A Flexible Class Hierarchy Locking Technique in Object-Oriented Database Systems

Woochun Jun and Le Gruenwald
Dept. of Computer Science
University of Oklahoma
Norman, OK 73019

Abstract

In this paper, we present a locking-based concurrency control scheme for object-oriented databases (OODBs). Our scheme deals with class hierarchy which is an important property in OODBs. Our scheme is based on so called special classes and can be used for any applications, with less overhead, by adjusting the number of special classes in a class hierarchy. For our scheme, we prove the correctness.
Key word: concurrency control, OODB

1. Introduction

An OODB is a collection of classes and instances of these classes. In an OODB, both classes and instances are referred to as *objects*. A class consists of a set of *attributes* and *methods* through which the class' instances are accessed. Users can access objects by invoking methods. In order to make sure atomicity of user interactions, the traditional transaction model can be used in an OODB. That is, users can access an OODB by executing transactions, each of which is defined as a partially ordered set of method invocations on a class or an instance object [1]. In general, there are two types of access to an object: instance access (instance read and instance write) and class definition access (class definition read and class definition write) [2]. An instance access consists of consultations and/or modifications of attribute values in an instance or a set of instances. A class definition access consists of consultations of class definition and/or modifications of class definition such as adding/deleting an attribute or a method.

Concurrency control involves synchronization of access to the database, so that the consistency of the database is maintained [3, 4]. Although a concurrency control scheme allows shareable access among users, it incurs an overhead which may have a critical effect on OODBs, where many transaction are long-lived. Thus, it is necessary to reduce the overhead to improve transaction response time. 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.

One of the major properties of an OODB is inheritance. That is, a subclass inherits definitions (attributes and methods) defined on its superclass. Also, there is an *is-a* relationship between a subclass and its superclass. Thus, instance of a subclass is a specialization of the superclass (and conversely, instance of a superclass is a generalization of subclass). This inheritance relationship between classes forms a class hierarchy. There are two types of operations on a class hierarchy: *class definition write* and *queries* [5]. The class definition writes include adding/deleting an attribute or a method, changing domain of an attribute, adding/deleting a class, and so on. Thus, while a class and its instances are being accessed, the definitions of the class' superclasses should not be modified. Also, due to the *is-a* relationship between classes, the search space for a query against a class, say C, may include the instances of the class hierarchy rooted at C as well as instances of C.

In this paper, we present a locking-based concurrency control scheme for class hierarchy in OODBs. In the literature, there are two approaches dealing with class hierarchy: *explicit locking* and *implicit locking*. Both approaches may work well for only specific applications in OODBs. Especially, in explicit locking, it may have less locking overhead only for transactions concerned with only non-query type instance access (i.e., instance access performed for only one class) or class definition read. On the other hand, in implicit locking, it may have less locking overhead only for transactions concerned with query or class definition write. Our scheme is based on a so called *special class*, which will be defined in Section 3, and can be used for any applications, with less locking overhead, by adjusting the number of special classes. We show that our scheme works correctly.

This paper is organized as follows. In the next Section, we present previous studies and discuss their advantages and disadvantages. In Section 3, we discuss our class hierarchy locking scheme. The paper concludes with further work in Section 4.

2. Related Work

As discussed in Section 1, due to inheritance, class definition write methods and queries on a class may need to access more than one class on a class hierarchy. There are two major existing approaches to perform locking on a class hierarchy: *explicit locking* [2, 6] and *implicit locking* [5,7]. In *explicit locking*, for a query involving a class, say C, and all of its subclasses, and for a class definition write on a class C, a lock is set not only on the class C, but also on each subclass of C on the class hierarchy. For other types of access, a lock is set for only the class to be accessed (we call *target class*). Thus, for a class definition write or query, this protocol reduces the number of locks required by transactions that access a class near the leaf level of a class lattice than transactions that access a class near the root of a class lattice. As another advantage of *explicit locking*, it can treat *single inheritance* where a class can inherit the class definition from one superclass, and *multiple inheritance* where a class can inherit the class definition from more than one superclass, in the same way. But, it increases the number of locks required by transactions that access a class at a higher level in the class lattice. Also, for class definition writes or queries, this protocol may need to search all subclasses of a class, say C, in order to decide if a lock on class C can be granted or not.

