Flexible concurrency control using value dates

W. Litwin, H. Tirri

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 845

FLEXIBLE CONCURRENCY CONTROL USING VALUE DATES

Witold LITWIN
Henzi TIRRI

MAI 1988

RR_0845

Flexible Concurrency Control using Value Dates

W. Litwin
INRIA
78153 Le Chesnay,
France

H.Tirri
Department of Computer Science
University of Helsinki,
Teollisuuskatu 23
SF-00510, Helsinki
Finland

ABSTRACT

A new paradigm for the concurrency control is proposed based on the concept of a value date. A data value is sure only after its value date and uncertain otherwise. We show that among the family of value date based concurrency control policies, there exists an application semantics independent method which guarantees that transaction executions are serializable as well as deadlock and livelock free. We also show that without changing the value date paradigm, a scheduler design may take into account specific properties of a class of applications to enhance the efficiency of the concurrent processing within this class. After studying the relationship between value date method and classical methods, we show that locking, timestamp based methods and the time warp mechanism can all be derived from this same concept. Value dates exhibit also an interesting relationship to the concept of commitment as they constitute an implicit commit operation without any special negotiation protocols. All these properties together with the simplicity of the implementation make the paradigm attractive, especially for multidatabase management in a distributed environment.

Contrôle de Concurrence Flexible par Dates de Valeurs

W. Litwin
INRIA
78153 Le Chesnay,
France

H.Tirri
Department of Computer Science
University of Helsinki,
Teollisuuskatu 23
SF-00510, Helsinki
Finland

RESUME

Un nouveau paradigme de contrôle de concurrence est proposé, basé sur le concept de date de valeur. La valeur d'une donnée est sûre seulement après sa date de valeur et incertaine autrement. On montre que dans la famille de polices de contrôle de concurrence basées sur les dates de valeur, il existe une méthode (VDAS) indépendante d'application qui garantit que les exécutions de transactions sont sérialisables ainsi que libres d'interblocage et de reprise permanente, indépendamment de la sémantique de transactions. On montre aussi que sans changer le paradigme, la conception du contrôleur peut prendre en compte des propriétés spécifiques d'une classe d'applications pour améliorer l'efficacité du traitement concurrent. Après avoir étudié la correspondance entre la méthode de dates de valeur et celles classiques, on montre que les méthodes classiques peuvent être dérivées du concept proposé. Les dates de valeur montrent aussi un lien intéressant au concept de committement, car elles constituent un committement implicite sans un protocole de négociation spécial. Toutes ces propriétés conjuguées à la simplicité d'implémentation, rendent le paradigme attractif, en particulier pour des systèmes multibases dans un environnement distribué.

1. INTRODUCTION

After the introduction of the notion of serializability by Eswaran et al [EGL 76] database concurrency control has become a widely studied field. There are currently hundreds of algorithms [BGH 87] for the concurrency control in databases. Most of the known methods are based on locking [EGL 76] or on timestamping [BeG 81]. The former methods are sometimes called pessimistic, the latter optimistic[1] according to the mechanism to enforce consistency, i.e. blocking or restarting. In practice the most widespread of the locking methods is two-phase locking [EGL 76], likely due to its simplicity both conceptually and from the implementation point of view. In addition most methods are usually coupled with commitment protocols, i.e. algorithms that guarantee the atomicity property [Gra 80] of transactions.

A closer analysis of these algorithms show that they are all based on a common principle of transparency, i.e. each user of the database or the application program should feel as being the single user of the system. Thus the system should be totally transparent with respect to concurrency and the application program should not be concerned with conflicts which may occur with other users of the database. The current assumption is that when the transaction is executed in a concurrent environment, it should not require any additional logic with the possible exception of lock, unlock or commit statements. The concurrency control module should handle all the occurring conflicts. At the implementation level a conflict means that at least one of the transactions is forced to wait or alternatively to restart, all in a way invisible to the user.

