our propositions ameliorate the performances in the large majority of the cases.

KEY WORDS

Temporal DB, Concurrency Control, Formal Verification, SPIN Model Checker, Performance Evaluation.

1. INTRODUCTION

Several applications need to manage simultaneous access to data. They must avoid inconsistency in the shortest possible time. This is one of the most important challenges for a database management system (DBMS) when many transactions can have simultaneous access to the same data. The access concurrency controller is an essential component of a DBMS. It must ensure the database (DB) consistency, i.e. the guarantee that any simultaneous execution of transactions produces the same results as some sequential execution [11].

The concurrency control (CC) takes new dimensions when applied to temporal DB (TDB), which allow the management of data history. We can find in the literature some CC algorithms based on the pessimistic approach [10] [7] [8]. These algorithms have tried to find solutions for the CC within the framework of the TDB, while improving the performances. In particular, they have tried, by benefiting from the characteristics of the TDB, to decrease the degree of conflict, and this by the use of à priori knowledge or the reduction of the granule sizes. But these algorithms have not reached the objectives desired within the framework of the TDB.

The goal of this paper is to present a complete study of the CC for TDB using an optimistic approach. Knowing that the pessimistic approach of CC presents some disadvantages [8], on the one hand, and the optimistic approach has improved the parallelism degree in some evolved environments [1], on the other hand, so we have proposed optimistic algorithms suitable for TDB [5] [17] [18] [19] [20]. Our propositions were formally checked using both the serialization theory [2], which is a theoretical formalism used to check the CC algorithms, and the SPIN tool [13], which is one of the most powerful model checkers. SPIN is an appropriate tool for analyzing the logical consistency of concurrent systems. It is largely used, not only in the research areas, since it is freeware, but also in industrial ones [7] [6] [3].

Moreover, we have proceeded to a performance evaluation of our algorithms and of other pessimistic and optimistic ones. The goal of this work is to compare these algorithms and to choose the ones ensuring the best performance at a minimum cost for the TDB environment. We note that there has been a great deal of interest in the performance of CC algorithms in the literature in recent years. Most of these studies have been proposed for real-time systems [22] [16] [24].

This paper is organized as follows. In the next section, we present structure of TDB. We discuss after that, in section 3, related work and our contribution. In section 4, we present required elements of the concurrency control before starting the performance evaluation: the choice of granule size, the scheduling tasks of the algorithms, the conflict detection and the formal verification. We present, in section 5, the performance studies and, in section 6, the simulation results. Section 7 concludes our paper.

2. STRUCTURE OF TDB

Data item of a TDB can be stamped by their transaction-time and/or their valid-time. We use the acronyms TTR (transaction time relations), VTR (valid time relations) and BTR (bitemporal relations) to design relations where data are stamped with their transaction-time, their valid-time and both times, respectively.

TTR store data versions by stamping them using the transaction time start (TTS) and the transaction time end (TTE) which are generated by the DBMS [14]. In a VTR, data versions are stamped using the valid time start (VTS) and the valid time end (VTE). The time interval formed by VTS and VTE refers to the validity interval during which the data item exists in the real world. It is called *valid-time lifespan* [14] and supplied by the user.

To ensure a complete history, we must use BTR which reassemble the characteristics of both TTR and VTR. Indeed, data versions are stamped using the VTS, the VTE, the TTS and the TTE. It would be then possible to record the retroactive and postactive updates, on the one hand, and to view the state of the DB at any moment of past, and particularly the incoherent states, which are not destroyed.

Let us take the following example of the *Salary_Emp* BTR:

	Emp_num	**Salary**	**VTS**	**VTE**	**TTS**	**TTE**
V1	10	1000	06/10/1	08/3/31	06/9/2:8:8:3	06/11/9:9:3:5
V2	10	1200	06/10/1	08/3/31	06/11/9:9:3:5	08/3/31:23:59:59
V3	10	1300	08/4/1	09/10/31	08/2/2:3:6:8	!
V4	10	1450	09/11/1	10/9/30	08/5/1:2:8:6	!

The error concerning the salary value in the first tuple V1 was corrected at the moment "06/11/9:9:3:5". So, the corrective tuple V2 was introduced with the same validity interval. TTE of V1 and TTS of V2 have taken the same value corresponding to the update moment. V2 has ceased to be the current tuple when the current time has reached its VTE value. The tuple V3 was inserted in advance ($TTS_{v3}<VTS_{v3}$). Now, it is the current version. Like the tuple V3, the future tuple V4 was inserted in advance ($TTS_{v4}<VTS_{v4}$). The graphic representation according to the validity time and transaction time is shown in figure 3.

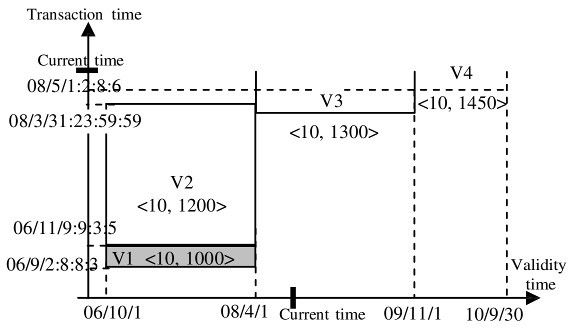

Figure 3. Graphic representation of the *Salary_Emp* BTR

We note that only the valid versions (V2, V3 and V4 in our example) are concerned by modifications and so by the concurrency controller.

In our framework [4], we indicate by:

- Generic Key (GK), the set of attributes which form the relation key without temporal consideration. In our example, the GK is Emp_num attribute.
- Generic Tuple (GT), each set of tuples which has the same value of generic key. In our example of the BTR relation, V1, V2, V3 and V4 belong to the same GT.

We describe, in the following, the semantics of modification operations of any transaction T_i, for a VTR and a BTR. These operations can be implemented by any temporal language.

- **Read (Rel, select_cond[, ts[, te]])**: T_i asks to read values of tuples of the relation *Rel* satisfying the selection condition (*select_cond*) during the time period [*ts*, *te*]. The concurrency controller receives the read message "*Read message (T_i, Rel, gk, vts, vte)*" for each selected valid tuple having "gk" as value of its GK. [*vts*, *vte*] define the time interval equal to the intersection of the time period specified in the read operation and the validity interval of this tuple.
- **Insert (Rel, gk, vts[, vte] [, attribute-name: data-value]*)**: T_i asks to insert in the relation *Rel* a new version having a GK equal to "gk". [*vts*, *vte*] is the validity interval of the new version. The concurrency controller receives the insert message "*Insert message (T_i, Rel, gk, vts, vte)*".
- **Delete (Rel, select_cond[, ts, te])**: T_i asks to delete from the relation *Rel* data satisfying *select_cond* and valid during the time period [*ts*, *te*]. The concurrency controller must receive the delete message "*Delete message (T_i, Rel, gk, vts, vte)*" for each deleted generic tuple.
- **Update (Rel, select_cond [,ts, te], attribute-name: data-value [,attribute-name: data-value]*)**: T_i asks to update data value satisfying *select_cond* and valid during the time period [*ts*, *te*]. The concurrency controller receives the update message "*update message (T_i, Rel, gk, vts, vte)*" for each updated generic tuple.

