Concurrency Control of Persistent Object Transactions in CORAL

Chandrasekaran Venkatapathy[1] and **Barrett R. Bryant**[2]
Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Boulevard
Birmingham, Alabama 35294
U. S. A.
{venkatap, bryant}@cis.uab.edu

Daniel T. Chang
Data Base Technology Institute
IBM Santa Teresa Laboratory
555 Bailey Avenue
San Jose, California 95161
U. S. A.
dtchang@vnet.ibm.com

Abstract

CORAL is an object-oriented database programming language which supports management of both relational database objects and persistent complex objects. This paper defines the notion of a persistent object transaction in CORAL and introduces a scheme for controlling concurrent access to objects being shared by multiple applications. The scheme presented here is a combination of pessimistic and optimistic approaches and may be adapted to the semantic constraints of the application. The result is the development of CORAL for transaction management across multiple database applications.

Keywords: Concurrency Control, Database Programming Languages, Object-Oriented Databases, Object Persistence

1 Introduction

Database applications are often limited to what the database query language has to offer. Applications are developed in most cases by interleaving a query language with a high-level language or invoking the query language by a remote procedure call. Such approaches require the developer to be familiar with more than one environment. Database programming languages serve to integrate the power of high-level programming languages with database management capabilities. One such database programming language is CORAL, the Concurrent Object-oRiented Application Language, which was developed at IBM [Chan90]. CORAL provides support for managing large amounts of persistent objects which may be in the form of complex or relational database objects [Brya93]. Furthermore, these persistent objects may be shared among many different applications and concurrently accessed by these applications. That is, data access and manipulation may take place concurrently. The present implementation of CORAL does not provide for transparent concurrency control as the programmer must explicitly lock and unlock shared data. Without such concurrency control, there may be a failure to maintain data consistency and integrity.

In this paper, we present a flexible mechanism for controlling concurrent access to persistent objects in CORAL. Our objective is to manage concurrent transactions with minimal effect upon the throughput or degree of concurrency. Toward this end we use a variation of two-phase locking

[1]Author's Present Address: Cincinnati Bell Information Systems, Inc., 851 Trafalgar Court, Maitland, Florida 32751, U. S. A., chandra/p2k.sda@cbis.com

[2]This research was carried out in part while the author was a Visiting Scientist at the Data Base Technology Institute, IBM Santa Teresa Laboratory, San Jose, California, U. S. A.

which may be adjusted to the semantic constraints of the objects being defined. The general approach is pessimistic in that we expect there to be conflicting requests for objects and these requests are resolved by producing a serializable schedule. However, we also support optimistic concurrency control as well through transaction roll backs. The exact method to be used may be application dependent.

The paper is organized as follows. Section 2 presents an overview of CORAL and defines the nature of a transaction. In section 3, we review relevant concurrency control methods, present our method for providing concurrency control of CORAL objects along with an example of the method and details of the implementation. We conclude in section 4 with a discussion.

2 CORAL

CORAL is a general purpose programming language which provides support for the development of sequential, parallel and distributed applications in a transparent manner. It permits the integration of persistent complex objects and relational database objects into a unified framework. This allows database applications to be developed using persistent objects, which can be executed in a distributed environment. It allows a program to manipulate transient and persistent objects in a uniform manner. Like any other object-oriented programming system, CORAL provides support for encapsulation, inheritance, and polymorphism. Some of the unique features that CORAL provides include support for concurrency among objects and interfaces to conventional languages such as FORTRAN and C [Chan92]. Syntactically, CORAL is compatible with the Common Object Request Broker Architecture Interface Definition Language (CORBA IDL) [OMG91].

CORAL supports three kinds of classes, namely persistent classes, relational database classes, and transient classes, which is the default class. A class may be defined with generic types which can be used for attributes, methods, parameters, or local variables. Objects, which are instantiations of classes, are in encapsulated units and they can communicate with each other through message passing. Because of encapsulation, objects can be considered as individual computational units and this facilitates them to be executed in a parallel and distributed environment.

Typically, a CORAL program consists of class definitions and objects instantiations. A database application may be developed by defining objects to be persistent. The state data of persistent objects is stored in a file or relational database. On invocation, the state data from these data stores are loaded into the relevant data structures. Destruction of an object will result in deletion of its state data from the data store. Concurrency is assumed to be transparent to the user, although the support has not been provided in the existing implementation. The aim of this work is to provide a concurrency control scheme for concurrent access to objects transparently.

First we must define the notion of a CORAL transaction. In CORAL, persistent objects are manipulated through the methods defined for the class to which the object belongs. Each such method defines a transaction. If a method contains calls to other methods which manipulate persistent objects, then these are subtransactions. These subtransactions may have other subtransactions and so on. This is essentially a nested transaction model [Moss85]. In a CORAL application, the top-level transactions may be very long with many different persistent objects being accessed.

An example of a CORAL program is given in Figure 1. This example defines a Student class which is persistent. The class has three attributes, name, id_number and grades, the latter con-

```
class Registration {
    attribute string course;
    attribute char grade;
};

class Student
    - persistent
{
    attribute string name;
    attribute int id_number;
    attribute registration grades[5];
    void read() {...};
    void write() {...};
};

Object Transactions
{
    Student S1, S2, S3, S4, S5, S6;

    void Init() {...};

    void T1()
    {
        S1 . read();
        S2 . read();
        S3 . read();
        S4 . read();
        S5 . read();
    };

    void T2()
    {
        S2 . write();
        S6 . read();
    };
}
```

Figure 1: Example of CORAL Class and Object Declarations

structed using the Registration class (i.e., grades is an array of 5 Registration objects). There are two generic methods read and write which respectively read and write objects which are instances of this class. In practice, the many different methods belonging to a persistent class would be categorized as either read or write methods by the CORAL compiler. If we assume that these methods do not call any other methods, then they are the smallest grain of transaction in the system (i.e. they have no subtransactions). The object Transactions defines six Student objects and three methods which manipulate these objects. We assume that Init assigns initial values to the six Student objects. T1 and T2 are the two transactions being defined for these objects and these transactions have subtransactions read and write defined earlier. It is possible for additional transactions to have T1 and T2 as subtransactions and so on. Note that the only shared object among the transactions is S2. We will return to this example in our discussion of the concurrency control algorithm.

3 Concurrency Control

Fundamental approaches used by all concurrency control schemes can be broadly categorized as either pessimistic or optimistic. In a pessimistic approach, conflicts between concurrent transactions are identified during a transaction's execution and resolved by imposing a delay on some transactions [Cell88]. In an optimistic approach [Garc87], conflicts are resolved by aborting and restarting some transactions at a later time. Some of the well known pessimistic methods are two-phase locking, multi-version time-stamps and hybrids of these approaches [Barg91]. Multi-version time-stamps can also be used for optimistic concurrency control. A proof theory for concurrency control mechanisms applied to the nested transaction model is discussed in [Beer89].

Most of the commercial database management systems use two-phase locking (2PL) for synchronizing database accesses. Many optimistic concurrency control schemes have been proposed in the literature [Thom90] and some of them have been incorporated in prototypes. Optimistic concurrency control scheme requires more support from the system. In order to make a decision in real-time or for proper validation in a high collision environment, there may be a need for a server/background task running on the system to monitor the transactions and take appropriate actions in order to increase the degree of concurrency [Atki92]. In our case, no assumption is made on the system which executes the language. So, the language (or the application) should take care of concurrency control schemes and its effects. Lock based protocols are found suitable for such independent development systems [Naho91]. In this paper, our primary approach is to use a locking strategy as it has been found more reliable in a commercial environment. Also, a scheme to migrate to optimistic concurrency control is presented.

3.1 Objects and Events

An object is an encapsulation of data and each object has operations that define a set of messages which the object may receive. The only way of communicating with the objects is through these messages. A concurrency control mechanism is needed whenever there is conflict between (among) transactions. A transaction in this case is a sequence of messages that are sent to the objects. It transforms the objects (database) from one consistent state to another. We assume that all transactions terminate, irrespective of their outcome. A transaction can either commit or abort depending on its state. Consistency violation is solved by allowing only consistency-preserving

schedules of concurrent transactions [Kim89, Ozsu91]. We assume transactions are consistent with serializable execution. The unit of database granularity that we consider here is an object that is persistent. The level of concurrency allowed by a protocol depends not only on the object and operation semantics but also on the type of conflict and transaction [Gray93]. The user should be able to choose an appropriate protocol for optimal throughput of the transaction, based on the type and nature (a priori information) of the transaction. Ideally, users would like to have such options for development of database applications.

3.2 Transaction Mechanism

The scheme that is used here is a lock based protocol. Two-phase locking (2PL) protocols, which are widely used in commercial databases [Naho91], do not require much knowledge about the particulars of the application. However, if it is a long transaction, resources are blocked until the transaction is completed. The performance of a 2PL mechanism is unacceptable for these applications. In such cases, in order to provide more concurrency, the semantic information about transactions and operations can be used. For example, in a long transaction if an object is no longer needed by it, then it could be released to other transactions. However, the transactions that are using these objects should satisfy certain requirements. Such a scheme is called *altruistic locking* [Barg91] and it is an extension of 2PL locking. The additional information that one extracts from the transactions are the access information, which can be negative access information and positive access information. Both together describe the objects that transactions release when there is a conflict.

A transaction may be in one of two states. First is the *acquiring state*, where the transaction tree is formed, semantic information is extracted and locks are acquired. Next is the *commit state*, in which the transactions are actually executed.

