Concurrency Control in Admin

Parimala N, Naveen Prakash and N. Bolloju
R & D Group, Computer Maintenance Corporation Ltd, Jeevan Vihar, 3, Parliament Street, New Delhi—110001, India

A practical method for scheduling multiple users is described here. This has been done by defining two resource lists, a retrieval list and an update list, which contain the resources required for retrieving and updating the database, respectively. For each user these lists are built at the time of the definition of a subschema. The resource lists thus prepared are used to schedule the users in a suitable manner.

INTRODUCTION

Concurrency control in database systems has been a subject of research for several years. The problem of concurrency control concerns itself with ensuring that, when multiple users access the database, then each user sees a consistent view of the database. The inconsistencies which may arise include lost updates, dirty reads and unrepeatable reads.[1]

In order to avoid these undesirable situations the underlying principle in most systems is to define a unit of locking.[2,3] The lockable unit can be a field, a record occurrence, a record type, etc. If the unit is fine (for example a field) then there is a higher amount of concurrency, but the overhead of the system managing this locking protocol is higher. If the unit is coarse (for example a file) then the level of concurrency falls, but the management becomes simpler. Thus, there is a trade-off between the degree of concurrency and the overhead and this trade-off is determined by the granularity of the lockable unit.[2] Ries and Stonebraker[4] have shown that a relatively coarse granularity is sufficient to give enough parallelism. Summarizing the results of a simulation study carried out by them, these authors state, 'Under the assumptions mentioned in the description of our model, it appears that a small number of granules is sufficient to allow enough parallelism for efficient machine utilization. Furthermore, a large number of granules, corresponding to locking a page or record, is extremely costly. Any advantages due to additional parallelism are outweighed by this cost'. In fact, these authors go on to conclude that 'a very crude concurrency control scheme seems most desirable'.

In database systems, it is well established that a resource is locked in exclusive mode for updates and in shared mode for read only.[3] We shall use exclusive and shared modes of locking a resource in the sense stated above.

Once the units of locking have been defined, each user locks the resources as dictated by his requirements. A serious problem encountered is that of a deadlock. In principle, one can handle deadlock by either preventing it or by detecting and rolling back from a potential deadlock situation. When deadlock is prevented there is normally a loss in concurrency because some users may not be scheduled even when resources are available.[3]

In Admin, we define a corec type (a record type of CODASYL) as a resource. Whenever a subschema is defined the DDL processor determines the resources that would be required upon invoking the subschema in a user program. The resource requirement is determined only once for the lifetime of a subschema. Whenever a user opens the database the resources of his subschema are locked automatically by the DBCS and are released when he closes the database. This protocol is invisible to the user. The locking of resources and subsequent releasing is done all at once, i.e. as an indivisible operation. Furthermore, a user is allowed to proceed beyond opening the database only if all the resources are available to him. If partial resources are available then the user shall not lock these but wait till all the resources are available. Thus deadlock is prevented. It must be noted that we do not incur major expenses at runtime, as in Ref. 3, in preventing deadlock. This is because the resource requirement is determined at compile time, i.e. at the time of the definition of the subschema. This has been possible because access rights are explicitly specified in our system.[5] Therefore, at the time of subschema definition we are aware of all the permitted DML operations on the data structures that are included.

The layout of the paper is as follows. In the next section we identify the resources needed for operating on the data structures and then describe the manner in which the DDL processor determines the resources of a subschema. The scheduling algorithm along with its merits and demerits is discussed in the final section.

RESOURCES NEEDED

The basic structures, in Admin, are a corec type and a coset type (record type and set type of CODASYL, respectively). Further, a new corec type[6] constructed using already defined corec types, which are referred to as base corec types can be either an evolving corec type or a virtual corec type (see below). A corec type which is neither a new corec type nor a base corec type will be referred to as non-base corec type. The coset types using which a new coset type[6] is defined, shall be referred to as non-new coset types. A new coset type itself can be either a type 1 or a type 2 coset.