However, there are various fundamental problems with these principles which makes it difficult or impossible to apply them in practical applications such as a large multidatabase system [LiA 86], particularly in a distributed environment [LiZ 88]. In a multidatabase system (e.g. viable for banking or travel business related applications) there typically exist a large set of interconnected databases. It seems that there is no way to find an acceptable method for concurrency control starting from the concept of locking, due to the nature of the method. As the transactions are varying in length (i.e. from seconds to days) it is totally unacceptable to keep data locked for a long period of time thus blocking the access of many other transactions. Similar observation has already been discussed in [BKK 85] in the context of engineering applications (CAD) and recently in [GaS 87], where the concept of sagas was introduced. In locking additional problems are caused by deadlocks, which are very hard to detect and solve in distributed systems.

The other basic mechanism based on timestamps seems to suffer from similar problems; long transactions are bound to be restarted and the overall effect in the system may well be that very few

[1] There is some confusion in the literature about the term *optimistic*; it is sometimes associated with certification methods such as [KuR 81], sometimes the scope of the term is extended to cover any method that uses restarts to enforce correct executions. Here we adopt the latter view.

transactions will ever be finished as a result of continuous restarting. An interesting (and a very little studied) additional feature in a multidatabase environment is the fact that data might be sometimes unavailable not only because of failures but due to natural causes, as for example a database may only be open during business hours. As far as we know this unavailability aspect of data has not been addressed in any studies of transaction modelling or database concurrency control.

Due to all the reasons discussed above a new paradigm is needed. In the following we will discuss such a concept, its consequences and implementation issues. Our underlying fundamental assumption is that the total transparency requirement has to be relaxed in the sense that the transactions (i.e. applications issuing transactions) know that they are operating in a concurrent environment. This idea will be realized by requiring transactions to use value dates, a concept well known in the banking business environment. A value date is the date from which on the value is official, i.e. valid. It is established precisely to let a user know that some concurrent operation is or was going on until this date. An important observation is that it may, and usually does differ from the actual time of the operation setting up the value. If the value date follows the operation date, the value is considered unsafe in the meanwhile. For instance, the value date of a check received is usually a day after the actual date of the check processing. The value date may also precede the actual date, meaning that some time was necessary for the concurrent processing that physically has delayed the value definition. This may be useful in some applications, such as interest rate computations.

In this paper we will analyze the concurrency control principles based on value dates. We will introduce a set of rules that a well-formed transaction using value-dates has to satisfy. For these safe transactions there exists a value date policy called VDAS which will guarantee that all the concurrent executions are serializable, deadlock and livelock-free. These properties together with the simplicity and extensibility of the scheme make the value date based approach highly promising both conceptually and pragmatically. In fact we are able to show that classical concepts like locking and timestamp methods can be conceptually derived from our notion of a value date. Through this general relationship the limitations of these classical methods can be more easily understood.

This paper is organized as follows. Section 2 presents the basic concepts and Section 3 discusses a special class of transactions using value dates that we call safe transactions, and the VDAS policy for them. In Section 4 we discuss different ways to relax the assumptions of safe transactions. The corresponding policies for unsafe transactions may be more efficient, while remaining correct for some particular classes of applications. Section 5 presents a comparison of the value date concept to classical methods and Section 6 concludes the discussion.

2. BASIC CONCEPTS

In the literature the notion of a transaction is understood as a collection of consistency preserving database operations which realizes a set of queries expressed in a database manipulation language, such as MSQL [LAZ 88]. We will now proceed by giving rigorous definitions for the concepts needed.

Definition 1. A *transaction T* is an intention leading to manipulation of some data in database(s). The intention is generally formulated in a high-level language, but is represented by a multistatement program with operations of form T:read(D) or T:write(D). In particular, T may consider that some data may sometimes be unavailable for a while or even during the lifetime of the whole transaction. This may happen even if T is alone in the system. Transaction T may then switch to access some other data. If no choice exists, T may issue an T:undo operation which would restore all the values the transaction has modified to the state they were before T started. It may also ask for a restart later on. We assume that for any T:write(D) operation is preceded with T:read(D), so there are no "blind" writes. Notice that although for simplicity only read and write operations are considered, no descriptive power is lost as the delete operation corresponds to T:read(D) followed by T:write(Null), while insertion can be described with a sequence T:read(Null) and T:write(D).

