An integrated concurrency control in object-oriented databases

Woochun Jun and Le Gruenwald

School of Computer science
University of Oklahoma
Norman, OK 73019
E-mail : Gruenwal@mailhost.ecn.uoknor.edu

Abstract

In this paper, we present a concurrency control scheme to increase concurrency among methods in Object-Oriented Databases. We are concerned with all types of access to an object : instance access and class definition access. For instance access, our work has the following characteristics. First, construction of commutativity relation among methods can be automated. Second, it provides more concurrency than read and write access modes on methods. Third, deadlocks due to lock escalation can be reduced. Finally, concurrency is increased further with the use of run-time information. For class definition access, we allow class definition access methods to run concurrently by taking fine granularity. We also allow more parallelism between class definition access methods and instance access methods.

1. Introduction

In object-oriented database systems (OODBS), a database is a collection of classes and instances where classes and instances are called *objects*. Users can access objects by invoking methods in OODBS. To make sure atomicity of user interactions, the traditional transaction model can be used in OODBS. That is, users can access OODB by executing transactions, each of which is defined as a partially ordered set of method invocations on class or instance objects (Agrawal, 1992).

Concurrency control involves synchronization of multiple access to the database, so that the consistency of the database is maintained (Bernstein, 1987). Like in conventional databases, concurrency control in OODBS also requires logical consistency of data and transactions. Concurrency control requires an application-dependent correctness criterion to maintain database consistency while transactions are running concurrently on the same object. Serializability is a widely used correctness criterion. Transactions are serializable if the interleaved execution of their operations produces the same output and has the same effects on the database as some serial execution of the same transactions (Bern81, Bern87).

In OODBS, one of main concerns is to increase concurrency among methods so that more transactions can run in parallel. Usually, transactions are long in typical OODB applications. Thus, aborting a long transaction due to conflicts wastes system resources. Also, holding resources by a long transaction may delay other transactions. Commutativity is a widely used criterion to determine whether a method can run concurrently with those in progress on the same object. Two methods commute if their execution orders do not affect the results of the methods. Two methods conflict each other if they do not commute.

In general, there are two types of access to an OODB : instance access (instance read, instance write) and class definition access (class definition read, class definition update) (Cart, 1990). An instance access consists of consultations and modifications of attribute values in an instance or a set of instances. A class definition access consists of consulting class definition, adding/deleting an attribute or a method, changing the implementation code of a method or changing the inheritance relationships between classes, etc.

In this paper, we present a locking-based concurrency control scheme to increase concurrency among methods. For instance access methods, our scheme has several important characteristics. First, it does not put the burden of determining commutativity for methods on application programmers. Second, it provides more concurrency than read and write access modes on methods. Third, we reduce deadlocks due to lock escalation, which is a main source of

deadlocks (Malta, 1993). Finally, it takes run-time information into consideration to improve concurrency. For class definition access methods, we allow them to run concurrently by taking fine locking granularity. Also, we allow more parallelism between class definition access methods and instance access methods.

The paper is organized as follows. In the next section, we review related studies and discuss their advantages and disadvantages. In section 3, we propose a scheme to increase concurrency among methods. The paper concludes with further work in section 4.

2. Previous Work

2.1. Concurrency in instance access

Several techniques have been proposed to increase concurrency among instance access methods (Agrawal, 1992; Badrinath, 1988; Badrinath, 1992; Chrysanthis, 1991). In order to decide commutativity of instance access methods, they require application programmers to perform semantic analysis on the methods. In (Agrawal, 1992), they use *right backward commutativity* to provide more concurrency among methods. But, in order to support *right backward commutativity*, application programmers need to know all possible outcomes of each method and recovery should be based on update-in-place policy. In (Badrinath, 1988), the affected set of each method is adopted to give fine concurrency. In their work, two methods commute if the intersection of their affected set is disjoint. But, application programmers need to know the affected set of each method. The recoverability is used to provide enhanced concurrency in (Badrinath, 1992). An operation o_1 is *recoverable* relative to another operation o_2, if o_2 returns the same value whether or not o_1 is executed immediately before o_2. This work requires application programmers to know all outcomes of each method for possible input parameters. A formal scheme to extract concurrency from method is presented in (Chrysanthis, 1988). To support this work, application programmers need to know effects of each method. Also, dependency relation between each pair of method should be provided by application programmers.