As stated above, a new corec type can be either an evolving corec type or a virtual corec type. An evolving corec type reflects all changes made to its bases and all changes made to it are reflected in its base corec types. A virtual corec type, on the other hand, cannot be updated. Further, the occurrences which constitute a virtual corec type are those which can be constructed at open-db time.

CCC-0010-4620/84/0027-0062 $02.50

These occurrences remain constant till such time as the database is closed.

In the rest of the paper, we shall use the terms corec type and corec interchangeably. Further, an instance of a corec type shall be referred to either as an occurrence or as a record of this type. Similarly, we shall use the terms coset type and coset interchangeably. We shall refer to an instance of a coset as its occurrence.

Tables 1–5 give the resources required for operating upon the data structures defined above. We shall first consider the resource requirements for corec manipulation followed by those needed for operations on a coset and lastly the requirement for a field modification.

Corec retrieval

Consider Table 1. To traverse in a non-base or a base corec it is sufficient to have available the corec type itself. On the other hand, to find an occurrence of a new corec type, fields from both the bases have to be picked up. Therefore, the bases as well as the corec itself must be available for retrieving a new corec occurrence.[6] It must be noted that, perhaps, it is possible to translate the operations on a new corec to operations on the base corecs and do away with the new corec as a resource itself. But, in accordance with the rule that every corec type is a resource we have included a new corec in the set of resources. It must be noted that in doing so we have not reduced the level of concurrency available in the sytstem.

Table 1. Corec *R*—find/obtain

	Resources
(1) *R* is a non-base or a base.	*R*.
(2) *R* is a virtual corec or an evolving view corec built over the bases *Base*1 and *Base*2.	*R*, *Base*1, *Base*2.

Corec update

We shall now consider the requirements for updating a corec type. In doing so it must be borne in mind that no update operations are allowed on a virtual corec.

Store. For storing a record the resources required are given in Table 2. For the moment we shall postpone the explanation of the base of a virtual corec and deal with it when we are talking about the base of an evolving corec.

Whenever a record is stored it has to be linked in the appropriate coset occurrences of all cosets in which it is a member with INCLUSION automatic. Therefore, the resources required, aside from the corec itself are owner corecs of all such cosets.

When a record of an evolving corec is being stored then a record of either or both the base corecs may also have to be stored.[6] Consequently, not only the evolving corec but also its two base corecs must be treated as resources to be made available. It is possible, however, that the base corecs themselves be members of cosets with INCLUSION automatic. This requires that the base records just stored be linked in appropriate coset occurrences. Therefore, the owner corec of all such cosets must be made available as resources. It must be noted that an evolving corec itself cannot take part as a member in a coset with INCLUSION automatic.

Table 2. Corec *R*—store

	Resources
(1) *R* is a non-base or a base of a virtual corec.	*R*. Owners of all cosets in which *R* is a member and which have INCLUSION automatic.
(2) *R* is an evolving corec built over *Base*1 and *Base*2.	*R*, *Base*1, *Base*2. Owners of all cosets in which either *Base*1 or *Base*2 is a member and which have INCLUSION automatic.
(3) *R* is one of the base corecs of an evolving corec *R*3. Let *R*2 be the other base corec.	*R*, *R*2, *R*3. Owners of all cosets in which *R* is a member and which have INCLUSION automatic.

The base of an evolving corec has to be treated differently from that of a base of a virtual corec. Whenever a record of the base of an evolving corec is stored then zero, one or more records of the evolving corec have to be created.[6] Therefore, in such a situation the evolving corec as well as both the bases should be available as resources. On the other hand, the occurrences of a virtual corec are those which can be constructed upon opening the database. Therefore, the base of a virtual corec can be treated like a non-base corec.

Delete. As explained above the base of a virtual corec will be treated like a non-base corec. Whenever a record is deleted it has to be removed from all cosets in which it takes part as a tenant.[7] Therefore, the resources required, other than the corec itself, would be the owner of all cosets in which the corec is a member together with the members of all cosets in which the corec under consideration is an owner corec. These corecs are referred to as the other tenants in Table 3.

Table 3. Corec *R*—delete

	Resources
(1) *R* is a non-base or a base of a virtual corec.	*R*. For all cosets in which *R* is a tenant the other tenants.
(2) *R* is an evolving corec built over *Base*1 and *Base*2.	*R*, *Base*1, *Base*2. For all cosets in which one of *R*, *Base*1 and *Base*2 is a tenant, the other tenants.
(3) *R* is one of the base corecs of an evolving corec *R*3 which represents either a 1:1 relationship between *R*′ and *R* or represents a 1:1 or 1:*N* relationship between *R* and *R*′.	*R*, *R*′, *R*3. For all cosets in which either *R* or *R*3 is a tenant the other tenants.
(4) *R* is one of the base corecs of an evolving corec *R*3 and *R*3 represents a 1:*N* relationship between *R*′ and *R*.	*R*, *R*3. For all cosets in which either *R* or *R*3 is a tenant the other tenants.

Deletion of a record of an evolving corec might imply the deletion of either or both of the base records constituting the record just deleted.[6] Therefore, it is necessary that both the base corecs and the other tenant of the cosets in which the base corecs participate be made available as well.

In order to understand items (3) and (4) in Table 3 a note on the implementation of an evolving corec would be worthwhile, as the implementation dictates the resource requirements in these cases. Let r and r' be the occurrences of R and R' respectively which constitute a record $r3$ of an evolving corec $R3$. When $R3$ represents a 1:1 relationship between R and R' then r has a pointer pointing to r' and r' has a similar pointer but pointing to r. When $R3$ captures a $1:N$ relationship between R and R' then only r' has a pointer pointing to r. However, more than one record occurrence of R' may point to r. Now, when either r or r' get deleted then zero, one or more records of $R3$ may get deleted.[6] In order to capture this it may become necessary to zero out the pointer in either or both the base records, as the case may be.

Coset-retrieval and update

Consider Table 4. We shall first look at the resource requirement for purely retrieval operations and later consider update operations. In order to navigate in a non-new coset it is sufficient to have the tenants made available. To find a member record of a type 1 coset $S(O, M)$ the system has to first navigate in the coset $S'(O, R)$ and then find records of M by traversing in the coset $S''(R, M)$. Therefore, the resources required would be O, R and M. In other words, the missing corec and the tenants of the type 1 coset are needed. Similarly, for a type 2 coset the tenants of the coset are required as resources. However, a type 2 coset has a tenant which is an evolving corec. Hence, the base corecs of the evolving corec are also needed to navigate in such a coset.

In considering the update operation of a coset type we have to deal only with a non-new coset. This is because no update operations are allowed on a new coset. A little thought will show that for any of the update operations on a non-new coset the resources needed are just the tenants of this coset.

Table 4. Coset $S(O, M)$—find/obtain

	Resources
(1) S is a non-new coset.	O, M.
(2) S is a type 1 coset built over $S'(O, R)$ and $S''(R, M)$.	O, M, R.
(3) S is a type 2 coset with the tenant which is an evolving corec built over *Base*1 and *Base*2.	O, M, *Base*1, *Base*2.

Coset $S(O, M)$—remove, transfer, insert

	Resources
(1) S is a non-new coset.	O, M.

Table 5. Field F of the corec R—modify

	Resources
(1) R is a non-base corec or a base of a virtual corec.	R. Owners of all cosets in which F is defined as a sorting field.
(2) R is one of the base corecs of an evolving corec $R3$. Let the other base be R'. Let $F3$ be the field in $R3$ corresponding to F in R.	R. Owners of all cosets in which F is defined as a sorting field. If F is a join field then R' and the other tenants of all those cosets in which $R3$ is a tenant. Owners of all cosets in which $F3$ is defined as a sorting field. The second base if $F3$ is a sorting field of some coset.
(3) R is an evolving view. Let the field in the base corec corresponding to F be F' and this base corec be R'.	R, R'. Owners of all cosets in which F is a sorting field. The second base if F is a sorting field of some coset. Owners of all cosets in which F' is a sorting field.