How the conformity of program T to its intention is achieved is outside the scope of this paper. As a practical example of a transaction T considering unavailability one could consider a "rent any car" transaction. This transaction may be required to switch to the Hertz database, if the Avis database is shut down for some reason (and vice versa). Note that this notion of unavailability was not considered in the properties of a classical transaction [Gra 80]. We consider that an execution where unavailability of a data entity D due to a conflict triggered an alternative choice in transaction T is correct, provided that the unavailability of D triggers the same choice for T when it is executed alone in the system.

Definition 2. A *conflict* between concurrent transactions T_1 and T_2 occurs when read-sets and write-sets of these transactions intersect. By solving a conflict we mean that after the execution of both T_1 and T_2 the results (both the database state and information acquired by the transactions)[1] are correct according to the correctness criteria adopted, e.g. serializability.

There exist numerous canonical examples where a conflict can lead to an undesirable result. Examples of such conflicts are the well known conflict in two elementary sequences realizing transactions T_1 and T_2 manipulating the same data entity D, i.e. T_1:read(D) T_1:write(D) and T2:read(D) T2:write(D).

[1] If serializability is chosen as the correctness criteria like in Section 3, we adopt the stricter notion of view serializability [Yan 81] rather than state serializability [Pap 79]

It is obvious that if nothing is assumed about the intentions of the transactions there are cases where the result of the execution

T_1: read(D) T_2:read(D) T_1:write(D) T_2:write(D)

could wipe out the update by T_1 and the result would be incorrect.

Another well known problem caused by the conflict is one variant of the phantom problem [EGL 76]. If T_1 performs a sum of accounts D_i ; i = 1,2,..k and T_2 performs a fund transfer of size 1000$ from D_2 to D_k after T_1:read(D_2), but prior to T_1:read(D_k), then this particular 1000$ would be summed twice, which obviously is incorrect. To be of general interest the method for concurrency control has to universally address solving all types of conflicts.

Our purpose is to propose a scheme for concurrency control with all the above properties (managing unavailability of data, viable in a distributed multiple linked database environment, long transactions) which is both conceptually simple and easy to implement. In the literature the term "locking method" means in fact a family of different locking policies. Similarly we introduce also a family of methods, which we call the *Value date algorithms (VDA)*. This general method is more in tradition of timestamp algorithms [BeG 81] than locking, but as we will see it provides much more flexibility than either of these methods.

Definition 3. *A value date* V_d is an attribute appended (virtually or physically) to a data entity D subject to an operation by a transaction T. Value date has the semantics of declaring the moment in time after which the value is valid. Hence as long as the actual date A_d is such that $A_d < V_d$, then D is safe for the transaction T only. It may remain unchanged until $A_d = V_d$, but it may change as well, being in particular subject to an undo operation. Given A_d one may define V_d as either $V_d > A_d$, $V_d = A_d$, or even sometimes $V_d < A_d$.

The basic idea in transactions using value dates is that each T receives from the scheduler[1] the pair (D, V_d(D). If the data entity D is safe, transaction T is allowed to manipulate it. If D is unsafe transaction T is allowed to decide, whether it still uses D, switches to an alternative D' or asks the scheduler to solve the conflict. The last alternative means that T asks either a restart later on, or possibly a safe value of D which can be provided by undoing the transaction whose value date is in D. This new freedom allows transactions to enhance their performance and flexibility with respect to all classical schemes. However, due to the power of the mechanism one must be careful to avoid errors in case of some conflicts, as well as deadlocks, livelocks etc.

[1]Following the common tradition in concurrency control literature we call the concurrency control module scheduler.

3. SAFE TRANSACTIONS

We will first show that there exists a value date algorithm which works independently of the application semantics. This can be achieved by assuming that all the concurrent transactions respect some basic safety rules with respect to value dates and disregard unsafe values. Later on we will present examples of situations where these rules may be gracefully relaxed for specific classes of applications.

