Design and Implementation of a Concurrency Control Mechanism in an Object-Oriented Database System

Qu Yunyao (曲云尧), Tian Zengping (田增平), Wang Yujun (王宇君) and Shi Baile (施伯乐)

Department of Computer Science, Fudan University, Shanghai 200433

Received May 10, 1995; revised January 24, 1996.

Abstract

This paper presents a practical concurrency control mechanism — Object-Locking in OODBMS. Object-Locking can schedule transactions, each of them can be considered as a sequence of high level operations defined on classes. By the properties of parallelity and commutativity between high level operations, proper lock modes for each operation are designed and the compatibility matrix is constructed. With these lock modes, phantoms are kept away from databases and a high degree of concurrency is achieved.

Keywords: Concurrency control, object-oriented database, transaction management.

1 Introduction

Recent years have seen dramatic research and development progress in the area of OODB (e.g., Orion, O2, Iris, Gemstone, etc.[1,2]). Much effort has been devoted to models, languages, architecture, and data management[3], but few publishment have appeared on designing concurrency control mechanisms. For the next generation information system, concurrency control mechanisms are required to handle **High level operations** on complex objects and to meet high throughput demands. For example, many applications such as Office Information System, CAD/CAM, Software Engineering, etc., require high performance and support of complex information. Traditional concurrency control mechanisms in DBMS where the operations are just **Read and Write**, do not support these applications well.

Semantics-based concurrency control techniques have been widely studied[4–7]. It could exploit richer semantic information from objects and high level operations to get a higher degree of concurrency. Hence, many researchers believe that this kind of concurrency control is best suited for the transaction management of OODB systems[3–5,7–9,]. Nevertheless a number of semantics-based concurrency control schemes proposed for object-oriented systems considered only some special Abstract Data Types, e.g., Stack and Queue[4,5,7]. None of them deal with a practical system.

In this paper, we demonstrate the design and implementation of a semantics-based concurrency control mechanism in an OODB system — FOODB. FOODB is

an OODBMS, which was developed by the Computer Science Department of Fudan University. The concurrency control mechanism is designed to schedule transactions composed of high level operations. There are three kinds of operations in FOODB: **Schema operations**, **Statistical operations**, and **Instance operations**. Schema operations include querying or updating the structure of a class. Statistical operations include querying or updating a set of objects (instances) satisfying some conditions. Instance operations include finding, updating or deleting an instance from a class, or inserting an instance to a class.

2 The FOODB System

2.1 Architecture

FOODB is an object-oriented database system. It has a client/server architecture. Database is stored on the server. All database operations of the clients are delivered to the server, where those operations are executed. The unit of data transfer between client and server is object. The transaction manager is responsible for concurrency control and recovery. The object manager is responsible for managing objects in buffer, swapping objects between disks and memory, and packing or unpacking objects for transfer. Transferring objects between client and server is the task of the communication system. GUI(Graphical User Interface) includes an interactive interface and a programming interface and a set of system maintenance tools. Users may use Object SQL (OSQL) or Extended C++ to query or update database in interactive or programming environment respectively.

2.2 Data Model

FOODB supports fundamental object-oriented concepts: class, attribute, method, object, object identity (OID), encapsulation, inheritance(single or multiple inheritance), polymorphism, etc. Complex objects are constructed through OID references between objects. Three types of references are used:

(1) shared reference: the object may be referenced by several objects;

(2) exclusive reference: the object may be referenced by only one object;

(3) dependent reference: the reference is exclusive and if X references Y then the existence of object Y depends on the existence of object X.

In FOODB, we also regard a class as an object. Schema operations and statistical operations are sent to this kind of objects. Operations in the system are classified as follows.

(1) Schema operations

- Schema query: query the structure of a class;
- Schema evolution: modify the structure of a class, e.g., add or drop a class, change the superclass/subclass relationship between a pair of classes, etc.

We stipulate that the structure of a superclass cannot be modified in its subclasses.

(2) Statistical operations

Statistical operations operate on a set of instances satisfying some given conditions. For example, *selecting employees (from a class) aged over 60* is a statistical operation.

(3) Instance operations

Instance operations include: insert an object to a class, delete an object from a class, query or update an object in a class.

2.3 Transaction Model

A transaction is a sequence of database operations. Users interact with database by invoking transactions. A transaction can access an object or modify the state of an object only by invoking operations (methods) defined for that object. Operations scheduled by the transaction manager are not only simple read or write, but also various high level operations of classes. Users' transactions in a system are a succession of method invocations.

$$\text{Begin } Oid_1.M_1 \cdots Oid_i.M_i \cdots Oid_n.M_n \text{ End}$$

Oid_i: object identifier, M_i: name of a method

3 Commutativity and Parallelity

Parallelism allowed by a concurrency control mechanism depends to a large extent on the properties of parallelity and commutativity between the operations defined on objects. Let X be an object that is usable through operations O_i and O_j. The pair of operations (O_i, O_j) has the property of **Parallelity** if whatever is the initial state of object X, for the concurrent executions of the two operations by two different transactions, the effects on object X and the transactions are the same. In other words, two parallel operations may be performed concurrently without having to be controlled. For example, two query operations are parallel. When two operations are not parallel, **Commutativity** may be used, which is a weaker property.

The pair of operations (O_i, O_j) has the property of Commutativity if, whatever is the state of object X, execution of operation O_i by transaction T_i followed by execution of operation O_j by transaction T_j has the same final effects on object X and the same results for the transactions as the execution in the reverse order (O_j followed by O_i).

Let S be a database state, O be an operation. State (O, S) is a state that is produced by the execution O on state S. Return (O, S) is the return value of operation O executed on state S. We assume that every operation will return a value, at least a status or condition code (e.g. read operation may return a value read by the operation, write operation may return a status indicating success or failure of its execution). The above operation O may also be a sequence of operations. Thus,

in this case, state (O, S) is the state produced after the execution of the operations in O;·return (O, S) is the return value set of operations in O executed on S.

Let $O_1//O_2$ mean that operations O_1 and O_2 execute in parallel. O_1O_2 means that operation O_2 is executed following the execution of O_1.

Definition 1. *Two operations O_1 and O_2 are parallel if O_1 and O_2 can be executed concurrently without any control on the order of execution of their lower level operations, and for all states S, state(O_1 // O_2, S)=state(O_1O_2,S), state(O_1 // O_2,S)=state(O_2O_1,S), return(O_1 // O_2,S)=return(O_1O_2, S), return(O_1 // O_2,S)= return(O_2O_1,S).*

Definition 2. *Two operations O_1 and O_2 are commutative if, for all states S, state(O_1O_2,S)=state(O_2O_1,S), return(O_1O_2, S)=return(O_2O_1, S).*

It is obvious that if two operations are parallel then they are commutative. In this paper, for simplicity, we schedule parallel operations and commutative operations in the same manner (in fact, we can exploit higher concurrency from parallelity than from commutativity). We call two operations to be **Conditionally Commutative,** if they must satisfy some conditions to commute. For example, for database state $S = \{x = 1, \cdots\}$, there is an integrity constraint: $x > 0$. Two operations, O_1: $x = x + c$, $O_2 : x = x - d$. O_1 and O_2 are commutative if the condition $1 > d$ is true, otherwise they do not commute. By conditional commutativity, the concurrency control mechanism can exploit more semantic information for concurrency. In OODBS, there are many such semantic data in operations and objects[4,5]. Therefore, how to exploit them is important for the efficiency of a system.

Definition 3. *Two operations are conflict iff they are not commutative.*

4 Locking Techniques

4.1 Class Hierarchy Locking

From the operational semantics in OODB, we know that if an operation operates on class C, then it may also operate on the subclasses of C. The **intention lock** mode and simple granularity locking protocol are satisfactory, if only single inheritance relationships are permitted. But this does not work for a class hierarchy in which a class may have more than one superclass. Fig.1 is a multiple inheritance class hierarchy. Class C has two superclasses C_1, C_2, and one subclass SC. C_1 and C_2 have superclasses SC_1 and SC_2 respectively. Suppose a transaction T_1 sets a readlock on C_1 and an intention lock on SC_1, implicitly locking subclasses C and SC in readlock mode. Now another transaction T_2 sets a writelock on C_2 and an intention lock on SC_2. T_2 will also lock class C and SC in writelock mode implicitly. The conflict on class C and SC between T_1 and T_2 is not detected. This means that setting intention lock on the superclasses of the class being locked is not sufficient to detect all conflicting requests from different transactions.

To solve the above problems and make locking mechanisms more efficient, for every class C, we maintain a list L that records all C's subclasses that have more than one superclass.

We now give the class hierarchy locking protocol.

(1) Lock class C.

(2) Lock those of C's superclasses along any one superclass chain in intention lock mode.

(3) Find the set S_i of C's subclasses with more than one superclass through L, for every $e \in S_i$, lock e.

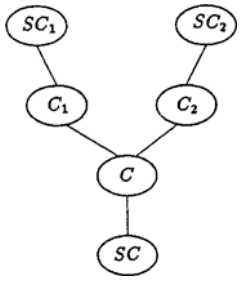

Fig.1

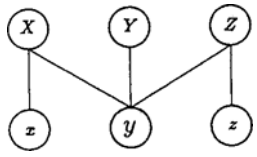

Fig.2

4.2 Complex Object Locking

In OODB, the unit of access to a database is an object, that is, the smallest units of locking are objects. Objects are usually complex, i.e. they can be constructed from other objects. Moreover, these objects may also be shared by several different objects. So, object locking is a complicated task. The general techniques of complex object locking have been studied in [7,10]. In this paper, in order to exploit structural semantics of objects, we use referential semantics to improve concurrency in addition to using commutativity property between operations.

Now let's see the impacts of complex object locking. X, Y, and Z are objects constructed from subobjects x, y, z as shown in Fig.2. Object y is shared by X, Y, Z. Transactions are

T_1 : $O_1(X)O_2(Z)$

T_2 : $O_3(Y)$

Suppose one schedule S for T_1 and T_2 is

S : $O_1(X)O_3(Y)O_2(Z)$

If we only lock X, Y, and Z, but do not lock their subobjects x, y, z, then it seems that T_1 and T_2 do not conflict, so, S is permitted. In fact, the high level operations $O_1(X), O_2(Z)$, and $O_3(Y)$ must be converted into suboperations on x, y and z:

$O_1(X) \Longrightarrow o_1(x)o_1(y)$

$O_2(Z) \Longrightarrow o_2(y)o_2(z)$

$O_3(Y) \Longrightarrow o_3(y)$

So, the schedule S becomes S'.

S' : $o_1(x)o_1(y)o_3(y)o_2(y)o_2(z)$

If both operation pairs $(o_1(y), o_3(y))$ and $(o_3(y), o_2(y))$ do not commute, then S' is not serializable. Hence S is not correct.

The problem here is that the conflicts at subobjects are ignored! So, when locking complex objects, not only should we consider the objects, but also their subobjects. If, in the above example, the referential type of Y to y is exclusive, that is, y cannot be referenced by X and Z, then S must be serializable.

In order to lock complex objects correctly and efficiently, we may use the referential semantics in the locking protocol. We stipulate that if object X references object Y exclusively, then, for every object Z referenced by Y, the referential type of Y to Z is also exclusive.

Now we give the complex object locking protocol.

(1) Lock object X.

(2) If X references object Y in shared type, then lock Y.

By this complex object locking protocol, it is unnecessary to lock the subobjects that are referenced exclusively.

5 Operations and Lock Modes

Operations in classes are the units of scheduling by concurrency control mechanism in FOODB. For every operation, there is a corresponding lock mode. When an operation begins to operate on object X, the concurrency control mechanism should lock X in its corresponding lock mode before it is executed.

All operations in FOODB are classified as three types.

1) Schema operation

This type of operations includes querying or updating class hierarchy.

(1) query the structure of a class

(2) create or drop a class

(3) add (drop) an attribute to (from) a class

(4) add (drop) an inheritance relationship to (from) a class

(5) add (drop) a method to (from) a class

2) Statistical operation

This type of operations includes querying or updating a set of instances that satisfy some given conditions.

(1) query a set of instances that satisfy a given condition

(2) update a set of instances that satisfy a given condition

(3) delete a set of instances that satisfy a given condition

3) Instance operation

(1) find an instance from a class

(2) update an instance in a class

(3) insert an instance to a class

(4) delete an instance from a class

The lock modes are designed as follows.