In *implicit locking*, setting a lock on a class C requires extra locking on a path from C to its root as well as on C. *Intention* locks [8, 9] are put on all the ancestors of a class before the target class is locked. An intention lock on a class indicates that some lock is held on a subclass of the class. Thus, for class definition writes or queries on a target class, a concurrency control protocol needs not search all subclasses of the target class for conflict checking. It only searches from the target class to a root of the target class. But, *implicit locking* requires a higher locking cost when a target class is near the leaf level in the class hierarchy.

For example, consider the following class hierarchy. In order to update the class definition in class, say C, each scheme works as follows.

Implicit locking in Orion [5]		*Explicit locking* in O_2 [2]
IW lock	A	
	↓	
IW lock	B	
	↓	
W lock	C	Cw lock
	↓	
	D	Cw lock
	↓	
	E	Cw lock

In *implicit locking*, intention locks IWs corresponding to W (Write) locks are required for all classes on the path from C to the root A. Thus, if another transaction needs to update the class definition in A, it does not have to search each class through the class hierarchy for conflict checking by the help of the intention lock IW on class A. On the other hand, explicit locking does not require any intention locks. But, it requires a Cw (Class Write) lock on each subclass (i.e., class D and E) of the target class through the class hierarchy since any modification of class definitions in C may affect the class definitions of its subclasses.

3. Proposed class hierarchy locking

3.1. Basic idea

As we have discussed, two existing class hierarchy locking schemes may work well for only specific OODB environments. Our work is to develop a new class hierarchy locking scheme which can be used for any OODB applications with less locking overhead. To achieve this, we designate some classes in the class hierarchy as *special classes*. A *special class* is a class on which class definition writes or queries are performed frequently.

In our scheme, intention locks are set on special classes only; thus, locking overhead is reduced compared to those in implicit locking. When a transaction needs to access a *special class* which is already intention locked, by invoking a class definition write or query on it, it is not necessary for a concurrency control to search the subclasses of the special class due to the help of the intention lock. On the other hand, if a class has little or no possibility to be accessed by a class definition write or query, there is no need to set an intention lock on that class since any access other than class definition writes or queries does not use the intention lock to check

conflict. Thus, unlike implicit locking, we do not have to set an intention lock on every class on the path from a target class to a root.

In order to reduce locks required for a class definition write or query more than explicit locking, our scheme works as follows: for a non-query type instance access or class definition read, a lock is set on only the target class, like explicit locking. For class definition writes or queries, locks are set on every class from the target class to the first special class through the subclass chain of the target class (if there is no such special class, then set locks to leaf classes). Thus, as we have discussed, by choosing a special class as a class on which class definition writes or queries are performed frequently, we can reduce locking overhead.

Note that, the use of special classes in our class hierarchy locking scheme is flexible. For those applications in which there is no or few class definition writes or queries, special classes can be ignored. That is, we do not implement intention locks. So, in this case, our scheme behaves the same as explicit locking. For those applications in which class definition writes or queries are performed frequently, special classes are needed in order to have lower lock overhead than explicit locking and implicit locking.

For example, consider the following class hierarchy (Fig 1.a). Assume that a transaction T1 invokes a class definition write or query method on class C7. Let LS1 be a lock setting for T1. Fig. 1.b, 1.c and 1.d show how explicit locking, implicit locking, and our scheme are done. SC denotes *special class*.

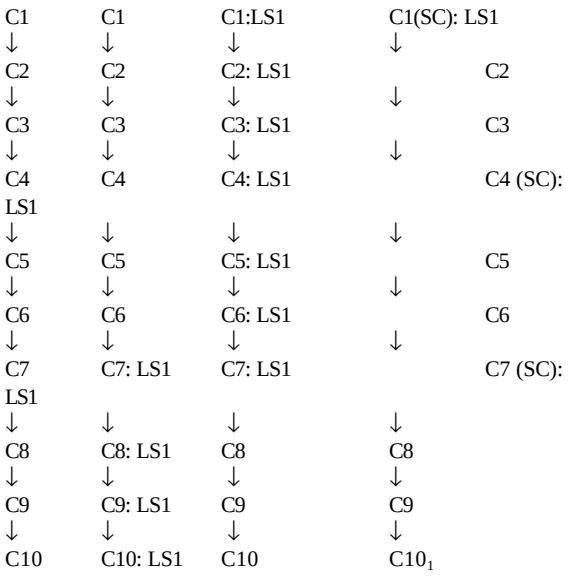


Fig. 1.a Fig 1.b Fig 1. c Fig. 1.d

Class hierarchy	Explicit locking	Implicit locking	Our scheme

3.2. Lock Modes

For the description of our class hierarchy locking, we have the following lock modes for class definition access and instance access. We adopt instance level granularity for instance access methods and entire class object for class definition access, like Orion [5] or O_2 [2]. Thus, we can focus only on class hierarchy locking. For simplicity, we assume that a class has only one superclass (i.e., single inheritance). Below we show locks needed for different types of operations. For convenience, we use upper-case letters and lower-case letters to name locks a class and instance, respectively.

- Class definition write: CW (on target class) and INT-CW(intention lock on every class on the path from the target class to its root)
- Class definition read: CR (on target class) and INT-CR (intention lock on every class on the path from the target class to its root)
- Instance read: r(on target class) and T-R, INT-S-R, IMP-R, QR (on target class or its superclasses)
- Instance write: w (on target instance) and T-W, INT-S-W, IMP-W (on target class or its superclasses)

< Instance read >

- T-R lock means that some (not all) instances of a target class are *r* locked. An T-R lock is set on a *target class* whenever a *r* lock is set on its instance.
- An intention lock INT-S-R is set on every class on the *superclass chain* from a target class to its root whenever any instance read lock is set on the target class or its instance. It indicates that some instance read lock is held on a subclass of the class. Thus, for class definition writes and queries, a concurrency control protocol needs not search all subclasses of the target class for conflict checking. It only searches from the target class to its root.
- IMP-R lock (on target class) means that all instances are *read* locked *implicitly* (i.e., we reduce locking overhead by setting an IMP-R lock on the target class , not individual instances)
- QR (*Query Read* on a target class) means that all instances of subclasses as well as instances of a target class are *read* locked (i.e., we reduce locking overhead by setting an QR lock on only the target class, not setting IMP-R lock on the entire subclasses of the target class)

< Instance write >

- T-W lock (on target class) means that some (not all) instances of a target class are *w* locked. An T-W lock is set on a *target class* whenever *w* lock is set on its instance.
- INT-S-W lock is set on every class on the *superclass chain* from the target class to its root whenever an instance write lock is set on instance or class.
- IMP-W lock means that all instances of a target class are *w* locked *implicitly* (i.e., we reduce locking overhead by setting the lock on the target class, not individual instances)

3.3. Commutativity Relation Table

We provide commutativity relation among lock modes introduced above. In tables 1 and 2, we give two commutativity relations, one for instances and the other for classes. O, X stand for *commute*, and *not commute*, respectively.

a) instance

		lock holder	
		r	*w*
lock	*r*	O	X
requester	*w*	*w*	X X

Table 1. Commutativity relation for an instance

b) Class

	lock holder									
lock	CW	INT-CW	CR	INT-CR	T-R	INT-S-R	T-W	INT-S-W	IMP-R	IMP-W
CW	X	X	X	X	X	X	X	X	X	X
INT-CW	X	O	O	O	O	O	O	O	O	X
CR	X	O	O	O	O	O	O	O	O	O
INT-CR	X	O	O	O	O	O	O	O	O	O
T-R	X	O	O	O	O	O	O	O	O	X
INT-S-R	X	O	O	O	O	O	O	O	O	X
T-W	X	O	O	O	O	O	O	O	X	X
INT-S-W	X	O	O	O	O	O	O	O	O	X
IMP-R	X	O	O	O	O	O	X	O	O	X
IMP-W	X	O	O	O	X	O	X	O	X	X
QR	X	X	O	O	O	O	X	X	O	X