Definition 4. A *transaction T is safe* if:

i) T is of finite execution length.

ii) T starts with getting from the system some value date $V_d(T)$.

iii) T disregards unsafe D's it could get. If needed and possible, T can reissue T:read(D) after $A_d > V_d(D)$, as value of D could have changed in the meantime.

iv) if D is safe then T validates it with $V_d(T)$ at the first read operation T:read(D). Any further operations from T on D do not change $V_d(D)$, i.e. D keeps $V_d(T)$.

v) if T is not terminated by $V_d(T)$, then T is undone before A_d is V_d. This means that all the values T has written become as they were before T was started.

We will deal with the rules for choosing a suitable value date $V_d(T)$ later on. Obviously, this date should usually be at least far enough in future to allow T to terminate in a non-concurrent environment. We further assume that no two safe transactions can have the same V_d (this can easily be guaranteed). Thus V_d value is also an identifier for a safe transaction.

Definition 5. A transaction is safely undone if no value it has written has been used by another transaction. Such an undo is called safe as well.

Undoing a transaction can have a "domino effect" in an unsafe environment where the values not yet validated have already been used by other transactions. Therefore unsafe undo's should generally be avoided.

3.1. Value date algorithm for safe transactions (VDAS)

With safe transactions it will be easy to develop a policy that guarantees the widely accepted correctness notion called serializability [EGL 76] for any type of applications.

Definition 6. An execution (also called a schedule in the literature) of a transaction set $T = \{T_1, T_2, ..., T_n\}$ is *serializable* if after finishing the execution all the information used in

transactions in the set T and the final database state is the same as in some serial (one-by-one) execution of the same transaction set.

Notice that according to our model of transaction in the case of unavailable data the equivalence of the execution is tested against the behavior of the corresponding transaction when executed alone in the system. Hence, serializability requires that if the transaction is prepared to handle the unavailability it will do it similarly in both in the concurrent execution and in some serial execution.

We will first note that a set of safe transactions behaves well with respect to undoing.

Proposition 1. If all transactions are safe, then any transaction T may be safely undone anytime during its execution.

Proof. T will validate with $V_d(T)$ any data entity D it writes. No $T' \neq T$ will then use a data entity D stamped by T, until $A_d = V_d(T)$. As undo T may occur at most at $V_d(T)$, any undo of transaction T is safe o

Definition 7. *(Value date algorithm for safe transactions (VDAS))* Assume that some transaction T_1 issues T_1:read(D_1) to data D_1 already validated by $T_2 \neq T_1$. If T_1 behaves in the following way

(i) if $A_d \geq V_d(T2)$ then T_1 obtains D1.
Otherwise, i.e. if $A_d < V_d(T2)$ then
(ii) if $V_d(T_1) > V_d(T_2)$, then T_1 is scheduled after the end of T_2 that continues its execution.
(iii) if $V_d(T_2) > V_d(T_1)$ and D_1 unavailability was not considered by the transaction description, then T_1 or T_2 is undone otherwise
(iv) if D_1 unavailability was covered by the transaction description, then T_1 continues without accessing D_1

then T_1 is said to follow VDAS policy.

Theorem 1. All executions of a transaction set T that follows VDAS are serializable.

Proof. Consider that T_1 and T_2 have started concurrently. If their datasets do not intersect, then there is no conflict and the execution is serializable. If they conflict on a data entity D_1, consider that T_1 access was scheduled to follow T_2, the other case being symmetric. Then there cannot be a data entity D_2 requested further on concurrently by T_1 and for which the conflict resolution would lead T_1 to precede T_2. Indeed, consider T_1 reading the data entity D_1 validated with $Vd(T_2)$. In the case (i) above there is no conflict, as D_1 value is safe and so T_2 has already terminated. The only case when

T_1 may be scheduled to follow T_2 is case (ii). Consider this case and that now T_2 reads the data entity D_2 already validated by T_1. But, then $V_d(T_1) > Vd(T_2)$ and so either T_2 or T_1 is undone, unless the unavailability of D_2 was programmed to transaction T_2. In all the cases, T_1 would not be scheduled to precede T_2 o

Theorem 1 simply means that VDAS guarantees serializability because of the order imposed by value dates on the termination time of safe transactions. It is easy to see that only the transaction with later value date may be scheduled to wait until the other terminates. If the other transaction requests access to data already validated by the first one, then either it can continue in presence of unavailability of this data, or one of the two transactions has to (and also can be) safely undone. Thus the serial order of conflicting operations is always preserved.