3. RELATED WORK AND CONTRIBUTION

3.1. Related work

The two categories of CC methods, pessimistic and optimistic, differ in two aspects: the time when they detect conflicts and the way that they resolve conflicts. Each one of these two categories has some disadvantages.

Earlier works concerning the CC methods proposed in the context of TDB are based on the pessimistic approach ([10] [7] [8]). The algorithm proposed in [7] is distinguished by the fact that it takes into account timeslice queries to avoid false conflict detection and then to ameliorate the time execution. Indeed, it considers that each transaction operates on a time interval and then the conflicts are detected according to this time interval. So, when two transactions operate in the same time interval, each one will be decomposed in two sub-transactions: $T^{validable}$ and $T^{conflict}$. Then, only the latter must be serialized with the other conflict sub-transaction. At the validation moment of each transaction, [7] proposed to use the 2PC (2 phases commit) technique in order to ensure that each sub-transaction is ready to be validated. Using this proposition, the transaction atomicity is revised [7].

The main contribution of the algorithms proposed in [10] and [8] for TTR is the maintaining of the strong consistency of the DB [23] to have consistent views of the DB [9] [15]. They stamp the transactions through their arrival moments and synchronize them according to this order. In order to reduce the time execution, the algorithm proposed in [8] and the improved version of the MO-2PL algorithm proposed in [10] need resource knowledge, which is difficult to implement.

3.2. A new approach is needed

There is a major limitation with the pessimistic approach, as well as an important aspect which was not well studied concerning the timeslice query in the case of VTR and BTR.

The main disadvantage of the pessimistic approach is the use of the transaction blocking technique in order to avoid inconsistency. This technique decreases the degree of parallelism, on the one hand, and can cause the appearance of the deadlock problem, on the other hand. These two disadvantages are avoided with the optimistic approach. Indeed, the contribution of the optimistic approach is appreciated in some evolved environments, like the real-time systems, where the degree of parallelism is improved ([12] [1]).

Moreover, as we have seen for delete and update operations in the case of VTR and BTR (as seen in §2.3), only valid data during the validity interval which corresponds to the intersection of the time period specified in the delete or update operation and the validity interval of the manipulated tuple, is modified. So, only this interval must be considered by the concurrency controller. This leads us to consider a granule as a time interval rather than a tuple. We think that this proposition is more suitable for VTR and BTR, since a query can be defined according to some portion of time. We also think that this proposal is more adequate for the optimistic approach than the pessimistic one. For this latter, the granule to be locked must be *a priori* known; as a tuple for example. It is more difficult to ensure this condition when the granule is defined as a temporal interval. In addition, this leads to decompose each transaction in two sub-transactions as proposed by [7] and to revise the concept of transaction atomicity.

The optimistic approach can also have disadvantages: high degree of transaction abortion and high length of the period during which resources are locked in the validation phase. However, we show in the next sub-section how we can avoid these disadvantages successfully.

3.3. Our contribution

To build our algorithms of CC, our contribution consists in:

- Adopting the optimistic approach since it can be considered as the most suitable in the context of TDB. We have started from the validation strategy of broadcast optimistic method with critical section [21], in order to detect conflicts as soon as possible.

- Using the end of transaction (EOT) marker technique to reduce, to the maximum, the period during which resources are locked in the validation phase.
- Considering the granule as a time interval for a GT rather than a tuple in order to avoid the risk of false conflict detection, and then to reduce the abortion degree. This proposal has the advantage of reducing the granule size to the minimum, without a need to decompose transactions. This can constitute an interesting track and leads us to consider the optimistic approach as the most suitable in the context of TDB.

Moreover, our contribution consists in proposing two kinds of optimistic algorithms: the first allows maintaining the strong consistency, used when the final results must be the same as the sequential execution in the strict order of transaction arrivals; the second is limited to ensure the consistency. Indeed, many applications require just ensuring that the simultaneous execution of transactions produces the same results as any sequential execution. This allows avoiding the wait for transactions arriving to their validation phase and then reducing the execution time.

In addition, our two optimistic algorithms are addressed to any type of temporal relation by carrying out the necessary treatment in each case. Indeed, they consider the suitable granule according to the type of relation.

4. Description of the Concurrency Control Algorithm

4.1. Choice of the granule size

For TTR, the smallest element which will be modifed when executing a modification operation is the tuple. Then we consider the granule as a tuple and not as a GT [5]. Since modification of past version is not authorized in TTR, only the current version is taken into account by the concurrency controller.

Definition: In the case of TTR, the CC granule is defined as follows:

Granule = (Rel, gk), with:

Rel: Relation name,

gk: generic key value.

But for VTR, the modifications of data item states are achieved in accordance with some portions of time. So, we have proposed to define a new type of granule, as the temporal interval in which a tuple is valid, rather than this tuple. This allows avoiding the risk of false conflict detections ([18] [19]). Indeed, the temporal interval in which a tuple is valid is always lower or equal to the whole validity interval of the tuple.

We can consider, for BTR, the same granule definition proposed for VTR. Indeed, the modifications of data item states of a BTR are achieved in accordance with some portions of time, on the one hand, and the same valid timeslice operation defined for VTR are applied for BTR in the same manner, on the other hand.

Let us consider the example of figure 3. After executing the update operation *Update (Salary_Emp, "Emp_num=10", 10/3/1, 10/9/30, salary: 1500)*, only valid data during [10/3/1, 10/9/30] are modified. So, only the validity interval [10/3/1, 10/9/30] must be considered by the concurrency controller. Our new granule, defined as a time interval, is then shorter than a tuple. In the worst cases, we consider that it is equal to the tuple. In the case where the validity interval of the updated operation overlaps many version validity intervals, it can be necessary to use many granules and not only one for this operation.

Definition: In the case of VTR and BTR, the CC granule is defined as follows:

Granule = (Rel, gk, ti), with:

Rel: Relation name,
gk: generic key value,
ti: time interval.

4.2. Tasks scheduling of the concurrency controller

An optimistic concurrency controller maintains for snapshot relations two sets for each transaction: a read set (RS) and a write set (WS). In the case of temporal relations, we continue to use one read set, but we propose to define three different write sets rather than one ([18] [19]), for the reasons given in §4.3.

Proposition: In order to avoid false conflict detection, the concurrency controller must maintain for each transaction T_i four sets:

RS_i (Read Set), the set of objects read by T_i,
IS_i (Insert Set), the set of objects inserted by T_i,
US_i (Update Set), the set of objects updated by T_i,
DS_i (Delete Set), the set of objects deleted by T_i.

To build our algorithms of CC, we start from the validation strategy of broadcast optimistic method with critical section [21]. Based on this strategy, a transaction T_e consists of three phases:

- A read phase during which the required objects are read from the DB and the write operation are performed on a local copy of the transaction, imperceptible for the other transactions.
- A validation phase during which the checking of conflicts is performed. This phase starts at the execution of the T_e commit order.
- A write phase during which the objects modified are written in the DB.

The validation and write phases form the critical section, during which manipulated objects are locked and thus can not be acceded by another transaction.

For each transaction T_e which wants to execute its commit order, the concurrency controller must validate T_e vis-à-vis of each concurrent transaction T_k which is not yet validated. If there is a conflict between T_e and T_k, the transaction having the lowest priority is aborted.