Table 2. Commutativity table for a class

3.4. Class Hierarchy locking algorithm

Our scheme is based on two-phase locking which requires each transaction to obtain a read (or write) lock on a data item before it reads (or writes) that data item, and not to obtain any more locks after it has released some lock [10]. For a given lock request on a class, say Y, we set locks on Y and all classes on the class hierarchy to which the class Y belongs as follows.

Step 1) (applied to all types of lock request): For each *special class* (if any) through the superclass chain of Y, check conflicts and set an *intention lock* if commute. If not commute, block the lock requester.

Step 2) If the lock requesting method is neither CW (class definition write) nor QR (query), check conflicts and set one of T-R, T-W, IMP-R, IMP-W (depending on the lock request type) or CR (class definition read) on only the target class Y if commute and set an *r* or *w* lock on the instance to be accessed (we call *target instance*) if *a* method is invoked on the instance and commute. If not commute, block the requester. If the lock requesting method is CW or QR, then, from class Y to the first *special class* (or leaf class if there is no *special class*) through the subclass chain of Y, check conflicts and set CW or QR lock on each class if commute. We set a lock on the first special class so that we let other incoming transactions that access a subclass of Y check conflicts. Also, the reason we set a lock on each class (besides the first special class) through the subclass chain of Y is as follows: if a lock is set only on the first special class, say, C, then some conflict may not be detected (for example, a requester accesses a subclass of a lock holder's class which is CW locked). If class Y has more than one subclass, then, for each direct subclass of Y, perform the same step 2).

As an example, consider the following lock requests by three transactions T_1, T_2 and T_3 on a class hierarchy in Fig. 2.a.

a) T_1: class definition write(CW) request on class C6
b) T_2: class definition read on class C5
c) T_3: instance read request on class C12

Fig 2.b, 2.c, and 2.d show results of lock settings for T_1, T_2 and T_3 by our scheme, explicit locking, and implicit locking, respectively. Let LS_i be a lock LS set by transaction T_i. Assume that class C1, C4, C7 and C11 are special classes. In our scheme, as we can see, the lock request by T_3 is denied because of a conflict on class C7. Likewise, in explicit locking and implicit locking, the lock request is denied because of a conflict on class C12 and C6, respectively.

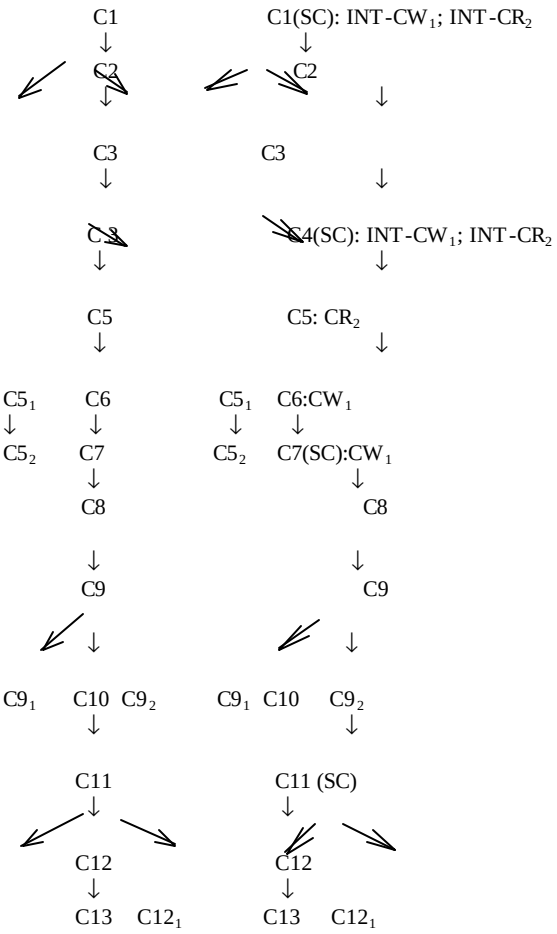
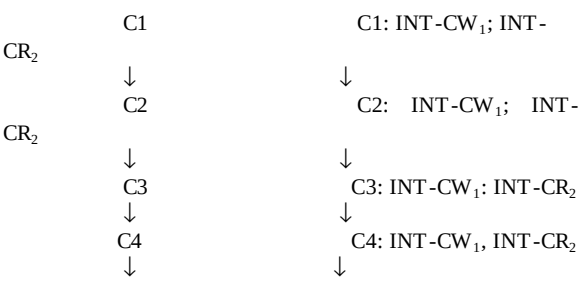