It can be easily demonstrated that this simple policy solves the basic conflict types correctly. For example consider the case of our basic conflict, assuming that T_1:read(D) validates D. The operation T_2:read(D) could occur only after T_1:write(D), or one of the read operations would be undone and so the conflict would be solved. Similarly in the variant of the phantom problem T_2 would be unable to execute T_2:read(D2), as $A_d < V_d(D2) = V_d(T_1)$ holds. Depending on the relationship between $V_d(T_2)$ and $V_d(T_1)$, this read would be delayed adequately or either T_2 or T_1 would be undone and restarted later on.

3.2. Deadlock and livelock freedom of VDAS

Unlike the timestamping mechanism VDAS forces transactions sometimes to wait until A_d exceeds the value date V_d of the conflicting data entity (Definition 7 (ii)). However, as opposed to e.g. two-phase locking it can be shown that VDAS does not cause a deadlock.

Proposition 2. Transaction set T following VDAS does not deadlock.

Proof. For any two T_1 and T_2 that run concurrently, either $V_d(T_1) > V_d(T_2)$ or vice versa (Definition 4). Consider the first case, the other one being symmetric. Let us assume that D_1 has $V_d(T_1)$ and $D_2 \neq D_1$ has $V_d(T_2)$. If the request T_1:read(D_2) is issued, it is delayed until $A_d \geq V_d(T_2)$. If during this delay the request T2:read(D_1) is also issued, then T_2 will be undone, unless it has a strategy for D_2 unavailability. In both cases T_1 will terminate and no deadlock occurs. Similar situation would occur for any longer cycle $T_1, T_2,..,T_k, T_1$, with T_1 waiting for T_2 etc., leading to an undo of T_k o

Proposition 3. Consider a safe transaction T_1 following VDAS whose V_d is sufficiently far in its future for T_1 to terminate in the absence of conflicts. There are executions that may successfully terminate any such T_1 in progress in the presence of any number of conflicts.

Proof. By Theorem 1 any T_2 in progress that has validated data T_1 needs can be undone. In such an execution, T_1 neither needs to wait for the successful end of T2 nor needs to be undone if $V_d(T_1) < V_d(T_2)$ o

A normal way to solve a conflict when T_1 requests data with $V_d(T_2)$, while $V_d(T_2) < V_d(T_1)$, is to execute the corresponding request once T2 terminated. These conflicts may nevertheless delay T_1 enough to make it unable to terminate at $V_d(T_1)$. It could happen any number of times when T_1 restarts, leading in the worst case to a livelock, i.e. the situation where no transaction terminates since they are undone and restarted all the time. However, Proposition 3 states that fortunately there are executions avoiding this danger. If needed, for instance after a fixed limit of *n* restarts, any current transaction may be executed until its end. Note that it is not the case in any execution of safe transactions. For instance, it would not be the case in an execution undoing only T in case of T:read(D) with $V_d(D) > V_d(T)$.

3.3. Further advantages of VDAS policy

Serializability, absence of deadlock and of livelock are the most important properties of safe transactions following VDAS. In addition VDAS has many other properties highly useful for practical applications such as multidatabases that were discussed in the Introduction.

The proposed scheme is much simpler to implement than many of the schemes proposed in the past. It is likely that it is even easier to implement than the two-phase locking accompanied with some protocol for the deadlock and livelock resolution.

When a transaction T requests for an unsafe D, it basically receives $V_d(D)$. Hence, as opposed to locking methods it will know how long it should wait and whether it may get a safe value at all before $V_d(T)$, unless the scheduler undoes the other transaction. Transaction T may use this information to enhance its performance. For instance, it may do some processing in the meantime, including access to other "non-blocked" data. It may also avoid to use D at all if the delay is too long or $V_d(D) > V_d(T)$, and some alternative D' is available. Finally, it may also use this information for trying to convince the scheduler to arbitrate in its favor.