Let us keep in mind that, in the algorithms proposed in [5] and [18], the priority order is attributed according to the arrival order. So, the transactions are stamped by the moments of their arrivals and then the last coming one has the lowest priority. In order to ensure a validation order according to these stamps, we propose to add a certification phase which precedes the validation one of each transaction. In the certification phase, the concurrency controller must check, before starting the validation phase, that T_e has the highest priority regards to transactions which are not yet validated. If it is the case, the concurrency controller passes T_e to its validation phase. Otherwise, T_e is put in a waiting list to be certified later on. Then, the definition of a certification phase ensures the strong consistency. This algorithm is named OCCA_SC/TDB as an optimistic CC algorithm ensuring the strong consistency for TDB.

But if only the consistency is needed, it is sufficient to synchronize transactions by the moments of their validations [19]. This allows having an order of coherent states which corresponds to the order of validation of transactions. Thus, a transaction T_e which reaches its validation phase is considered as the transaction having the highest priority and will be automatically validated. If a conflict is detected with another concurrent transaction T_k, then T_k is aborted. This algorithm is named OCCA/TDB as an optimistic CC algorithm for TDB.

In addition, we propose an important amelioration for our optimistic algorithms [5] [18] [19]; this amelioration consists in reducing the period of the critical section during which all the manipulated granules in writing by T_e must be locked. This period extends normally during the two writing and validation phases of T_e. Based on the fact that the writing phase must be carried out during the critical section, we propose to place the validation phase out of the critical section and to integrate the "EOT marker" technique for a correct definition of the conflicts. Indeed, this period is considerably much shorter than the whole validation phase as shown by the experiment result in section 6. So, when a transaction T_e passes the certification test, the concurrency controller marks its end in the RS_k of each transaction T_k not yet validated. To check the absence of conflicts between T_e and each transaction T_k, it uses the $RS_k(T_e)$ which is the set RS_k limited to the objects read from the beginning to the end of transaction mark of T_e.

After receiving a read, insert, delete or update message from any transaction T_i, the concurrency controller adds to RS_i, IS_i, DS_i or US_i the corresponding granule.

After receiving a rollback or a commit message from any transaction T_i, the concurrency controller proceeds as follows:

- For a **Rollback message (T_i, Rollback)**, it eliminates RS_i, IS_i, US_i and DS_i.
- For a **Commit message (T_i, Commit)**, it checks if there is a conflict between T_i, the transaction to be validated, and the transactions which are not yet validated.

4.3. Conflict detection and proof

To prove our propositions, we will use the serialization theory [2] that we introduce as following. Suppose that a transaction T_r performs an operation set O_r which can contain data operations over a granule x (read: $R_r[x]$, insert: $I_r[x]$, delete: $D_r[x]$, update: $U_r[x]$) and terminating operations (commit: C_r, rollback: R_r). T_r is formalised as a partial order (T_r, $<_r$) where:

1. $T_r \subseteq O_r$;
2. $R_r \in T_r$ iff $C_r \notin T_r$;
3. If to_r is C_r or R_r (whichever is in T_r), for any other operation $o_r \in T_r$, $o_r <_r to_r$; and
4. If $o_{1r}[x], o_{2r}[x] \in T_r$, then $o_{1r}[x] <_r o_{2r}[x]$ or $o_{2r}[x] <_r o_{1r}[x]$.

Condition 1 defines the kinds of operations in the transaction T_r, which represents the set of its executed operations. Condition 2 says that this set contains a Commit or a Rollback, but not both. Condition 3 says that the Commit or Rollback must follow all other executed operations. Condition 4 requires that $<_r$ specify the order of execution of Read and Write operations on a common data item.

If $T = \{T_1, T_2, \ldots T_n\}$ is a set of transactions, a complete history H over T is a partial order with ordering relation $<_H$ where:

1. $H = U^n_{i=1} T_i$;
2. $<_H \supseteq U^n_{i=1} <_i$; and
3. For any two conflicting operations p, q $\in$ H, either $p <_H q$ or $q <_H p$.

Condition 1 says that the execution represented by H involves precisely the operations executed by the transactions $T_1, \ldots, T_n$. Condition 2 says that all operation orderings specified within each transaction is conserved in H. Condition 3 says that the ordering of every pair of conflicting operations is determined by $<_H$.

To show how we can detect conflicts between two transactions T_o and T_f, for each kind of operations, we suppose that T_o and T_f operate simultaneously, and T_o is the transaction having the highest priority (o < f).

The different order combinations of data operations of T_o and T_f are represented in the following matrix:

$T_o \backslash T_f$	R_f	I_f	D_f	U_f
R_o	$R_o \backslash R_f$	$R_o \backslash I_f$	$R_o \backslash D_f$	$R_o \backslash U_f$
I_o	$I_o \backslash R_f$	$I_o \backslash I_f$	$I_o \backslash D_f$	$I_o \backslash U_f$
D_o	$D_o \backslash R_f$	$D_o \backslash I_f$	$D_o \backslash D_f$	$D_o \backslash U_f$
U_o	$U_o \backslash R_f$	$U_o \backslash I_f$	$U_o \backslash D_f$	$U_o \backslash U_f$

The cases of $R_o \backslash R_f$, $R_o \backslash I_f$, $R_o \backslash D_f$ and $R_o \backslash U_f$ do not constitute conflict situations because R_o read the old state of the granule, since T_o is the first to be validated. Also, there is no conflict risk for $U_o \backslash I_f$, since I_f is always rejected, for $U_o \backslash D_f$ and $D_o \backslash D_f$, since the considered tuple is always deleted, and for $U_o \backslash U_f$ since the final result is the one produced by T_f. So, only the cases $D_o \backslash R_f$, $D_o \backslash U_f$, $U_o \backslash R_f$ and $I_o \backslash I_f$, can constitute conflict situations, and the cases $I_o \backslash R_f$, $I_o \backslash U_f$, $I_o \backslash D_f$ and $D_o \backslash I_f$ can produce operation failures.

These cases of conflict situations and operation failures are the same for TTR, VTR and BTR. We present, hereafter, some examples when the considered relation is the BTR. For TTR, the modification operations are defined without valid time interval and the granule is considered as a tuple.

Case of no conflict situation: Let us consider that T_o and T_f operate on the GT of the state of figure 3. Each one of these two transactions contains the following delete operation:

```
delete(Salary_Emp,"Emp_num=10",10/1/1,10/12/31)
```

Before the validation of T_o and T_f, the two delete operations are executed, and the concurrency controller receives from T_o the *delete message (T_o, Salary_Emp, 10, 10/1/1, 10/9/30)*, and from T_f the *delete message (T_f, Salary_Emp, 10, 10/1/1, 10/9/30)*. Then the delete sets maintained for T_o and T_f becomes as follows:

```
DSo ={(Salary_Emp,100,[10/1/1,10/9/30])}
DSf ={(Salary_Emp,100,[10/1/1,10/9/30])}
```

Suppose that T_o is the first to be validated. So, the delete operation is effectively done before the validation of T_f.

When T_f reaches its validation phase, the effect of its delete operation must not be reverberated in the DB, since the data to delete was deleted by T_o. However, there is no inconsistency, and so no effective conflict between these two operations.

Case of conflict situation: Let us consider that T_o and T_f operate on the GT of the state of figure 3: T_o with a delete operation and T_f with an update operation.

```
To: delete(Salary_Emp,"Emp_num=10",10/1/1,10/12/31)
Tf: update(Salary_Emp,"Emp_num=10",10/1/1,10/9/30,  salary: 1500)
```

When the delete operation is executed, the concurrency controller receives from T_o the *delete message (T_o, Salary_Emp, 10, 10/1/1, 10/9/30)*. [10/1/1, 10/9/30] is the intersection of the validity interval of the delete operation and the validity interval of V4. Then the delete set maintained for T_o becomes as follows:

```
DSo ={(Salary_Emp,100,[10/1/1,10/9/30])}
```

When the update operation is executed, the concurrency controller receives from T_k the *update message (T_f, Salary_Emp, 10, 10/1/1, 10/9/30)*. Then the update set maintained for T_f becomes as follows:

```
USf ={(Salary_Emp,100,[10/1/1,10/9/30])}
```

Suppose that T_o is the first to be validated and T_f has executed its update operation before the T_o validation. So, the delete operation of T_o is effectively done, before the validation of T_f, as shown in figure 4, by the insertion of V5.

When T_f reaches its validation phase, the effect of its update operation must not be reverberated in the DB by data manager, since the data value to update was deleted by T_o. We can detect and declare this conflict by checking the intersection $DS_o \cap US_f$. When re-executed, T_f will not find data value to update; so there is no problem.

Proposition: In order to guarantee the TDB coherence, when the concurrency controller receives a commit message of a transaction T_e, it must check the intersection of DS_e and the update set of each transaction T_k not yet validated. T_k must be aborted if the intersection $\mathbf{DS_e \cap US_k}$ is not empty.

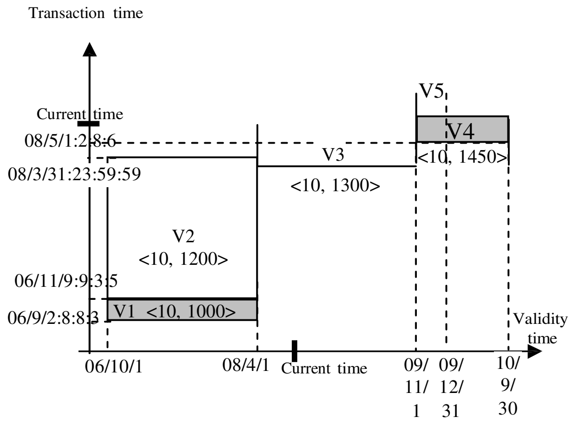

Figure 4. state of the *Salary_Emp* BTR after deletion

Proof: Let T_e be a transaction to be validated and T_k a transaction not yet validated, such that $DS_e \cap US_k \neq \emptyset$. A complete history H_s for T_e and T_k is serial iff all operations of T_e appear before all operations of T_k. A history H is serializable if the committed projection of H, denoted C(H), is equivalent to H_s (C(H) is obtained by omitting from H all operations of transactions that are not yet committed in H).

Suppose that T_k is not aborted. So, for each conflict between $p_e \in T_e$ and $q_k \in T_k$, if $p_e <_H q_k$ then $p_e <_{C(H)} q_k$. In our case, since the delete operation of T_e (D_e) and the update operation of T_k (U_k) are in conflict and T_e is the first transaction to be validated, we must have $D_e <_{C(H)} U_k$.

This means that the update operation of T_k is executed before the delete operation of T_e. So we have $U_k <_{C(H)} D_e$. This implies that this execution is not serializable and data operations of T_e and T_k are not correctly performed.

Remark: If T_k is the first to be validated, the two transactions are not in conflict. Indeed, after validation of T_k, T_e can execute its delete operation since there are valid data during [10/1/1, 10/9/30]. Then, valid data deleted are those updated by T_k. So, this final result is equivalent to the serializable one. Thus, to avoid false conflict detection, the concurrency controller must not check the intersection $US_k \cap DS_e$ at the validation of T_k.

So, we must distinguish between the time period during which data values are deleted and the time period during which data values are updated. Indeed, if we merge the two sets DS and US in the same set, a false conflict is declared. This is equivalent to checking the intersections $DS_e \cap US_k$ and $US_e \cap DS_k$. The other conflict cases $D_e \backslash R_k$, $U_e \backslash R_k$ and $I_e \backslash I_k$, are treated in the same manner as the Delete/update conflict.

Proposition: In order to guarantee the TDB coherence, when the concurrency controller receives a commit message of a transaction T_e, it must check the intersections $\mathbf{DS_e \cap RS_k}$, $\mathbf{DS_e \cap US_k}$, $\mathbf{US_e \cap RS_k}$ and $\mathbf{IS_e \cap IS_k}$ ([18][19]).

4.4. Operation failure detection and proof

An operation fails if it does not find valid data to manipulate. In the case of a read, update or delete operation failure of a transaction T_f, there is a risk of a conflict with a transaction T_o, having the highest priority. Such a conflict occurs when operations of T_o insert new tuples which must be manipulated by T_f, and T_f executes its operations before the T_o validation. Then, the result produced by T_f is erroneous; T_f must read, update or delete data values inserted by T_o.

There is also a risk of an erroneous failure if T_f has to insert a tuple and T_o has to delete previous tuples, in a way that T_f will be able to execute its insert operation. But, T_f executes this operation before the T_o validation. Since this insert operation can not be executed, T_f must be aborted.

Proposition: For each failure of a read, update, delete or insert operation, the concurrency controller must add the granule to the read set of the transaction T_k in order to guarantee the TDB coherence. In addition, it must check, at the validation moment of any transaction T_e, the intersections $\mathbf{DS_e \cap RS_k}$ and $\mathbf{IS_e \cap RS_k}$ for each transaction T_k not yet validated. T_k must be aborted if these intersections are not empty ([18][19]).

Proof: Let T_e be a transaction with an insert operation I_e to be validated. Suppose that there is a transaction T_k not yet validated, containing an update (or a delete) operation U_k of the tuple inserted by I_e. So, we must have $I_e <_{C(H)} U_k$. Suppose now that T_k is not aborted ($IS_e \cap US_k$ is not checked in order to avoid false conflict detection [18], as shown hereafter). This means that the update operation of T_k is executed before the insert operation of T_e. So, it failed since there are no data to update, and we have $U_k <_{C(H)} I_e$. This implies that this execution is not serializable: T_k must be aborted in order to have $I_e <_{C(H)} U_k$. To do that, we must add this granule to RS_k and then check $IS_e \cap RS_k$.

4.5. Structure of the algorithms

We present hereafter the main procedure of the concurrency controller. According to the received message, the concurrency controller carries out the appropriate treatment.

```
CC ( )
Begin
  While there is a message for the CC Do
    Receive_message
    Case message of
      1: Read_M ($T_i$, Granule)
         Notify_READ ($T_i$, Granule)
      2: Insert_M ($T_i$, Granule)
         Notify_INSERT ($T_i$, Granule)
      3: Delete_M (T_i, Granule)
         Notify_DELETE (T_i, Granule)
      4: Update_M (T_i, Granule)
         Notify_UPDATE (T_i, Granule)
      5: Rollback_M (T_i)
         TREAT_ROLLBACK (T_i)
      6: Commit_M (T_i)
         TREAT_COMMIT (T_i)
    EndCase
  EndWhile
End.
```

The procedures Notify_READ, Notify_INSERT, Notify_DELETE and Notify_UPDATE, for a transaction T_i, add a granule to RS_i, IS_i, DS_i and US_i, respectively. Treat_ROLLBACK(T_i) drop all T_i sets. Finally, Treat_COMMIT(T_i) procedure deals with the validation request of T_i. The TREAT_COMMIT procedure is defined as follows:

```
TREAT_COMMIT (T_i)
Begin
    If the strong consistency is needed then
      TREAT_COMMIT_SC (T_i)
    Else
      TREAT_COMMIT_C (T_i)
    EndIf
End.
```

When the CC algorithm is limited to ensure the BD consistency, the TREAT_COMMIT_C procedure is as follows:

```
TREAT_COMMIT_C (T_i)
Begin
    VALIDATION (T_i)
    Change the T_i state from "execution" to "end"
End.
```

When the CC algorithm must ensure the strong consistency, the TREAT_COMMIT_SC procedure is as follows:

```
TREAT_COMMIT_SC (T_i)
Begin
  V := CERTIFICATION (T_i)
  If  (v = 0)  Then
    Put T_i in the list of transactions waiting for certification.
  Else
    VALIDATION (T_i)
    awaking ( )
    Change the T_i state from "execution" to "end"
  EndIf
End.
```

The certification function ensures that T_i is the transaction having the highest priority. In this case, T_i can be validated. In the other cases, T_i must be put on waiting for certification. The validation procedure is defined as follows:

```
VALIDATION (T_i)
Begin
  < /* BEGIN OF THE CRITICAL SECTION */
```

```
    Send "Ok to reverberate the writings on the DB" to the transaction
     manager which will begin the critical section of $T_i$.
    j := stamp of the first transaction which just arrived after $T_i$.
    While  j <> "!"  Do
      Add the $EOT_i$ element to $RS_j$, $US_j$ and $IS_j$.
      j := stamp of the first transaction which arrived just after $T_j$.
    EndWhile
    Send an authorization message, for the transaction manager, to put an
     end to the critical section of $T_i$ when all $T_i$ writings on the DB are
     achieved.
  > /* END OF THE CRITICAL SECTION */
  j := stamp of the first transaction which arrived just after $T_i$.
  While  j <> "!"  Do
    If  (CONFLIT ($T_i$, $T_j$) = 1)  Then
      Delete all elements from $RS_j$, $IS_j$, $DS_j$ and $US_j$
      Give an order to abort and re-execute $T_j$ to the transaction manager.
    EndIf
    j := stamp of the first transaction which arrived just after $T_j$.
  EndWhile
End.
```

Finally, the CONFLICT procedure called to check if there is a conflict between two transactions is defined as follows:

```
CONFLICT ($T_i$, $T_j$): integer
Begin
  r := 0
/* $XS_j(T_i)$ is the objects manipulated by $T_j$ until $EOT_i$ */
  If  $DS_i \cap RS_j(T_i) \neq \emptyset$ or $DS_i \cap US_j(T_i) \neq \emptyset$ or
      $US_i \cap RS_j(T_i) \neq \emptyset$ or $IS_i \cap IS_j(T_i) \neq \emptyset$ or
      $IS_i \cap RS_j(T_i) \neq \emptyset$
  Then  r := 1
  EndIf
  Return (r)
End.
```

4.6. Formal verification using SPIN

The SPIN tool [13] is one of the most powerful model checkers. It is an appropriate tool for analyzing the logical consistency of concurrent systems. The systems analyzed by SPIN are described with the PROMELA language (PROcess MEta LAnguage), which is a specification language for finite state systems. Under PROMELA, a system is represented by a set of parallel processes which communicate via global variables and/or communication channels. PROMELA also allows checking properties which are specified in linear temporal logic (LTL).

SPIN proceeds in two steps. In the first one, "deadlock" or "unreachable code" errors are detected. In the second step, the validity of the system's quality properties is checked through the application of appropriate LTL formulae. For each detected error, SPIN gives the shortest way which leads to this error.

For the specification of our system, we have used different transactions which operate simultaneously. Each transaction is dealt with a process. Thus, our system is composed of the *init* process, the *concurrent processes* and the *concurrency controller* process.

Knowing that each process communicates with the concurrency controller, we then declared a message channel of the type "rendez-vous" (represented by a size equal to zero):

*chan **trans_inst** = [0] of {byte,byte};*

This declaration allows avoiding the risk of the non detection of conflicts which can occur if we use the "buffer" type. Indeed, a given transaction can finish the execution of its statements, but its operation messages still remain in this channel [17].

To declare our system's objects, we have used the *typedef* declaration which allows declaring a user defined type of data. Thus, the following *transaction* type gathers the transaction characteristics.

```
typedef transaction
  {
  byte nom;
  byte order; /* Represents the arrival order of the transaction. */
  byte ordre_validation; /* Indicates a transaction validation order */
  byte state; /*Represents the transaction state*/
  byte rs[size]; /* Represents the read set */
  byte us[size]; /* Represents the update set */
  byte ds[size]; /* Represents the delete set */
  byte is[size]; /* Represents the insert set */
  bit restart;  /* takes the value "true" when the transaction is
   aborted. */
   ...
  };
```

The list of transactions is maintained in the list_tr array:

```
    transaction list_tr[nb_tr];
```

In the first step of validation, SPIN allowed us to detect and to correct a blocking error for the CC algorithm ensuring the strong consistency [17]. This blocking situation is due to the attribution of the value "finished" to the element "*state*", defined in the "*transaction*" type, at the end of the process execution (after the awakening of the concurrent transaction). This problem is avoided by placing the statement of attribution of the value "finished" to the element "*state*" before calling the awakening procedure. After having corrected this error, SPIN display a valid result: zero error (errors: 0) and zero unreached state (unreached in proctype pi) in all the processes ([17], [18], [19]).

In the second step of validation, we define LTL formulae in order to guarantee that safety properties are conserved.

We check that the strong consistency, in the case of a CC algorithm which must ensure this requirement, is guaranteed. We use for this formula the two elements: "*order*" and "*order_validation*" defined in the "*transaction*" type. To make sure that our system guarantees the strong consistency, we must verify, at the end of the execution of any transaction T, that its order is equal to its validation order. We define the following property, called p:

#define p
(list_tr[0].order == list_tr[0].order_validation)

The LTL Formula which we applied is as follows: "<>[]p" which means that there is at least a state from which we will have the property "p" true forever. No error is detected in this checking phase when applying the formula <>[]p.

In the next LTL formulae, we check that in case of a conflict, the transaction having the lowest priority was aborted. Our formulae are then based on the values which can be taken by data item. We suppose that each transaction, when updating a data value, gives it a specific value: data item value updated by T_1 takes the value *tr1* (element "salary"). At the end of the execution, the final value of the element "salary" must be equal to that assigned by the transaction having the lowest priority. If it is not the case, it means that there is a non solved conflict between two transactions on this granule (the transaction having the lowest priority was not aborted).

The LTL formula when the strong consistency is required:

```
[]((<>(a&&b)-><>c)&&(<>(!a&&d)-><>e))
```

a, b, c, d and e properties are defined as follows:

```
#define a (list_tr[0].order<list_tr[1].order)
#define b (list_tr[0].state==finished)
#define c (valid_version[0].salary==tr2)
#define d (list_tr[1].state==finished)
#define e (valid_version[0].salary==tr1)
```

This LTL formula treats the two possible cases between T_1 and T_2 according to their priority orders.

Case 1: if $T_1 > T_2$ (a = true) and if T_1 is finished (b = true) Æ we must be sure to have:
c = true in a future state (element "salary" = "tr2").

Case 2: if $T_1 < T_2$ (a != true) and if T_2 is finished (d = true) Æ we must be sure to have :
e = true in a future state (element "salary" = "tr1").

The application of this formula gives a valid result ([17][18]).

The LTL formula when only the consistency is required:

```
[](  (<>a -><> b) && (<>!a-><> c)  )
```

a, b, c, d and e properties are defined as follows:

```
#define a  (list_tr[0].order_validation  < list_tr[1].order_validation)
#define b  (valid_version[0].salary==tr2)
#define c  (valid_version[0].salary==tr1)
```

This LTL formula treats the two possible cases between T_1 and T_2 according to their priority orders.

Case 1: if T_1 is the first validated (a = true) Æ we must be sure to have:
b = true in a future state (element "salary" = "tr2").

Case 2: if T_1 is not the first validated (a != true) Æ we must be sure to have :
c = true in a future state (element "salary" = "tr1").

The application of this formula gives a valid result [19].

5. BASIC ELEMENTS OF THE PERFORMANCE EVALUATION

For the performance study, we have decided to examine two CC algorithms, a pessimistic one and an optimistic one, vis-à-vis of each one of our two optimistic algorithms: OCCA/TDB and OCCA_SC/TDB.

The decision to take a pessimistic algorithm is based on the fact that the CC methods are classified in two main categories: pessimistic methods and optimistic ones. These two categories differ in two aspects: the time when they detect conflicts and the way that they resolve conflicts (the conflict resolution policy). So, it would be interesting to carry out this study between these two categories.

In other respects, we have decided to consider an optimistic algorithm in order to compare our optimistic algorithm vis-à-vis of another of the same category.

5.1. The algorithms ensuring consistency

To evaluate our optimistic algorithm OCCA/TDB, we have chosen to compare its performance vis-à-vis of the 2PL algorithm, on the one hand, and the Broadcast Optimistic Method (BOM) [21], on the other hand.

The choice of the 2PL pessimistic algorithm rather than the algorithm proposed in [7] is based on the fact that applying the algorithm of [7] leads to revise the transaction atomicity. In other respects, the 2PL pessimistic algorithm is the most popular one, used in many DBMS.

The choice of the BOM algorithm is based on the fact that it validates each finished transaction vis-à-vis of each concurrent transaction not yet validated. This strategy allows detecting conflicts as soon as possible.

5.2. The algorithms ensuring strong consistency

For this study, we have chosen to examine vis-à-vis of the OCCA_SC/TDB, the pessimistic algorithm called 2PL-MO [10] and the optimistic one called BOM_SC. Since the BOM algorithm [21] ensures only the consistency, we have added the stamping of transactions by their arrival moments and a certification phase to ensure that the validation order between transactions is in accordance with their arrival order. Thus, the BOM algorithm now ensures the strong consistency and it is called BOM_SC.

In other respects, to our knowledge, there is no algorithm proposed to study the strong consistency problem for TDB according to the optimistic approach. Thus, we have decided to compare our optimistic algorithm vis-à-vis of the BOM_SC algorithm that we have enhanced to ensure the strong consistency. In addition, let us recall that we have started from the validation strategy of broadcast optimistic method with critical section. Then, we have integrated the "EOT marker" technique. Thus, the main difference between our algorithm and the BOM_SC algorithm, in the case of TTR, is the use of this technique. So, evaluating our algorithm vis-à-vis of the BOM_SC algorithm allows us to check the importance of the "EOT marker" technique integration.

5.3. Model Parameters

The following table summarizes the parameters and the values used in our experiments.

Arrival rate	20trans/sec to 50trans/sec
Database size	800 items
Database location	In the memory .. On disc
Transaction size	5 to 6 operations
Rsize	3 to 80
Wsize	1 to 60
R/W ratio	0.2 to 0.5

The database size determines the number of data items in the database. Note that the database can be located not only on disc, but also in the memory. Indeed, the technology of the DB in memory is a technology in continuous growth to be able to reach high performances.

The number of operations defined in a transaction is given by the transaction size parameter; there are two types of transactions. For the first one, the R/W ratio is 0.5, which means that each write operation is being preceded by a corresponding read operation on the same data item. For the second type, each transaction consists of four read and one write operations, i.e. the R/W ratio is

0.2. In addition, we denote by Rsize and Wsize the number of data items that a transaction reads and writes, respectively.

We present, hereafter, the various scenarios used for the comparative evaluation of the performance of our algorithms. We describe the parameter settings used in our model for the two cases: when database is located in the memory and when database is located on the disc. The performance metric used is the time execution of transactions. Note that each simulation result is determined after a series of trial runs until the obtained results are stable.

Also note that the number of scenarios and the MPL (MultiProgramming Level) maximum value are not *à priori* now. Our idea is to start from the first scenario and then we increase the number of data items manipulated in order to increase the conflict degree. When the result of the performance evaluation becomes clear, we stop the definition of new scenario.

We use the same strategy to define the MPL maximum value. We start from a minimum value of MPL into a scenario and then we increase the number of transactions launched simultaneously. Then, we stop the incrementing of the MPL when the result of the performance evaluation becomes clear.

Database on disk. In this experiment (table1), we simultaneously launched transactions of the first type, i.e. the R/W ratio parameter is equal to 0.5. In addition, the arrival rate is 50trans/sec, which corresponds to an inter arrival time equal to 2ms. So, only Rsize and Wsize parameters are varied. We stop the experiment with three scenarios, knowing that the conflict degree is increased from scenarios1 to scenarios3.

Database in the memory. In this experiment (table2), we have varied the arrival rate and the number of data items that a transaction reads and writes. We stop the experiment with four scenarios, knowing that the conflict degree is increased from scenarios1 to scenarios4. We apply two types of transactions. A transaction of the first type reads and writes 15 data items, whereas a transaction of the second type reads 20 data items and writes 5 data items.

Table1: Scenarios when DB on disc

	Arrival rate	Rsize	Wsize
Scenarios1	50/sec	3	3
Scenarios2	50/sec	6	6
Scenarios3	50/sec	12	12

Table2: Scenarios when DB in the memory

	Arrival rate	Rsize	Wsize
Scenarios1	20/sec	15 to 20	5 to 15
Scenarios2	50/sec	15 to 20	5 to 15
Scenarios3	50/sec	30 to 40	10 to 30
Scenarios4	50/sec	60 to 80	20 to 60

6. Simulation Results and Discussions

We present in this section just some examples of the comparative tables. In all ways, the other tables confirm the results announced in each case.

6.1. Evaluation result when the DB is in the memory

We have made the performance evaluation of our algorithms from scenarios1 to scenarios4. We present, hereafter, the experiment results of OCCA_SC/TDB in the case of VTR and BTR according to scenarios4. We have the same results in the case of TTR.

Evaluation of OCCA_SC/TDB: case of VTR and BTR

As shown in figures 4, the performances of all compared algorithms are identical when the number of transactions in the system is small. But when the multiprogramming level in the system increases, the superiority of our optimistic algorithm OCCA_SC/TDB becomes evident.

The reason that our optimistic algorithms outperform the pessimistic ones can be attributed to the fact that an optimistic algorithm uses the transaction abortion technique in order to avoid inconsistency, whereas a pessimistic algorithm uses locking techniques. Indeed, the locking technique is more expensive than the transaction abortion technique when database is located in the memory. Thus, when the multiprogramming level increases, the conflict degree also increases and the blocking time becomes larger and larger. So, as shown in figure 4, which corresponds to scenarios4, the time execution when applying the 2PL-MO algorithm reaches the value of 28 000, whereas the time execution when applying the OCCA_SC/TDB algorithm does not exceed the value of 600.
Compared to the BOM_SC algorithm, the superiority of our algorithm is considerably lower than the pessimistic algorithms. Indeed, OCCA_SC/TDB and BOM_SC use the same technique to avoid inconsistency.

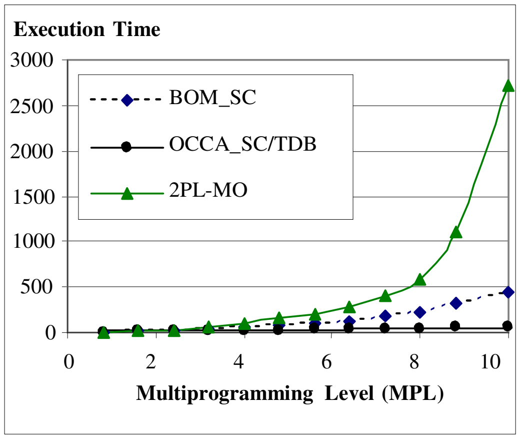

Figure 4. Execution time according to scenarios4

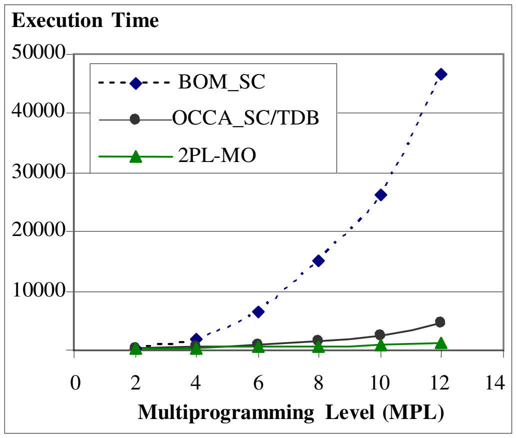

Figure 5. Execution time according to scenarios3

6.2. Evaluation result when the DB is on disk

For this evaluation, we have made the performance evaluation of our optimistic algorithms from scenarios1 to scenarios3. We present, hereafter, the experiment results of OCCA_SC/TDB and OCCA/TDB for two cases: the case of TTR and the case of VTR and BTR.

Evaluation of OCCA_SC/TDB: case of TTR

Figure 5 shows that the performances of all algorithms are also identical when the number of transactions in the system is small. But, when the multiprogramming level in the system becomes higher than 4, applying the 2PL-MO or the OCCA_SC/TDB algorithm is better in time execution than applying the BOM_SC algorithm. The superiority of the 2PL-MO algorithm becomes evident with regards to the OCCA_SC/TDB algorithm when the multiprogramming level in the system becomes higher than 10. Also note that the difference in the execution time between these two later algorithms is clearly smaller than the difference vis-à-vis of the BOM_SC algorithm [20].

The reason that 2PL-MO outperforms OCCA_SC/TDB can be attributed to the fact that the transaction abortion technique is more expensive than the locking technique when database is located on disc. So, when the multiprogramming level increases, the conflict degree also increases and the execution time of transactions becomes larger and larger. In addition, ensuring the strong consistency according to the optimistic approach, leads to the fact that any transaction can finish its operations and then remains on standby until it becomes the transaction having the highest priority. After that, it can be aborted. This leads to increasing the time execution.

Compared to the BOM_SC, the superiority of the OCCA_SC/TDB algorithm is considerably more evident than the case when database is located in the memory.

Evaluation of OCCA/TDB: case of TTR
We show in figure 6 that when the multiprogramming level in the system becomes higher than 6, applying the 2PL or the OCCA/TDB algorithm is better in time execution than applying the BOM algorithm. But contrary to the experimental result for algorithms ensuring the strong consistency, we have now the OCCA/TDB algorithm which outperforms the 2PL algorithm. This means that ensuring the strong consistency according to the optimistic approach is more expensive than ensuring it according to the pessimistic approach. So, as shown in figure 6, which corresponds to scenarios3, the time execution when applying the 2PL algorithm reaches the value of 1 500, whereas the time execution when applying the OCCA/TDB algorithm does not exceed the value of 1 000.

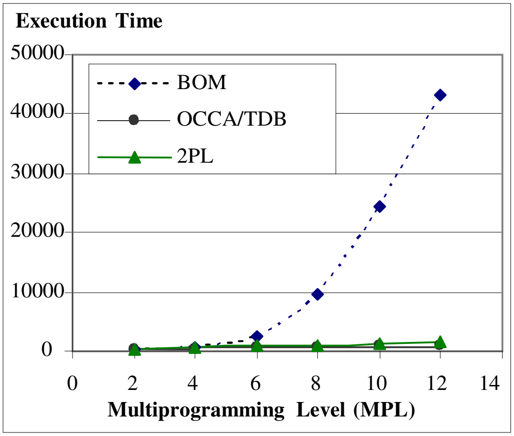

Figure 6. Execution time according to scenarios3

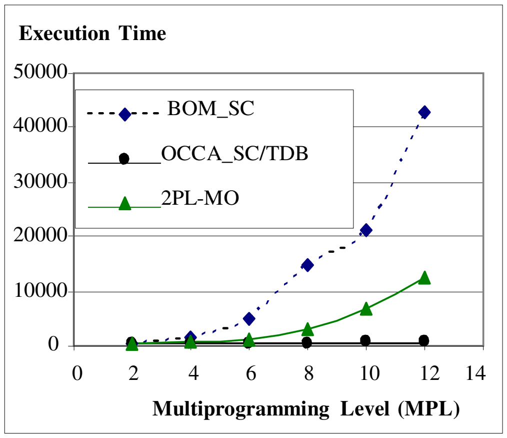

Figure 7. Execution time according to scenarios3

Evaluation of OCCA_SC/TDB: case of VTR and BTR
In figure 7, which corresponds to scenarios3, we always have a high execution time when applying the BOM_SC algorithm. But contrary to the case of TTR, and despite the high cost of the transaction abortion technique and the maintaining of the strong consistency according to the optimistic approach, OCCA_SC/TDB now outperforms the 2PL-MO algorithm. Indeed, this is due to the consideration of the new granule which allows reducing the conflict degree and then the abortion degree. So, as shown in figure 7, the difference between OCCA_SC/TDB and 2PL-MO becomes more and more evident. The time execution of OCCA_SC/TDB is almost sixteen times smaller. The experiment results of OCCA/TDB in the case of VTR and BTR, show that we have the same result as the OCCA_SC/TDB algorithm.

7. Conclusion

We have proposed in this paper new optimistic concurrency control algorithms for temporal databases which can ensure the consistency or the strong consistency. Our propositions are based on the broadcast strategy and integrate EOT marker technique to detect conflicts as soon as possible, and to reduce, to the maximum, the period during which the resources are locked in the validation phase, respectively. In addition, in the case of temporal relation supporting valid time, we have proposed a new type of granule defined as a temporal interval during which some tuples of a GT are valid, to avoid false conflict detection.

Through the study of the various conflict cases and the failed operations, we have defined rigorous CC algorithms, which detect all conflict situations, without false ones. These algorithms are formally verified using the serialization theory and the SPIN model checker.

After checking that our algorithms operate correctly, we have made a performance evaluation to check if our proposition ameliorates the performances, vis-à-vis of other concurrency control algorithms based on optimistic and pessimistic approaches.

Compared to the optimistic algorithms, the obtained result shows that we have a better result when applying our algorithms.

Compared to the pessimistic algorithms, our algorithms have not better results only when the strong consistency is needed for a transaction time relation maintained on disc. But for the other cases, our algorithms outperform the pessimistic ones, especially for relations supporting valid time. Indeed, this is due to the consideration of the new granule which allows reducing the conflict degree and then the abortion degree.

8. References

[1] L. Amanton, B. Sadeg, and J. Haubert, (2003) "Tradding Precision for Timeliness in Distributed Real-Time Databases", *Proc. of 5th Conference on Enterprise Information Systems*, Angers, France, Angers, France, Vol. 1, pp. 558-561.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, (1987) "Concurrency control and recovery in BDS", *ADDISON-WESLEY Edition*.

[3] J. Berstel, S. C. Reghizzi, G. Roussel, and P. S. Pietro, (2005) "A Scalable Formal Method for Design and Automatic Checking of User Interfaces", *International ACM Journal of Transactions on Software Engineering and Methodology*, Vol. 14(2), pp. 124-167.

[4] R. Bouaziz, M. Moalla, and C. Rolland, (1992) "Approche globale pour la gestion de l'historisation dans les bases de données temporelles", *Proc. of the Conference on Informatique des organisations et systèmes d'information et de décision*, Clermont-Ferrand, France, pp. 167-185,.

[5] R. Bouaziz, and A. Makni, (2005) "ACCO_CF/RTT: Un algorithme de contrôle de concurrence optimiste pour les relations temporelles de transaction", *International Journal of Information Sciences for Decision Making*, n°19, paper n°236.

[6] E. Brinksma, A. Mader, and A. Fehnker, (2002) "Verification and Optimization of a PLC Control Schedule", *International Journal on Software Tools for Technology Transfer*, Vol. 4(1), pp. 21-33.

[7] C. De Castro, (1998) "On concurrency management in temporal relational databases", *Proc. of the Symposium on SEBD*, pp. 189-202.

[8] S. D. Elloumi, R. Bouaziz, and M. Moalla, (1998) "Contrôle de concurrence multiversion dans les bases de données temporelles", *Proc. of the "Bases de Données Avancées" Conference*, Hammamet, Tunisie, pp. 135-155.

[9] M. Finger, and P. McBrien, (1996) "On the semantic of 'current time' in temporal databases", *Proc. of the Symposium on Databases*, Sao Carlos, Brasil, pp. 324-337.

[10] M. Finger, and P. McBrien, (1997) "Concurrency Control for Perceivedly Instantaneous Transactions in Valid-Time Databases", *Proc. Of the Conference on Temporal Representation and Reasoning*, pp. 112-118,.

[11] G. Gardarin, (1988) "Bases de données : Les systèmes et leurs langages", *EYROLLES Edition*.

[12] J. Harista, M. Carey, and M. Livny, (1992) "Data Access Scheduling in Firm Real-Time Database Systems", *Real-Time Systems Journal*, *4(3)*, 203-241.

[13] G. J. Holzmann, (1997) "The model cheker SPIN", *International Journal of Transactions on Software Engineering*, 23(5), pp. 279-295.

[14] C. S. Jensen, & al., (1998) "The Consensus Glossary of Temporal Database Concepts"—February 1998 Version, O.Etzion, S.Jajodia, S.Sripada, (Eds.), Temporal Databases—Research and Practice, *Lecture Notes in Computer Science*, Vol. 1399, Springer-Verlag, Berlin, pp. 367–405.

[15] C. S. Jensen, and D. B. Lomet, (2001) "Transaction timestamping in (temporal) databases", *Proc. Of the Conference on Very Large Databases*, Roma, Italy, pp. 441-450.

[16] N. Kim, S. Moon and Y. Sohn, (2004) "Secure one snapshot protocol for concurrency control in real-time stock trading systems", *Journal of Systems and Software*, Vol. 73, Issue 3, pp. 441-454.

[17] A. Makni, R. Bouaziz, and F. Gargouri, (2006) "Formal Verification of an Optimistic Concurrency Control Algorithm using SPIN", *Proc. Of the Symposium on Temporal Representation and Reasoning*, Budapest, Hungary, June 15 - 17, pp. 160-167, Published by IEEE Computer Society.

[18] A. Makni, R. Bouaziz, and F. Gargouri, (2006) "A New Optimistic Concurrency Control Algorithm for Valid-Time Relations", *Proc. of the 10th IASTED International Conference on Software Engineering and Applications*, November 13-15, Dallas, TX, USA, pp. 117-126. Published by ACTA Press, Paper n° 514-143, 2006.

[19] A. Makni, R. Bouaziz, et F. Gargouri, (2007) "Formal Verification of a new Optimistic Concurrency Control Algorithm for Temporal Databases", *Proc. of the 16th ISCA International Conference on Software Engineering and Data Engineering*, Las Vegas, Nevada, USA, July 9-11, pp. 235-242. ISBN 978-1-880843-63-5.

[20] A. Makni, R. Bouaziz, et F. Gargouri, (2007) "Performance Evaluation of an Optimistic Concurrency Control Algorithm Ensuring Strong Consistency for Transaction Time Relations", *Proc. of the International Conference on Enterprise Information Systems and Web Technologies*, Orlando, FL, USA, July 9 - 12, pp. 258-265.

[21] D. Menasce, and T. Nakanishi, (1982) "Optimistic vs Pessimistic control mechanism in database management systems", *International Journal of Information Systems*, Vol. 7(1).

[22] C. Park and S. Park, (2003) "The Freeze algorithms for concurrency control in secure real-time database systems", *Data & Knowledge Engineering*, Vol. 45, Issue 1, pp. 101-125.

[23] M. Rahgozar, (1987) "Contrôle de concurrence par gestion des événements", *PhD thesis, Paris VI University – MASI.*

[24] X. Yingyuan, L. Yunsheng and C. Xiangyang, (2006) "An efficient secure real-time concurrency control protocol", *Journal of Natural Sciences*, Vol. 11, Number 6.