Fig 2.a Class hierarchy

Fig. 2.b Our scheme



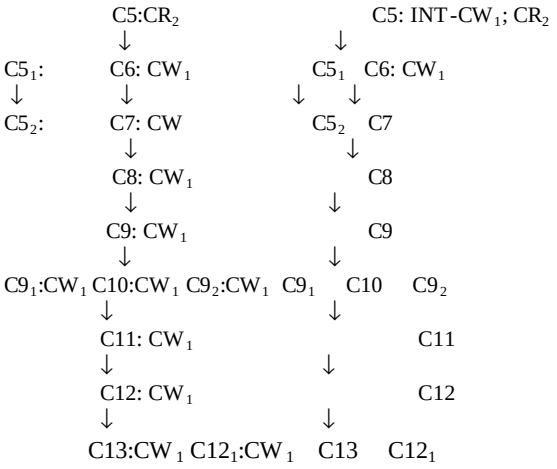


Fig. 2.c Explicit locking

Fig. 2.d Implicit locking

As we can see in Fig. 2, there are 7 lock sets for our scheme, 12 lock sets for explicit locking, and 11 lock sets for implicit locking. Our scheme incurs less locking overhead for queries or class definition writes with the help of special classes.

3.5. Correctness of our proposed class hierarchy locking algorithm

Now, we prove that our algorithm is correct, that is, it satisfies serializability [3]. We prove it by showing that, for any lock requester, it is guaranteed that the conflict with a lock holder (if any) is always detected. With this proof, since our class hierarchy locking scheme is based on two-phase locking, it is guaranteed that our scheme satisfies serializability [3,10]. If a lock requester is either a class definition read or an instance access performed on one class, then its lock holders consist of transactions/methods holding locks on the target class and all special class in the superclass chain of the target class. If a lock requester is a class definition write or query, then its lock holders include those defined above plus transactions/methods holding locks on each class from the target class to the first special class in the subclass chain of the target class.

There are four cases depending on the types of lock requesters and holders.

case i) *the lock holder is a class definition read or an instance access on only one class*

the lock requester is a class definition read or an instance access on only one class

If a lock holder and a lock requester access different classes, there is no conflict. If a lock holder and a lock requester access the same class, there is no conflict on all special classes through the superclass chain of the target class because intention locks on special classes are compatible with the lock requester; conflicts can be detected on the target class (for example, IMP-W (requester) and T-R (holder)) or target instance (for example, T-R and T-W)

case ii) the lock holder is a class definition read or an instance access on only one class
the lock requester is a class definition write or query

If the lock holder is holding a lock on a superclass of the lock requester's class, there is no conflict since the lock requester does not access the holder's class. If not, a conflict is checked either on the nearest *special class* through the subclass chain of the lock requester's class if there is a special class between the holder's class and the requester's class (case 1) or on the lock holder's class, otherwise (case 2). For case 1, there are two subcases as follows:

a) the lock holder is a special class(Fig. 3.a):
b) the lock holder is not a special class (Fig. 3.b)