3.4. Examples

Example 1. To show the flexibility of our approach, consider the following examples. Let us assume an interactive transaction T whose goal is to rent a car from one of databases: Avis, Hertz, National,... preferably from Avis. In this case transaction T can adopt the following approach :

T

get $V_d(T)$*;*

T:read(Avis), if $A_d > V_d(Avis)$*, then /* the scheduler puts then* $V_d(Avis) = V_d(T)$

start to discuss with the customer;

if the discussion finishes positively at $A_d < V_d(T)$*, then*

T:write(Avis)

else if $V_d(Avis) - A_d < 1$ *min then wait else read (Herz), etc.*

Example 2. Consider a transaction T with the goal to rent a car and a corresponding hotel H giving a reduced rate to the car rental company. Knowing the value dates, T may rent a car, then try to book a room from hotel H. At that moment T may have several choices :

(i) to simply book the room.

(ii) to wait for some time, if $V_d(H) > A_d$.

(iii) to ask for arbitration, if $V_d(H) > V_d(T)$.

In the last case, the scheduler may undo T or the other transaction T', if T' is just a hotel reservation, i.e. without the associated car rental, or if it is the n-th restart of T etc. In particular, T could also start with the hotel booking in the case of temporary unavailability of the car rental data.

3.5. Estimating value dates and other implementation issues

As we have shown above, the value date approach allows various application based solutions to conflicts, enhancing the efficiency of the concurrent processing with respect to the automatic classical approach. Nevertheless it leaves also room for automatic (re)scheduling. In particular, if T_1 finds D with $V_d(D) > A_d$, the difference $V_d(D) - A_d$ is small and $V_d(D) << V_d(T1)$, then the system may by itself decide to put T_1 into the wait state until $A_d \geq V_d(D)$. On the other hand generally the decisions to start a transaction and arbitrations are under the scheduler responsibility. An attractive property of the approach is that the scheduler is now able to know in advance the probable length of a transaction. Therefore there exist possibilities for flexible usage of various priority rules. The scheduler may give the priority to shorter transactions, or may avoid to undo the longest ones when they are close to their end etc.

We have not yet addressed the problem of finding a suitable value date for the transaction at start time. Obviously it should be based on the estimation of the transaction length in normal, i.e. non-concurrent execution. At least for interactive transactions, it will usually be mainly dependent on the semantic of the application and on human response time. The conflict frequency may have an effect that has to be taken into account, but the processing speed factor should be negligible. For short, batch oriented transactions the estimation may in contrast also include the estimation of the current system work load etc.

Value dates may be physically attached to data or they may exist only in dedicated tables of the scheduler. The detailed study of both approaches remains to be done. In particular, one may consider two special value dates, noted Min and Max: Min is assigned to available data found without any V_d, Max is assigned to currently unavailable data without any V_d. As we will show, even with this binary valued scheme concurrency control can be implemented (which is in fact an exclusive locking policy).

4. RELAXING THE SAFETY CONDITIONS

There are two basic ways to relax VDAS assumptions:

(i) transaction T or the scheduler is allowed to change value dates during the transaction,

(ii) transaction T is allowed to use unsafe values.

Both choices may also be combined and are very attractive in practice. Notice that it is always safe to change $V_d(T)$ before the expiration date once it has been assigned, although this may complicate implementation issues. This alternative may be particularly useful when the length of a transaction may vary greatly. However, in general the correctness of relaxed rules will depend on the application context. The transaction programmer may be aware of the effects that may occur and may be the best person to define the correct rules. On the other hand, a systematic study of various cases is needed and remains to be done. For the time being and the purpose of discussion, we will illustrate this extended approach only through examples showing enough of its practical flexibility.