Field modification

Consider Table 5. As before a field belonging to a base of a virtual corec will be treated like that belonging to a non-base corec. To modify a field of a corec it is obvious that the corec type must be available. Further, if the corec takes part as a member in some coset type, the members of which are sorted using the field being modified, then this action may involve a reordering of the member records. This implies that the owner of this coset type must be available. In order to understand items (2) and (3) in Table 5 the following points must be noted:

(a) Modification of a join field of one of the base corecs of an evolving corec might imply the deletion as well as addition of zero, one or more records of the evolving corec.
(b) If a field of a base is modified then the value in the corresponding field of the evolving corec is also considered to be modified and vice versa.
(c) According to the implementation rules, for any coset in which an evolving corec is a member all coset pointers are available with the second base.

The definition of resource requirement is, perhaps, not very simple when either a corec type is updated or when a field is modified. This is mainly due to the implementation strategy adopted for an evolving corec.

Constructing resource lists

Corresponding to each DML command there is an access right which grants permission to perform that particular operation. For example, in order to store an occurrence of a corec type, the STORE right must be granted for this corec type in the subschema. This correspondence allows us to classify the access rights into two classes—the retrieval set and the update set. We give in Table 6 the retrieval and update sets of access rights for the data structures of Admin.

For purposes of scheduling concurrent users we define two lists called the retrieval list (*Rlist*) and the update list (*Ulist*). *Rlist* specifies the list of resources required for

Table 6

Structure	Retrieval set	Update set
Field	—	modify
Corec	find, obtain	store, delete
Coset	find, obtain	insert, remove, transfer

Table 7

Option	Resources	Mode
retrieval	*Rlist*	shared
debug	*Ulist* and *Rlist*	shared
update	*Ulist*	exclusive
	(*Rlist*—(*Ulist* ∩ *Rlist*))	shared

purely retrieval operations and *Ulist* gives that required for update operations. These lists are prepared by the processor at the time it is processing a schema/subschema definition as follows.

When the processor comes across an access right on a data structure it determines whether the right belongs to the retrieval set or to the update set. In the former case, the resource(s) required for operating upon this structure is (are) entered in *Rlist*. A similar action is taken when the access right belongs to the update set except that in this case the resources are entered in *Ulist*.

The total resources available when a schema is defined is the set of corec types of the schema. Since all access rights are granted on each of the corecs and cosets comprising the schema[5] the *Rlist* and the *Ulist* of the schema will contain all the resources. When a subschema is defined, however, access rights are explicitly granted to the data structures. The *Rlist* and *Ulist* of this subschema are constructed according to the rules given above. It must be noted that a virtual corec defined in a subschema is treated like any other corec type. However, this resource is available only to this subschema and to its subschemas at level 2,[8] if any.

SCHEDULING

In this section we shall describe how the resource lists prepared at the time of a subschema definition shall be used in scheduling users.

First, let us consider the manner in which the database can be opened in Admin. There are three ways in which a program can open the database, namely (1) retrieval (b) debug and (c) update.[7] When the database is opened for retrieval then the operations that can be performed are determined by the retrieval set of access rights granted in this subschema. For example, the access rights for a corec type *R* could be FIND and STORE. When this subschema is opened for retrieval, only find of *R* should be allowed by the system even though there is an access right to store *R*.

When the database is opened for update then the operations corresponding to the update set as well as those corresponding to the retrieval set of access rights can be performed. The debug option is the same as the update option except that the modifications made by the program are not reflected in the database. These are made in a local copy for the user and cease to exist when the database is closed.

Now, let a user invoke a subschema *S* and open the database using one of the options given above. Upon encountering this statement the DBCS locks resources according to Table 7.

When the database is opened for debug it is sufficient to acquire resources in a shared mode as the updates are not reflected in the database. However, when the update option is chosen, resources belonging to *Ulist* must be locked in exclusive mode and resources in *Rlist* which do not figure in *Ulist* must be locked in a share mode. This is because the latter resource can never be updated by the user.

All the locks acquired are released by the DBCS when it encounters the close-db statement. Also, whenever a program terminates abnormally all the resources held by the program are released by the system.

An analysis

We shall list below the merits and demerits of the proposed concurrency control mechanism.

(a) For every subschema the resource requirement is analysed at the time of its definition. If two users are disjoint with respect to the resources needed by them then they can open the database concurrently. Let us now examine this general rule keeping in mind that there are two resource lists, *Rlist* and *Ulist*, associated with each subschema and every subschema can be opened for either retrieval, debug or update.

Let *R*1, *R*2, *R*3 and *R*4 be resources. Further, let *S*1, *S*2 and *S*3 be three subschemas having the resource lists as given in Table 8:

Let *P*1, *P*2 and P3 be three users associated with *S*1, *S*2 and *S*3, respectively. Table 9 gives the modes in which *P*2 and *P*3 can open the database once *P*1 has already opened the database.

Since *S*1 and *S*3 are disjoint, *P*1 and *P*3 can run simultaneously. This, however, gives rise to the following problem.

Let *P*1 open the database for update thereby locking *R*1 and *R*2 in exclusive mode. Now, let *P*2 make a request to open the database for update as well. *P*2 will have to wait in the queue as *R*2 is locked by *P*1. Now, let *P*3 make a request to open the database for update. It must be noted that the resources required by *P*3, namely *R*3

Table 8

Subschema	*Rlist*	*Ulist*
*S*1	*R*1, *R*2	*R*1, *R*2
*S*2	*R*2, *R*3	*R*2, *R*3
*S*3	*R*3	*R*3, *R*4

Table 9

*P*1 opens for	*P*2 can open for	*P*3 can open for
retrieval	retrieval, debug	retrieval, debug, update
debug	retrieval, debug	retrieval, debug, update
update	—	retrieval, debug, update

and $R4$, are not locked but $P3$ is behind $P2$ in the queue. In such a situation $P3$ is processed thereby giving a higher amount of concurrency. This, however, might lead to $P2$ being starved.

(b) A program proceeds beyond open-db only if all the resources are available and shall enter into a wait state till the resources are freed. Further, acquiring the locks as well as subsequently releasing them is performed as a unit action. Therefore, deadlock can never occur in our system.

(c) The locking mechanism is totally hidden from the user. The runtime support does the locking and unlocking of resources thereby requiring that the user know nothing about concurrency to run his program.

(d) There are no lost updates, dirty reads or unrepeatable reads. These properties arise by virtue of the fact that the granularity of locking is a corec type (not an occurrence) and locks once acquired are released only when either the database is closed or the program aborts.

(e) It is ideal if a system locks only those resources that are operated upon by the user. However, in Admin, the user intention is not considered and all resources that he can possibly operate upon are locked at open-db time. Perhaps, there is a trade-off between (b) and (e).

REFERENCES

1. James Gray, Notes on data base operating systems. *IBM Research Report RJ 2188*, San Jose (1977).
2. J. M. Gray *et al.*, Granularity of locks and degrees of consistency in a shared database. *IBM Research Report RJ 1654*, San Jose (1975).
3. D. B. Lomet, A practical deadlock avoidance algorithm for database systems. *Proc. ACM SIGMOD Conference,* Toronto, Canada (1977).
4. D. R. Ries and M. Stonebraker, Effects of locking granularity in database management systems. *ACM TODS* **2** (3), 233–246 (1977).
5. Naveen Prakash *et al.*, Access control in a network DBMS, *CMC Technical Report 6* (under communication) (1982).
6. Parimala N *et al.*, New corecs and new cosets in Admin, *CMC Technical Report 5* (under communication) (1982).
7. N. Bolloju *et al.*, The DML of Admin. *CMC Technical Report 9* (1982).
8. Naveen Prakash, Parimala N and N. Bolloju, Data definition facilities in Admin, *The Computer Journal* **26**, 329–335 (1983).

Received January 1983