Recently in (Malta, 1993), the process of constructing commutativity relation from method contents is automated. It is based on the notion of *affected sets of attributes* (Badrinath, 1988). That is, even if two methods conflict in terms of read or write operations, as long as their access modes on individual attributes do not conflict, two methods can run in parallel. Commutativity of methods is determined at compile-time so that run-time commutativity checking is avoided. As a preliminary step to construct commutativity relation among methods, they construct an Direct Access Vector(DAV) for each method. An DAV is a vector whose field corresponds to each attribute defined in the class on which the method operates. Each value composing this vector denotes the most restrictive access mode used by the method when accessing the corresponding field. An access mode of an attribute can have one of three values, N (Null), R (Read) and W (Write) with N < R < W for their restrictiveness. Access mode information is syntactically extracted from the source code of the method at compile-time. After the construction of DAVs of methods, commutativity of methods can be constructed as follows : two methods commute if their corresponding DAVs commute. In turn, two DAVs commute if their access modes are compatible for each attribute. This commutativity relation is defined in the form of a table.

The above technique takes access mode information solely from the source code of a method and thus frees the user from determining commutativity relations. Also, this approach can provide finer concurrency than mere read and write conflicts by examining attribute level. Since a DAV of a method is the union of its own DAV and DAVs of all other methods defined in that method, deadlocks due to lock escalation can be reduced by declaring the most exclusive access mode in a method. However, concurrency improvement offered by this technique is limited since run-time information on attributes is not taken into account.

2.2. Concurrency between class definition access

In existing OODB systems such as Orion, O_2 and Gemstone (Cart, 1990; Kim, 1990; Servio, 1990), a class definition update (also called schema update or class write) requires a lock on an entire class object. Thus, no matter what kind of update operation is performed on a class object, it blocks all other class definition access operations even if they need to access disjoint portions of the class object.

Recently, in (Agrawal, 1992), they provide more concurrency for class definition updates by providing finer locking granularity. But, class definition updates in their work are limited to updates on attributes and methods. For definition update on method, they classify it into three categories: 1) add a method to a class 2) delete an attribute from a class 3) replace the implementation of a method by a new implementation. For update on attribute, they classify it into two categories: 1) add an attribute to a class 2) delete an attribute from a class. Thus, as long as two class definition update methods access disjoint portions of a class definition, they can run concurrently. But, they do not consider any update on class hierarchy relationship. Also, it is suitable only for OODBS whose schema is continuously changing. For OODBS whose schema need not be changed frequently, the overhead may outweigh the concurrency provided.

2.3. Concurrency between instance access and class definition access

In most concurrency control schemes dealing with class definition update, a class definition update method blocks every instance methods as well as class definition read methods (Cart, 1990; Kim, 1990; Servio, 1990; Malta, 1991). Those studies take lock granularity as an entire class definition. In (Kim, 1990), since it provides a limited set of lock types, a class definition read does not commute with any instance write method. On the other hand, a class definition read method commutes with any instance read or write method in other studies (Cart, 1990; Servio, 1990; Malta, 1991).

Recently, in (Agrawal, 1992), they provide more concurrency for class definition updates by providing finer locking granularity. But, class definition updates in their work are limited to updates on attributes and methods. Also, each attribute accessed by an instance access method is also locked. This attribute locking is done individually at run-time, and thus incurs large overhead.

In this paper, we provide new commutativity relations among class definition update methods in order to increase concurrency. Our scheme includes all types of class definition updates and class definition read operations. We also provide a scheme which integrates instance access and class definition access to give better concurrency.

3. Our Approach

3.1. Concurrency among instance access

Our work improves the scheme developed in (Malta, 1993) to achieve further increase in concurrency. Our scheme has four objectives. First, it still automates the process of commutativity relation construction. Second, it provides more concurrency than read and write access modes on methods. Third, it reduces deadlocks due to lock escalation. Finally, it increases concurrency among methods by exploiting run-time information.