Example 3. A bank may wish to keep the amount of a check cashed unsafe for a day, although to process the check takes only minutes. One solution to model this is to define a safe transaction that lasts a day. Another alternative is to define a short safe transaction, but to terminate it in case of check preacceptance by the extension of the value date to the next day. Furthermore the bank policy may be to add the check amount to the customer account amount only after the value date. But it might be also that the amount of the check should enter the account with the date of the check reception, the delay for verification being the bank's internal affair. A natural solution may be to set back the value date to that date which is the end of the verification delay. In this particular case the value date would then be behind the actual time. It could be useful for instance to virtually undo an

actual negative balance that appeared in the meantime without the customer's fault. The new condition for the corresponding scheduler is in this case to be able to pass to the transaction the existing value date before the transaction marks its own value date.

Example 4. A transaction may know that some of its results are safe no matter how it terminates and may wish to make them available as soon as possible. One technique may be to allow the value date for the corresponding data to be advanced with respect to $V_d(T)$. It may be useful for longer transactions. An example of this type of an application is an airline reservation system, since frequently a customer who finished with the reservation may still spend time on non-critical requests about the type of plane, of meal, etc. Similarly this can also be the situation in engineering databases, where some subdesigns are already stable although the whole design is not.

Example 5. Consider again the transaction T to rent a car and a hotel giving a reduced rate to the car rental company. It may happen that transaction T has rented a car, and tries to book a room from the corresponding hotel H, but $V_d(H) > V_d(T)$. If nevertheless the number of free rooms F is large, then there is a very high chance of booking being successful. In this case transaction T may confirm the booking to the user and issue a compensating transaction only for the hotel booking, to be launched at $A_d \geq V_d(H)$.

5. VALUE DATE METHODS VS. CLASSICAL METHODS

Consider the special case of choosing $V_d(T) = Max$ (currently unavailable) for all transactions. Hence we can construct an exclusive locking policy based on value dates, with a special invoking mechanism that informs a waiting transaction when the data comes available again. Thus algorithms for locking and two-phase locking in particular are special cases of our approach. The deadlock problem with the locking occurs since the deadlocked transactions T_1 and T_2 have in fact the same value date that in addition is ahead of both transactions. Unlike for the safe transactions, the scheduler has therefore a priori no information to decide which transaction should be undone. The value date method provides higher flexibility and efficiency than locking since it carries more information for both transactions and the scheduler. Similarly shared locks are special cases of $V_d(T) = Max$ where the safety rules of the usage of unsafe values are relaxed for some classes of operations such as retrievals.

Timestamping may also be seen as a particular case where for all value dates one has $V_d(T) = A_d$ of the transaction start operation. They are then used to detect conflicts in a similar way than our value dates, but they provide very little flexibility to solve the conflict if one occurs. The value date approach is thus inherently more efficient than the timestamping when conflicts occur . When there are no conflicts, the value date based schedule works obviously with the same efficiency as the

timestamp method. Similar observation holds also for Jefferson's time warp mechanism [JeM 86], although the mechanism itself is much more complicated than our approach. Notice that also the certification methods such as in [KuR 81] can be derived from value date paradigm as methods that allow unsafe transactions.

A very interesting observation is that the value date may also be considered as a commitment point of the safe transaction as a whole and also for each manipulated value. The corresponding relationship is interesting, as value dates provide an implicit commit while the classical notion corresponds to some explicit negotiation protocol. This property seems to make value dates a potentially useful tool for protection against failures (as it was one of its initial intention in banking). As the commit point may be implicit and set up to an arbitrary future date, our paradigm should be especially useful for distributed environments. The method is in particular a solution for those contexts when classical commitment protocols are clearly inapplicable, e.g. when a single update concerns very many databases (20-100) which in addition are geographically distributed (see Example 6). In this case the unreliability of both the data communication links and systems themselves does not in practice allow for an atomic update for all the databases simultaneously, as due to the high probability of failures some of the databases are bound to be unavailable at any particular moment. Value date may then be a unique vehicle for both concurrency control and online or delayed commitment, guaranteeing the overall consistency at a given date.