1) Class lock

(1) c-ql: class query lock

(2) c-ul: class update lock

2) Instance lock

(1) i-sl: instance set query lock

(2) i-ql(ob): instance query lock

(3) i-ul(ob): instance update lock
(4) i-il(ob): instance insert lock
(5) i-dl(ob): instance delete lock

The i-ql, i-ul, i-il and i-dl have a parameter ob indicating the Oid of the object being locked.

Lock Mode	Operations
c-ql	schema operation(1)
c-ul	schema operation(2)—(5), statistical operation(2)—(3)
i-sl	statistical operation(1)
i-ql(ob)	instance operation(1)
i-ul(ob)	instance operation(2)
i-il(ob)	instance operation(3)
i-dl(ob)	instance operation(4)

Fig.3

	c-ql	c-ul	i-sl	i-ql(ob′)	i-ul(ob′)	i-il(ob′)	i-dl(ob′)
c-ql	y	n	y	y	y	y	y
c-ul	n	n	n	n	n	n	n
i-sl	y	n	y	y	n	n	n
i-ql(ob)	y	n	y	y	c	c	c
i-ul(ob)	y	n	n	c	c	c	c
i-il(ob)	y	n	n	c	c	y	c
i-dl(ob)	y	n	n	c	c	c	c

ob=ob′: c="n", ob $\neq$ ob′: c="y"

Fig.4

Every operation has a corresponding lock mode. The operations and their corresponding lock modes are described in Fig.3.

In order to reduce the number of locks set on objects to one at a time, statistical operations (2) and (3) use the same lock mode as schema evolution operations. To get a higher degree of concurrency, we design an i-sl lock mode for statistical operation (1).

By the semantics of operations and commutativity between them, we summarize the compatibility matrix of lock modes as Fig.4.

If two lock modes are compatible, then their corresponding operations can be executed concurrently. The "c" in matrix means that two lock modes are conditionally compatible, that is, their corresponding operations are conditionally commutative. For example, if two deletion operations delete two different objects then they can be executed concurrently.

If an object, which is inserted or deleted by one transaction T_1, may also be operated by another transaction T_2, then we say that the object is a **Phantom** for T_2, otherwise it is not a phantom object for T_2. In Orion, phantom problems

were not considered, whereas in [3], in order to avoid phantoms, insert or delete operation conflicted with all operations. By our scheme, phantoms are avoided, and the operation pairs (insert, insert), (insert, delete) and (delete, delete) can be executed in parallel or conditionally parallel.

6 Design and Implementation of Concurrency Control Mechanism — Object-Locking

In this section, we design and implement the concurrency control mechanism — Object-Locking on the basis of above discussions.

Fig.5 is the description of Object-Locking. It is a two-phase locking procedure, that is, once a transaction releases a lock, it never obtains locks again. Only when transaction's commit or abort operations are received, can the transaction's locks be released. Since deadlock detecting procedure is used, Object-Locking is deadlockfree.

Object-Locking
Input: transaction's operation O_i
Output: decision of whether O_i can be executed or not
Method: **Do Case**

Case $O_i \neq$ **commit and** $O_i \neq$**abort**
lock the object in a proper lock mode depending on the type of O_i
if locking succeeds then return(execute O_i)
else
begin
  put O_i in wait-queue
  call WFG-adjust /*adjust WFG */
  call deadlock-detect
  if there is a circle in WFG then
  begin
    call transaction abort(T_i)
    call WFG-adjust
    awake some transactions in wait-queue
    return
  end
end
suspend T_i until T_i is awaked
Case O_i**=abort**
call transaction abort(T_i)
call WFG-adjust
awake some transactions in wait queue
Case O_i**=commit**
call buffer-write(T_i)
call WFG-adjust
call lock-release(T_i)
awake some transactions in wait queue
EndCase

Fig.5

An operation O_i on a given object conflicts if it is not commutative with other operations executed on this object by still active transactions, i.e. those transactions that have not been committed or aborted. When an operation conflicts, the transaction requests that the operation be blocked, and deadlock detection needs to be initiated.

The process of checking for deadlock is achieved by using a wait-for-graph. When a transaction issues a request to execute an operation, the concurrency control mechanism, by using the compatibility matrix and locking process, determines whether the operation request conflicts or not. If the request conflicts, the transaction is made to wait. The corresponding wait-for edges are added and a circle detection algorithm is initiated. If a circle is found, the transaction making the request is aborted. When a transaction terminates successfully or unsuccessfully, the node that corresponds to the terminating transaction together with the edges associated with the node is removed from the WFG. The results of the transaction are written back to database or discarded simply. All locks used by the transaction are released.

7 Discussion

(1) High level operations as scheduling units

More and more researchers believe that high level operations as scheduling units of concurrency control are preferable in OODB or other engineering environments[4−8]. Because rich semantics of the operations can be exploited to improve concurrency, the concurrency control mechanism can be made more efficient. In this paper, there are three kinds of operations: schema operations, statistical operations, and instance operations. Using the semantics of these operations, we can schedule them in a more appropriate way. For example , two insert operations can always be executed concurrently, deletion and insertion operations can be executed conditionally concurrently, etc. Moreover, the phantoms can be described clearly and avoided rationally, whereas the simple lower level operation-based (read, write) transaction model cannot achieve this so clearly and conveniently.

(2) Recovery

The high level operation- based transaction model necessitates further researches in recovery methods[8,9]. Conventional recovery techniques based on log, for example, are not adequate for the new transaction model. When simply installing the database, the "before image" of update done by aborted transaction is not correct sometimes. In [8], the authors defined the notion of "strictness" for histories containing high level operations, and gave a recovery rule by simply executing their "inverse" operations. In [9], the authors developed a unified transaction model including both data operations and termination operations (commit, abort), which allows reasoning about the correctness of concurrency control and recovery within the same framework. The future important work is how to embed these techniques to real OODB systems.

8 Conclusion

We have developed a practical concurrency control mechanism in OODBMS and given the transaction model based on three kinds of high level operations: schema operations, statistical operations, instance operations. By the properties of parallelity and commutativity between these operations, the proper lock mode for each operation has been designed and the compatibility matrix is constructed. With these lock modes, phantoms are prevented and a high degree of concurrency is achieved.

References

[1] Bernstein P A, Hazilacos V, Goodman N. Concurrency Control and Recovery in Database System. Addison-Wesley, Reading, MA, 1987.

[2] Won Kim. Introduction to Object-Oriented Database. MIT Press, Cambridge, 1990.

[3] Cart M, Feeri J. Integrating Concurrency Control. In *Building an Object-Oriented Database/The Story of O2,* Bancilhon F, Claude Delobel, Paris Kanellakis, (eds.), Morgan Kaufmann Publishers, San Mateo, California, 1992.

[4] Panos K Chrysanthis *et al.* Extracting concurrency from objects: A methodology. ACM. PODS 1991.

[5] Badrinath B, Ramamritham K. Semantics-based concurrency control: Beyond commutativity. ACM PODS, 1992.

[6] Badrinath B, Ramamritham K. Performance evaluation of semantics-based multilevel concurrency control protocols. In *Proc. of ACM SIGMOD,* 1990.

[7] Weihl W. Commutativity-based concurrency control for abstract data types. *IEEE Trans. on Computers,* Dec. 1988.

[8] Rajeev Rastogi, Korth H F, Avi Silberschatz. Strict histories in object-based database system. ACM PODS, 1993.

[9] Hans-Jorg Schek, Gerhard Weikum, Haiyan Ye. Towards a unified theory of concurrency control and recovery. ACM PODS, 1993.

[10] Jorge F Garza, Won Kim. Transaction management in object oriented database system. In *Proc. of ACM SIGMOD,* 1988.

Qu Yunyao is an Associate Professor in Department of Computer Science, Fudan University. His current research interests include transaction processing and distributed Database.

Tian Zengping is a Ph.D. candidate in Department of Computer Science, Fudan University. His current research interests include in Multimedium Database.

Wang Yujun is a Ph.D. candidate in Department of Computer Science, Fudan University. His current research interests are data model and query language.

Shi Baile is a Professor in Department of Computer Science, Fudan University. His current research interests include database theory and application, software engineering.