Similar to (Malta, 1993), we need a two-phase pre-analysis which consists of two steps : 1) construction of DAV for each method and 2) construction of a commutativity table of methods. In each method, a break point is inserted by a programmer or a compiler when a conditional statement is encountered. Every method has a special break point called *first break point* before the first statement in the method. There are three kinds of DAVs in each method : 1) a final DAV of the first break point, which is a DAV of the entire method as in (Malta, 1993) 2) an initial DAV of the first break point, which is a union of access modes of each attribute used by statements between the first break point and the next break point and access modes of each attribute used by statements that are executed regardless of execution paths. A union operation is equivalent to *max*, e.g., N + W = W and 3) an initial DAV of every other break point, which contains access modes of all attributes used by statements between this break point and the next break point (or end of the method).

For example, assume that we have three methods M1, M2 and M3 and an class Y with four attributes a_1, a_2, a_3 and a_4. A, A1, A2, and A3 are breakpoints of M1, B is a breakpoint of M2, and C, C1, and C2 are breakpoints of M3. A union operation indicated as ⊕' takes two arguments among N (null: no operation), R (Read), and W (Write) and selects the more restrictive one. Table 1 illustrates how the union operation works.

	N	R	W
N	N	R	W
R	R	R	W
W	W	W	W

Table 1. Union operation table

The contents and DAVs of each break point in the method are given below.

method M1
[A]
read a_1
If (a_1 > 100) then
{[A1]
a_2 <= a_1
End if
read a_2 (*)
If (a_2 > 100) then
[A2]
a_3 <= a_2
End if
read a_3 (**)
If (a_3 > 100) then
{[A3]
call M2
End if

method M2
[B]
read a_1
read a_4
a_4 <= a_1

method M3
[C]
read a_1
If (a_1 > 100) then
{[C1]
return a_1}
else
{[C2]
read a_2
return a_2}
end if

The DAVs constructed for method M1 are :

initial DAV of [A] = {DAV of [A]} ⊕ {DAV of (*)} ⊕ {DAV of (**)}

= [R,N,N,N] ⊕ [N,R,N,N] ⊕ [N,N,R,N]
= [R,R,R,N]
initial DAV of [A1] = [R,W,N,N]
initial DAV of [A2] = [N,R,W,N]
initial DAV of [A3] = final DAV of M2 = [R,N,N,W]
final DAV of [A] = initial DAV of [A] ⊕ initial DAV of [A1] ⊕ initial DAV of [A2] ⊕ initial DAV of [A3]
= [R,R,R,N] ⊕ [R,W,N,N] ⊕ [N,R,W,N] ⊕ [R,N,N,W]
= [R,W,W,W]

Similarly, the DAVs for M2 are :

Final DAV of [B] = [R,N,N,W]
initial DAV of [B] = [R,N,N,W]

and DAVs for M3 are

Final DAV of [C] = [R,R,N,N]
initial DAV of [C] = [R,N,N,N]
initial DAV of [C1] = [R,N,N,N]
initial DAV of [C2] = [N,R,N,N]

While in the scheme proposed in (Malta, 1993), the DAVs for the methods would be:

DAV of M1 = [R,W,W,W]

DAV of M2 = [R,N,N,W]

DAV of M3 = [R,R,N,N]

After the construction of the breakpoints' DAVs in all methods, we create a commutativity relation of methods in the form of a table. In this table, a lock requester's entries contain names of the final DAVs of the first break points in all methods (denoted as N_F where N is the name of the first break point in each method). For example, A_F represents a final DAV of the first break point A in method M1, which is [R,W,W,W]. A lock holder's entries contain names of final DAV of the first break point (denoted as N_F), name of the initial DAV of the first break point (denoted as N_B) and names of the initial DAVs of other break points (denoted as Ni where 1 ≤ i ≤ number of breakpoints -1) in each method. For example, in method M1, A_F, A_B, A1, A2 and A3 represent the following DAVs, [R,W,W,W], [R,R,R,N], [R,W,N,N], [N,R,W,N] and [R,N,N,W], respectively. Since we assume the worst case access mode for each attribute before execution, lock requesters always have the most restrictive access modes (i.e., final DAVs of the first break points). But, after a method execution, a lock holder may have a less restrictive access mode (i.e., initial DAV of the first or of the other break points). Two break points commute if their corresponding DAVs commute. Two DAVs commute if, for every attribute, its access mode in the two DAVs commute. Fig. 1 gives the commutativity tables constructed in our scheme and in the scheme proposed in (Malta, 1993).

Our concurrency control is based on two-phase locking (Eswaren, 1976). When a transaction invokes a method on an object, it gets a lock containing the final DAV of the first break point in the method. As the transaction meets a break point during run-time, the break point is recorded. After the method execution, the lock is changed from N_F to N_B, N_J,...N_S where N_B is the name of the initial DAV of the first break point and N_J...N_S are the names of the initial DAVs of other break points encountered during the method execution. Since the union of DAVs of N_B, N_J,...N_S may be less restrictive than the DAV of N_F, this can give more concurrency to other transactions which request locks on the same object. For example, assume that a transaction T1 invokes a method M1 on instance i1 of class Y and has break points A_B, A_1, and A_2 after the execution of M1. Assume that another transaction T2 comes and invokes a method M2 on the same instance i1 while T1 still has a lock on i1. Applying our work gives commutativity between M1 and M2 since a method M2 commutes with each of A_I, A_B and A_2, by the commutativity table in Fig. 1. On the other hand, M1 and M2 do not commute by checking the commutativity table in Fig. 1 if the scheme in (Malta, 1993) is adopted. Note that O means commute, and X not commute.

Commutativity table of our scheme

lock holder		A_F	A_B	A1	A2	A3	B_F	C_F	C_B	C1	C2
lock	A_F	X	X	X	X	X	X	X	O	O	X
requester	B_F	X	O	O	O	X	X	O	O	O	O
	C_F	X	O	X	O	O	O	O	O	O	O

Commutativity table in (Malta, 1993)

lock holder		M1	M2	M3
lock	M1	X	X	X
requester	M2	X	X	O
	M3	X	O	O

Fig. 1. Examples of commutativity tables constructed for our scheme and for (Malta, 1993)

3.2. Concurrency in class definition access

Class definition updates can be classified in three categories in OODBs (Kim, 1990; Zicari, 1991). The first and second categories are updates to the definition of a class, that is, to attributes of a class and to methods of a class.

These updates include any changes to the attributes and methods defined for a class, such as changing the name or domain of an attribute, adding or dropping an attribute or a method. The third category is updates to a class-hierarchy structure. These include adding or dropping a class, and changing the superclass/subclass relationship between a pair of classes. We use CA, CM, and CCR to denote *Changes to an attribute*, *Changes to a method*, *, and Changes to the superclass/subclass relationship*, respectively. Likewise, for class definition reads, we use RA, RM and RCR to denote *read to definition of attributes*, *read to definition of methods*, *read to superclass/subclass relationship*, respectively.

We assume that updating the definition of a method does not affect the definition of any attribute. On the other hand, we assume that updating the definition of an attribute affects the definition of a method. This provides commutativity between CM and RA. But, we still assume that updating the definition of a class relationship may affect three definitions (attribute, method, class relationship) of a class. Based on these assumptions and the commutativity among class definition updates and class definition reads in (Cart, 1990; Kim, 1990), the following figure gives the commutativity relationships among class definition updates and class definition reads. The commutativity table defines relationships between lock requesters and lock holders on the same class.

	CA	CM	CCR	RA	RM	RCR
CA	X	X	X	X	X	O
CM	X	X	X	O	X	O
CCR	X	X	X	X	X	X
RA	X	O	X	O	O	O
RM	X	X	X	O	O	O
RCR	O	O	X	O	O	O

Fig. 2. Commutativity relationships among class definition updates and class definition reads

Using the above commutativity relationships, for class definition access methods, we get finer granularity locks and thus provide better concurrency than conventional OODBS such as Orion (Kim, 1990) and O_2 (Cart, 1990) do. The lock granularity in our work is one of CA, CM and CCR (for class definition update) and RA, RM. RCR (for class definition read). Whenever a class definition access method is invoked, we need to check commutativity between a lock holder and a lock requester using the commutativity table in Fig. 2 and grant a lock if they commute. The lock table format is of [*trans-name*, *lock-type*] where *trans-name* is a transaction holding a lock and *lock-type* is a class definition access lock type ∈ {CA, CM, CCR, RA, RM, RCR}. For example, consider the following transactions on class Y.

(transactions)	T1	T2
(time)		
t	CM (delete a method)	
t+1		RA (read an attribute)
t+2	RA (read an attribute)	
t+3		CA (delete an attribute)

The following table shows the locks obtained , at each time step, during the executions of transactions T1 and T2 in our scheme. We assume that the first execution occurs at time t.

<u>time</u> <u>locks obtained by each transaction</u>

----

t Y : [T1,CM]

// T1 can get a CM lock since no other transaction has a lock on class Y //

t+1 Y : [T1,CM] [T2,RA]

// T2 can get an RA lock since CM and RA commute using the table in Fig. 2 //

t+2 Y : [T1,CM] [T2,RA] [T1,RA]

// T1 can get an RA lock since RA and RA commute //

t+3 Y : [T1,CM] [T2,RA] [T1,RA]

// T2 cannot get a CA lock since CA does not commute with RA //

3.3. Concurrency between class definition access and instance access

For each class, when it is accessed for the first time by either an attribute definition access method or instance access method, an attribute access vector (AAV) is created. Also, when each class is accessed for the first time by a method definition access method or instance access method, a method access vector (MAV) is created. An AAV is to give parallelism between attribute definition access methods and between attribute definition access methods and instance access methods. Likewise, an MAV is to give parallelism between method definition access methods, and

between method definition access methods and instance access methods. Each field in the AAV represents an attribute. For each attribute field, a value can have one of three values: W (update, set by CA), R (read, set by RA or RM or CM or instance access method), and N (null). Each field in MAV represents a method. For each method field, a value can have one of three values: W (update, set by CM), R (read, set by RM or instance access method), and N (null). These vectors are updated when a class definition access on attribute (or method) or instance access method is granted a lock.

The use of these vectors to increase concurrency is done as follows.

- Lock requester is an CCR method: if lock is set by other transactions, block it. Otherwise, set CCR lock.
- Lock requester is an RA or CA method: if CCR lock is set, block it. Otherwise, commutativity is checked by using AAV and sets R (for RA) or W (for CA) on the corresponding attribute in AAV if, for each attribute to be accessed by the lock requester's method, the lock modes of the requester and holders are compatible.
- Lock requester is an RM or CM method: if CCR lock is set, block it. Otherwise, the commutativity checking consists of two steps. Assume that the definition access to the method M1 is requested.

    a) commutativity is checked by comparing AAV with the M1's DAV as follows: for each attribute whose value is R or W in the DAV of M1, check if the attribute is W locked in AAV. If so, block the lock request on M1. Otherwise, perform step b) as below.

    b) check if the method field of M1 is W locked in MAV. If so, block the lock request on M1. Otherwise, set R (for RM) or W (for CM) lock in MAV, and R lock (in AAV) for each attribute whose value is R or W in DAV of M1.

- Lock requester is an instance method: if CCR lock is set, block it. Otherwise, the commutativity checking consists of two steps.

    a) commutativity is checked by comparing AAV with the lock requester's DAV as follows: for each attribute whose value is R or W in the DAV of the instance access method, check if the attribute is W locked in AAV. If so, block the lock request by the instance access method. Otherwise, perform step b) as follows.

    b) check if the method field is W locked in MAV. If so, block the lock request by the instance method. Otherwise, set R lock (in MAV) in the corresponding method's field, and R lock (in AAV) for each attribute whose value is R or W in DAV of the instance access method.

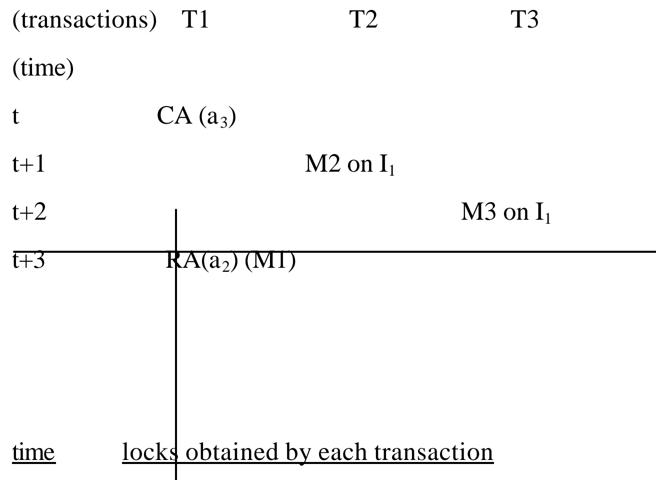
- Whenever an CA or RA is committed by its invoking transaction, the vector AAV is reset.
- Whenever an CM or RM or instance access method is committed by its invoking transaction, the vectors MAV and AAV are reset.

The following table gives the commutativity relationships among class definition updates (CA,CM,CCR), class definition reads (RA,RM,RCR), and instance access methods (denoted by I) where Δ means that two methods commute as long as they are accessing disjoint portions of an object.

	CA	CM	CCR	RA	RM	RCR	I
CA	Δ	Δ	X	Δ	Δ	O	Δ
CM	Δ	Δ	X	O	Δ	O	Δ
CCR	X	X	X	X	X	X	X
RA	Δ	O	X	O	O	O	O
RM	Δ	Δ	X	O	O	O	O
RCR	O	O	X	O	O	O	O
I	Δ	Δ	X	O	O	O	Δ

Fig. 3. Commutativity relationship among class definition access and instance access

For example, with class Y defined in Section 3.1, consider the following method invocations by transactions T1, T2 and T3. The following shows the locks obtained and changes in the vectors AAV and MAV, at each time step, during the execution of transactions T1, T2 and T3. We assume that the first execution occurs at time t.



(transactions)	T1	T2	T3
(time)			
t	CA (a_3)		
t+1		M2 on I_1	
t+2			M3 on I_1
t+3	RA(a_2) (M1)		

time | locks obtained by each transaction

t : Y : AAV [a_1:N, a_2:N, a_3:W(T1), a_4:N]

// Lock CA requested by T1 is granted since no other transaction has a lock on Y. Thus, T1 needs to create AAV and sets W on a_3 field. //

t+1 : Y : AAV [a_1:R(T2), a_2:N, a_3:W(T1), a_4:R(T2)]

MAV[a_1:N, a_2:R(T2), a_3:N]

I_1 : [M2(B_F), T2]

// T2 invokes M2 on I_1; check AAV if, for each attribute accessed by M2, there is an incompatible attribute access mode using DAV of M2. Also check the M2 field in MAV if some other transaction is updating M2. Since M2 does not access attribute a_3 and the definition of M2 is not updated, the values of AAV and MAV are changed and M2 can get a lock on instance I_1.//

t+2: Y: AAV[a_1:R(T2,T3),a_2:R(T3),a_3:W(T1),a_4:R(T2)]

MAV[a_1:N, a_2:R(T2), a_3:R(T3)]

I_1 : [M2(B_F), T2], [M3(C_F), T3]

// Repeat the work done in step t+1 for T3. By using commutativity table in Fig. 1, M2 request by T2 is granted on I_1. //

Assume that break points C_B and C_2 are met during the execution of M3 //

t+3: Y:AAV[a_1:R(T2,T3),a_2:R(T2,T3),a_3:W(T1),a_4:R(T2)]

MAV[a_1:N, a_2:R(T2), a_3:R(T3)]

I_1 : [M2(B_F), T2], [M3(C_B, C_2), T3]

// Check a_2 field in AAV if another transaction is already updating a_2.. If so, block the requester. Otherwise, set R on a_2 in AAV. Since a_2 is not being updated by any transaction, RA lock requested by T2 is granted. //

Note that the instance access requests by T2 and T3 are blocked if we adopt the locking schemes in (Cart, 1990; Kim, 1990). In our work, we can increase concurrency among class definition access and instance access by taking finer granularity locks on class definition access.