Example 6. Consider a safe transaction T that performs a multidatabase update U on some set B of distributed databases. The databases are expected to be mutually consistent only by the date $V_d(U)$ associated with U. The choosing of V_d may take into account the number of sites involved, their opening hours, type of the net etc. The date may then be chosen far enough to make reasonably certain that all sites finished the operation or an appropriate compensating action could be done. The update may be safely incorporated into each B_i because of the existence of $V_d(U)$, unlike in the classical approach, where it has to be kept until the commit outside B_i (note the simplification of the corresponding processing). A number of efficient strategies to guarantee the consistency appear then for consideration, depending on the expected reliability of the communication system, the number of databases involved etc.:

(i) An update of B becomes valid by default, i.e. unless a message invalidates it or increases value date V_d.

(ii) Vice versa, an update is undone at V_d unless a message confirms it prior to V_d or increases V_d.

(iii) Each site confirms the reception of the update by a message sent upon the reception (or upon the demand of the source site), like in two phase commit protocol.

etc.

It is likely that the first strategy will be the most popular for multidatabases with large numbers of databases. Notice also that values of V_d's may be chosen to be different from site to site.

6. CONCLUSIONS

Value dates are a new paradigm that is proposed for the concurrency control problems. We have shown that the concept leads to important practical advantages, with the simplicity of the basic scheme competing with that of two-phase locking. The usage of the concept is furthermore highly flexible through the possibility of manipulating value dates and uncertain values by the transaction itself. The corresponding knowledge of the semantics of transactions may therefore be incorporated into a concurrency control scheme. In many practical situations it then allows to enhance the degree of concurrency with respect to the classical application independent methods.

Further research should concern on one hand experiments with value date based schemes in practical environments (e.g. multidatabases), on the other hand more detailed quantitative comparison between VDAS and other basic methods is needed. Furthermore, many useful schemes with relaxed safety rules, but correct for particular classes of applications remain to be specified and analyzed. Any scheme in tradition of the classic methods where one replaces V_d = Max with a finite V_d and provides eventually access to uncertain values could be an interesting starting point.

REFERENCES

[BeG 81] Bernstein, Ph. and N.Goodman, Concurrency Control in Distributed Database Systems, *ACM Computing Surveys* 13:2 (1981),185-221.

[BGH 87] Bernstein, Ph., V.Hadzilacos and N.Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley,1987.

[BKK 85] Banchilhon, F. W.Kim and H.Korth, A model of CAD transactions. Proceedings of the 11th International Conference on Very Large Databases,1985, 25-33.

[EGL 76] Eswaran, K., J. Gray, R. Lorie and T.Traiger, The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19:11 (1976), 624-633.

[GaS 87] Garcia-Molina,H. and K.Salem, Sagas. Proceedings of the ACM Conference on Management of Data,1987, 249-259.

[Gra 80] Gray, J., A transaction model. IBM Research Report RJ2895, IBM research Laboratory, San Jose, 1980.

[JeM 86] Jefferson, D. and A.Motro, The time warp mechanism for database concurrency control. Proceedings of the IEEE International Conference on Data Engineering, 1986, 474-481.

[KuR 81] Kung,H. and J.Robinson, On optimistic methods for concurrency control. *ACM Trans. on Database Systems*, 6:2 (1981),213-226.

[LAZ 88] Litwin,W., A.Abdellatif, A.Zeroual, B.Nicolas and Ph.Vigier, MSQL: A multidatabase language. To appear in *Information Sciences*.

[LiA 86] Litwin,W. and A.Abdellatif, Multidatabase interoperability. *IEEE Computer* 19:12 (1986),10-18.

[LiZ 88] Litwin, W., Zeroual, A. Advances in Multidatabase Systems. European Conf. on Telecommunications EUTECO'88. R.Speth (ed.), 1988,1137-1152.

[Papa 79] Papadimitriou, C., The serializability of concurrent database updates. *Journal of the ACM*, 26:4 (1979), 631-653.

[Yan 81] Yannakakis, M., Issues of correctness in database concurrency control by locking. 13th ACM Symposium on Theory of Computing, 1981, 363-367.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique