PRACTICAL ISSUES ON CONCURRENCY CONTROL IN DATABASE SYSTEMS

M. R. S. Borges
Universidade Federal do Rio de Janeiro
Rio de Janeiro - *Brasil*

I - INTRODUCTION

For the last few years the area of concurrency control in database systems has been the object of a lot of research [ROSE78, BERN81, KOHL81, GARD82]. Until mid 70's it was generally accepted that locking policies were the best way to solve conflicts in a multi-processing environment. However, with the rise of distributed systems, the research for alternative approaches started again. The reason for this was the fact that communication costs play an important role on the overhead of process interaction. Also, because it is important to keep the site autonomy as the maximum possible. Several other alternatives have been proposed and many more will appear as was explained in [BERN82].

This paper does not introduce a new mechanism for concurrency control. Neither does it compare performance of different strategies. While these research works are considered quite important, we thought that would be interesting to throw some light on the practical aspects of the subject. By practical we mean those of benefit to real applications. This paper tries to demonstrate how the practical aspects relate to the research being carried out on the problem.

It is our opinion that if some factors are taken into consideration, it can improve the performance and the usability of concurrency control mechanisms for database software.

We examine three issues of database usage as it relates to:

- the notion of conflict;
- the types of application;
- integrated design.

We show in each of the following sections how these three issues may affect the design decisions of a concurrency control mechanism.

II- THE NOTION OF CONFLICT

Any set of transactions that access concurrently the same object in an incompatible mode (i.e. at least one is an update) is said to conflict {ESWA76}. The usual way to avoid conflicts is to serailize the conflicting access, in the same order in all objects. An attempt to override this rule would cause one of the following effects:

- an inconsistent update;
- a lost update;
- obsolete versions apparent to the user;
- inconsistencies apparent to the user.

The first two effects occur when two update transactions are performed on the same object concurrently. The inconsistent update is the result of overlapping actions of update transactions on objects related by some integrity constraint. The lost update is a result of an update based on obsolete (but consistent) data. In other words inconsistencies are caused by inappropriate Reads within update transactions. The lost update is caused by either inappropriated Read or Write actions within update transactions. The reader is referred to {BORG86} for more details and examples.

The last two effects may occur when processing Read and Update transactions in parallel. They do not constitute a problem for the database itself but for the user accessing it. This is true as long as the result of a read would not be used for further update, i.e. the Read action is part of a read-only transaction. The third effect occurs when the reads are issued on partially updated objects related by some integrity constraint. The fourth is caused by anticipating a read access on an object with some previous update transaction pending on it.

Whichever notion of conflict is followed, a mechanism is said to be correct if it prevents conflicts happening. In the case of update conflicts (first two effects), we agree that there is no sense in relaxing the conditions which cause them. However, we believe that for the majority of applications is does not constitute a problem if the user sees an obsolete database (how obsolete it can be is dependent on the application). We see three main reasons for that.

First, it is impossible for most applications to keep the databse in compass with the real world. it maybe the case that some facts are waiting to be stored in the database; then the version available is an obsolete one. Also there is no guarantee that the version the transaction was based on would be still valid when the user receives it, unless he prevents real world change occurring. It is our opinion that because of the nature of some applications, the effort the concurrency control mecanism makes to avoid obsolete versions for retrieval-only transactions is fruitless.

Second, other components of the system may arbitrarily contribute to this obsolence. The network, for example, does not usually assure that the messages will be received in the order they were transmitted. In that case the order of queries would be changed inside the system and this can make the answers obsolete.

Last but not least, the system can benefit a lot from permiting more paralellism. Simulation studies on protocols which relax this condition show that the performance of the concurrency mechanism can improve substantially in the case of "query predominant" databases [BORG85, BORG86].

These three resons show that obsolete versions are common-place in several applications. The users are naturally aware to this. The concurrency control mechanism should not bother in avoiding them unless the users express they want an "up-to-date" version.

Actually, these arguments are proven not to be completely true for some mechanisms. The best example is the multiversion protocol [BERN83], which use obsolete version in case of a transaction arrives after a later update has been processed in the same object. The serializability theory only guarantees that the execution will be equivalent to some serial one [BERN82]. This means that these protocols may permit a Read transaction to proceed even if a Write action, from a transaction which arrived earlier, is to be performed later on the same object.

However, most of the mechanism do not permit Reads while Writes are in progress, even if a two-phase commit protocol has been used. In the basic time-stamping ordering approach an out-of-order Read can cause the abortion of update transaction [BERN80]. In the optimistic approach, the validation of a Read transaction may cause the invalidation of an Update transaction later [KUNG81]. In a multiversion protocol an Update transaction can be invalidated by a Read action because the protocol does not diferentiate between Reads for update and Reads for retrieval-only transactions [BERN83].

The fourth effect is a more serious one. Relaxing the conditions which produce it would lead to more parallelism, but the result does not seem acceptable for most of applications. The only value of those results may be for statistical or uncompromised queries [BORG85].

It is our opinion that any concurrency control mechanism should provide the user with the option of relaxing the conditions which produce these last two effects. By doing this it can achieve better system performance when these effects do not matter, as estabilished by the application requirements. The concurrency control mechanism should be able to follow different protocols according to the user definition of transaction. The transactions may be divided into two classes; the normal and the

weak class of transactions. Some recent proposals modifying the basic appraches show that research is starting to be oriented on this direction [UNLA83, GARC83].

III - TYPES OF APLICATIONS

It has been the conclusion of several papers which compare concurrency control mechanism that the performance is very dependent, apart from other things, on the type of application [BADA80, LIN82, LIN83, KOHL83, PEIN83]. There is general recognition that any mechanism can outperform another if we select the appropriate aplication. Because of the characteristics of the concurrency problem, it seems unlikely that any future approach can claim that it has the best performance in all situations.

Now, these mechanism are supposed to be implemented in database softwares for general use. How should the user (DBA) react if he knows that the kind of aplication he has is not of the type in which the mechanism produces the best results? Should he have to choose the package according?

We believe the answer to this question is that the software has to allow more flexibility on the concurrency control mechanisms. This can be achieved by providing the software with a selection of concurrency control strategies similar, for example, to the choice it provides for access methods. The concurrency control should be seen as a interchangeable component in which the user (DBA) may have the option of selecting the most appropriate strategy.

Another alternative, perhaps more feasible, is to design a mechanism which can make the decision on the concurrency control protocol dynamically, according to the kind of transactions running. An example is the variable granularity locking protocol [KORT81]. In that approach, the size (granule) of the object referenced by the transaction.

We think that some research effort should be directed to design a more general mechanism which can be sensitive to the kind of transactions running on the system. At the moment it seems clear that those mechanism should incorporate ideas from various approaches in order to achieve great efficiency in all cases.

IV - INTEGRATED DESIGN

Most papers about concurrency control tend to analyse the problem as if it were the central issue in physical design in database systems. All the other parts of the system are seen only as they affect the concurrency mechanism. While this approach is useful for studying the algorithms it may cause several problems when we try to integrate the mechanism with the other components of the system.

The efficiency of the concurrency control is usually measured by the throughput produced from the input load. It has been demonstrated somewhere that this is very dependent on the characteristics of the load. Furthermore, we claim that the performance of a concurrency control mechanism measured by its throughput, does not necessarily mean the best overall performance of the system.

That aspect is being ignored by researchers when comparing different mechanisms. The throughput produced by a concurrency control mechanism is a potential one because it depends on parallelism in other parts of the system (disks, lines, etc). As the performance of a mechanism is very sensitive to parallel execution, we can have the **"do-all-for-nothing effect"** [BORG85]. This occurs when the mechanism spends a lot of time is discovering a non-conflictant parallel execution, yet at the end the actions would be serialized because of other constraints.

In our opinion, the search for better concurrency control protocols must be oriented to one which produces the best overall performance of the system. In saying this, we mean that concurrency control implementation must be sensitive to the particularities of the other components of the database and not viewed as a stand-alone feature.

Another aspect, perhaps not so visible but also very important, is to investigate how the concurrency mechanism affects other components of the system. The best example is the recovery protocol. Some concurrency mechanism are so complex that would turn it almost impossible to recover from a crash without implementing an algorithm as complex as the concurrency mechanism itself. It is assumed that a history ought to be reproducible when we have a crash. In other words a recovery from a crash must not alter the effect of transactions already commited. However, in some cases, in order to reproduce such history the recovery mechanism may have even to reproduce abortions (???). More important, the recovery can be very expensive and cause long delays in the system.

V - CONCLUSIONS

We have presented in this paper some ideas of how to deal with concurrency control in a more realistic way. While it has been important to search for new algorithms we think some effort should now be directed to producing mechanisms for use in real systems and not in hypothetical ones. We believe that at the moment there is a gap between the research and the implementation that has not been properly filled.

There are other aspects to be taken into consideration which were not covered in this paper. One of them is the uniformity of processing. There is little sense, for real applications, in arbitrarily speeding some transactions if that causes unacceptable delays on other transactions. (for example, due to sucessive fails).

All these ideas has been investigated in a Ph.D. project {BORG86} which simulates the performance of the mechanisms in practical situations. We incorporated in the simulation model different kind of transactions which eases the notion of conflict. We are experimenting with a wide range of application types (from query to update predominant system). We modeled all the components of a database at the physical level in order to examine the effects the concurrency mechanism has on other components as well.

REFERENCES

{BADA80} Badal, D.Z. : The analysis of the effects of concurrency control on distributed databases system performmance. 6th VLDB (1980) pp. 376-383..

{BERN80} Bernstein, P. and Goodman, N. : Timestamp-based algorithms for concurrency control in distributed databases, 6th VLDB (1980) pp. 275-284.

{BERN81} Bernstein, P. and Goodman, N. : Concurrency control in distributed databases systems. ACM Computing Surveys V13 N2 (June 1981) pp. 185-221.

{BERN82} Bernstein, P. and Goodman, N. : A sophisticate's introduction to distributed database concurrency control. 8th VLDB (1982) pp. 62-76.

[BERN83] Bernstein, P. and Goodman, N. : Multiversion concurrency control - Theory and Algorithms. ACM TODS V8 N4 (December 1983) pp. 465-483.

[BORG85] Borges, M.R. : Twords a flexible mechanism for concurrency control in database systems. 4th British National Conference on Databases. Keele, UK (1985) pp. 39-60.

[BORG86] Borges, M.R. : A flexible mechanism for concurrency control in systems. Ph.D. Thesis 1986, School of Information Systems, University of East Anglia, UK.

[ESWA76] Eswaran, K.P., Gray, J. Lorie, R.A. and Traiger, I. : The notions of consistency and predicate locks in a database system. Comm. ACM V19 N11 (November 1976) pp. 624-633.

[GARC83] Garcia-Molina, H. : Using semantic knowledge for transaction processing in a distributed database. ACM TODS V8 N2 (June 1983) pp. 186-213.

[GARD82] Gardarin, G. and Melkanoff, M. : Concurrency control principles in distributed and centralized databases. INRIA Rapport de Recherche no. 113 (January 1982).

[KOHL81] Kohler, W.H. : A survey of techniques for synchronization and recovery in decentralized systems. ACM Computing Surveys V13 N2 (June 1981) pp. 149-183.

[KOHL83] Kohler, W.H., Wilner, K.C. and Stankovic J.A. : An experimental comparison of locking policies in a testbed database system. ACM SIGMOD Conference (1983) pp. 108-119.

[KORT81] Korth, H.F. : A deadlock-free, variable gralularity looking protocol. Proc. 5th Berkeley Workshop on Dist. Data Manag. and Computer Networks. (February 1981) pp. 105-121.

[KUNG81] Kung, H.T. and Robinson J.T. : On optimistic methods for concurrency control. ACM TODS V6 N2 (June 1981) pp. 213-226.

[LIN 82] Lin, W.K. and Nolte, J. : Performance of two-phase looking. Proc. 6th Berkeley Workshop on Dist. Data Manag. and Comp. Network. (February 1982) pp. 131-160.

[LIN 83] Lin, W.K. and Nolte, J. : Basic timestamp, multiple version, and two-phase looking. 9th VLDB (1983) pp. 109-119.

{PEIN83} Peinl, P. and Reuter, A. : Empirical comparision of database concurrency control shemes. 9th VLDB (1983) pp. 97-108.

{ROSE78} Rosenkrantz, D.J., Stearns, R.E. and Lewis, P.M. : System level concurrency control for distributed database systems. ACM TODS V3 N2 (June 1978) pp. 178-198.

{UNLA83} Unlad, R., Praedel, U, and Schlageter, G.: Design alternatives for optimistic concurrency control schemes. 2nd ICOD (International Conf. on Databases). (August 1983). pp. 288-297.