One may argue that updating AAV and MAV whenever an instance access method is invoked incurs too much overhead. This is true especially for OODB systems whose schema need not be changed frequently. In this case, the overhead imposed by the technique proposed here may outweigh the concurrency increased. For such OODB systems, we take granularity as all attributes for RA or CA and all methods for RM or CM rather than individual attribute or method. Also, for instance access methods, we use RA and RM locks on class, instead of using AAV and MAV. That is, we adopt the following protocol, which is based on the technique discussed in Section 3.2.

- When a transaction invokes an instance access method, get RA and RM locks and check commutativity among instance access methods.
- When a transaction which has invoked an instance access method is committed, release RA and RM locks.

4. Further Work

This paper presents an integrated concurrency control scheme to enhance concurrency among methods in OODBS. The scheme deals with concurrency among instance access, among class definition access, and among class definition access and instance access. Especially, for better concurrency among class definition access and instance access, our scheme provides different treatments for two types of object-oriented databases : one whose schema is continuously changing, and one whose schema needs not be changed frequently.

In our work, an instance access method may have many break points depending on the method's logic. This requires larger commutativity tables and also incurs much run-time overhead for lock changes and commutativity checking. Thus, we need a way to reduce the number of break points in a method in order to reduce space and time overhead for lock changes and commutativity checking during run-time. In this paper, we do not include the techniques used to reduce break points. Also, in this study, we do not consider class hierarchy, which is an important property in OODBS. Due to inheritance, a query-type access or class definition access may involve a class and all its subclasses. Currently, we are developing a scheme, which incorporates class hierarchy into our current work, aiming at less locking overhead on class hierarchy.

References

Agrawal, D. and Abbadi, A. E. (1992). A Non-Restrictive Concurrency Control for Object-Oriented Databases. In Proc. of the 3rd Int. Conf. on Extending Data Base Technology (pp. 469--482), Vienna, Austria.

Badrinath, B. R. and Ramamritham, K. (1988). Synchronizing Transactions on Objects. IEEE Transaction on Computers, 37(5), 541--547.

Badrinath, B. R. and Ramamritham, K (1992). Semantic-Based Concurrency Control : Beyond Commutativity. ACM Transactions on Database Systems, 17(1), 163--199.

Bernstein, P. A. and Goodman, N. (1981). Concurrency Control in Distributed Database Systems. ACM Computing Surveys. 13(2), 185--221.

Bernstein, P. A., Hadzilacos, V. and Goodman, N. (1987). Concurrency Control and Recovery in Database Systems. Addison-Wesley.

Cart, M and Ferrie, J, Integrating (1990). Concurrency Control into an Object-oriented Database System 2nd Int. Conf. on Extending Data Base Technology (pp. 363--377). Venice, Italy.

Chrysanthis, P. K., Raghuram, K, and Ramaritham K. (1991) Extracting Concurrency from Objects : A Methodology Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data (pp. 108--117).

Eswaren, K. P., Gray, J. P., Lorie, R. A., and Traiger, I. L. (1976). The Notions of Consistency and Predicate Locks in a Database System Communications of the ACM. 19(11), 624--633.

Kim, W (1990). Introduction to Object-Oriented Databases. The MIT Press

Malta, C. and Martinez, J. (1991). Controlling Concurrent Accesses in an Object-Oriented Environment. 2nd Int. Symp. on Database Systems for Advanced Applications. Tokyo, Japan, (pp. 192 - 200)

Malta, C. and Martinez, J. (1993) Automating Fine Concurrency Control in Object-Oriented Database. Proc. of the 9th Int. Conf. on Data Engineering. Vienna, Austria. (pp. 253 - 260)

Servio Logic Corp. (1990). Chap. 16 : Transactions and Concurrency Control. in Gemstone Product Overview. Alameda, CA.

Zicari, R. (1991) A Framework for Schema Updates in An Object-Oriented Database System. 7th IEEE Int. Conf. on Data Engineering. Kobe, Japan. (pp. 2--13)