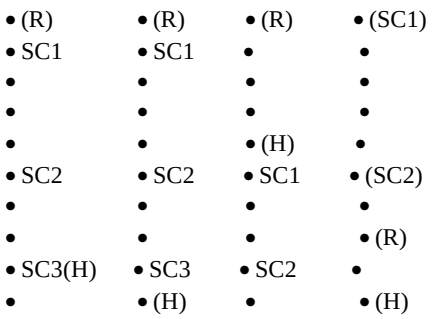


Fig 3. a Subcase a of case 1 | Fig. 3.b Subcase b of case 1 | Fig. 3.c Subcase a of case 2 | Fig. 3.d Subcase b of case 2

Let R and H be two classes on which the lock requester requests a lock and the lock holder holds a lock, respectively. In either case, a conflict (if any) is checked on SC1, which is the nearest *special class* of the lock requester's class through its subclass chain, since the holder has an intention lock on SC1 and the requester requests CW or QR on SC1. On the other hand, in case 2, for subcase a, a conflict (if any) is checked on H as in Fig. 3.c since the holder does not have any intention locks through the superclass chain of R and the requester needs to set a CW or QR lock on H. For subcase b, conflict (if any) is checked on H as in Fig 3.d since intention locks on all special classes through the superclass chain of H are compatible and the requester needs to set a CW or QR lock on H.

For other cases (case iii: the lock holder is a class definition write or query; the lock requester is a class read or instance access on only one class and case iv: the lock holder is a class definition write or query; the lock requester is a class definition write or query), we can prove similarly as in case ii). Thus, we can conclude that, for any lock requester, it is guaranteed that its conflict with a lock holder (if any) is always detected. Also, since our scheme is based on two-phase locking, serializability is guaranteed [10].

4. Further work

In this paper, we present a concurrency control scheme for class hierarchy in OODBs. Our scheme is based on *special classes* to reduce locking overhead. Our scheme can be used in any application in OODBs by adjusting the number of special classes.

In our work, locking granularity is instance object and class object for instance access and class definition access, respectively. We are currently developing a class hierarchy locking scheme with finer granularity. That is, we are adopting attribute level granularity instead of instance, and individual class definition instead of entire class object, in order to provide better concurrency among transactions.

References

[1] D. Agrawal and A. E. Abbadi, "A Non-restrictive Concurrency Control for Object-Oriented Databases", 3rd Int. Conf. on Extending Data Base Technology, Vienna, Austria, p. 469 - 482, Mar. 1992.

[2] M. Cart and J. Ferrie, "Integrating Concurrency Control into an Object-Oriented Database System", 2nd Int. Conf. on Extending Data Base Technology, Venice, Italy, pp. 363 - 377, Mar. 1990

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

[4] M. T. Ozsu and Patrick Valduriez, Principles of Distributed Database Systems, Prentice Hall, 1991.

[5] J. F. Garza and W. Kim, "Transaction Management in an Object-Oriented Database System", ACM SIGMOD Int. Conf. on Management of Data, Chicago, Illinois, pp. 37 - 45, Jun. 1988.

[6] Carmelo Malta and Jose Martinez, “Automating Fine Concurrency Control in Object-Oriented Databases”, 9th IEEE Conf. on Data Engineering, Vienna, Austria, pp. 253- 260, Apr. 1993.
[7] H. V. Jagadish and Daniel F. Lieuwen, “Multi-Granularity Locks in an Object-Oriented Database”, AT&T technical report, 1993.
[8] C. J. Date, *An Introduction to Database Systems*, Vol. II, Addison-Wesley, 1985.
[9] Henry F. Korth and Abraham Silberschartz, *Database System Concepts*, 2nd Edition, McGraw Hill, 1991.
[10] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, “The notion of consistency and predicate locks in a database system”, Communication of ACM, Vol 19, No. 11, pp. 624 - 633, Nov. 1976.