We categorize transactions into two types, namely, *master transactions* and *balloon transactions.* A transaction is said to be a master transaction if it is currently under execution and in whose context other transactions are validated. It controls the events of the transactions and in case of conflict, provides access information to other transactions. Transactions which are other than a master transaction are said to be balloon transactions. At any point of time, in our system, only one transaction is a master transaction. When a master transaction completes execution, either it dies out or make a balloon transaction as the master. The basic unit of resolving contention is a lock. Locks can be acquired in either READ or WRITE mode.

3.3 Balloon Transactions

A balloon transaction is a transaction that is in conflict with the ongoing master transaction. The name balloon has been given because this information would be floated so that it is made available to all the transactions. Even if some other transaction comes up at a later point of time, it would be able to see a balloon transaction, which is an indication of conflict. We say a balloon is *floated* to indicate such an event.

To start with, assume the initial transaction would be chosen to become the master transaction, hereafter denoted as MT. When two transactions come up simultaneously, we can resolve the contention and choose MT using a *time-stamp ordering* scheme [Cell88], in which we have as-

signed a time-stamp to each transaction at the time of execution (e.g. using the system clock). This ordering would ensure that there would not be any deadlocks as transactions are serialized. Contentions of this nature would never happen afterwards because there would always be a master transaction running. As transactions are being serialized this way, deadlocks would never occur. The transaction acquires locks for the data elements it is going to access in READ/WRITE mode. The transaction tree is formed and necessary semantic information is stored. While this transaction is going on and if another transaction (say T2) comes up, transaction T2 acquires locks. If there is no conflict, it proceeds and finishes. However, while acquiring locks serially, if there is a conflict, immediately it sends up a balloon. A balloon is an indication that a conflict has taken place. In order for the balloon transaction, hereafter denoted BT, to proceed, it has to validate itself with MT. In this validation, BT is requesting objects that are presently locked by MT. There will typically be different objects for each BT and some common objects as well. Once MT has finished processing an object, it will release its lock with respect to a particular BT that is requesting that object, since not all BT's are requesting that object. If the MT is in the commit state, then it sees the balloon transaction and validates with the semantic information it has already collected. Upon validation, if MT sees that an object can be freed for the balloon, it provides the balloon transaction with the access permission for the lock. The balloon transaction proceeds concurrently. If there is one more transaction, it also becomes a balloon transaction. Now, while validating, it has to validate not only with MT, but also with the other balloons which have been successful in acquiring permission. In this manner, many balloons can be floated successfully. After MT completes execution, it clears the locks it has acquired and transfers control to the next available unexecuted balloon.

The following postulates govern the overall execution of the transactions.

Postulate 1

At any point of time, only one transaction can be a master.

Postulate 2

All transactions other than the master transaction, which are in conflict with it are balloon transactions. A BT has to validate itself with the master transaction and the earlier created unexecuted balloon transactions successfully. The validation is done in the order of creation of balloons.

3.4 Concurrency Control Algorithm

The above mechanism is formalized in the algorithm given in Figure 2. This algorithm may be illustrated with the CORAL example given earlier in Figure 1. Let us follow the method suggested on transaction T1 which accesses objects S1, S2, S3, S4, S5 all in READ mode and transaction T2 which accesses S2 in WRITE mode and S6 in READ mode. Figure 3 shows a schedule for the transactions. In this schedule, it is T1 which comes up first. Stage 1 (steps 1-21), the *acquiring state*, begins starting from the start of the transaction and proceeds until the end of the transaction is reached. First, the transaction is tested for being the master transaction by the call **Am_I_the_master()**. Steps 1-4 return T1 as the master. Step 5 verifies that we are indeed at the beginning of the transaction. In steps 6-8, the transaction is repeatedly scanned and the transaction tree is formed. (In an actual implementation, this would be done only once and these two steps would represent a progression through the execution of the transaction.) At each iteration of the process, the function **Extract_information** in step 9 extracts positive access and negative access

```
/* Stage 1 */

 1. if (Am_I_the_master())
 2.     I_am_the_master = TRUE;
 3. else
 4.      I_am_the_balloon = TRUE;
 5. Check_for_the_start_of_transaction();
 6. while(not end_of_transaction) {
 7.      Scan_the_transaction();
 8.      Form_transaction_tree();
 9.      Extract_information();
10.      Acquire_locks();
11.      if (conflict_while_acquiring)
12.          if (I_am_the_balloon) {
13.              Float_a_balloon();
14.              Keep_trying_for_lock_permission();
15.              if (allowed_by_master_transaction)
16.                  Execute_the_Transaction();
17.          }
18.          if (I_am_the_Master)
19.              Keep_trying_for_lock_permission();
20.      }
21. }

/* Stage 2 */

22. while (not end_of_transaction) {
23.      Execute_the_transaction();
24.      if (balloon_is_seen_floating) {
25.          validate_MT_with_BT();
26.          if (successful_validation)
27.              provide_lock_access_permission_to_BT();
28.          else
29.              Keep_the_BT_waiting();
30.      }
31. }
```

Figure 2: Concurrency Control Algorithm

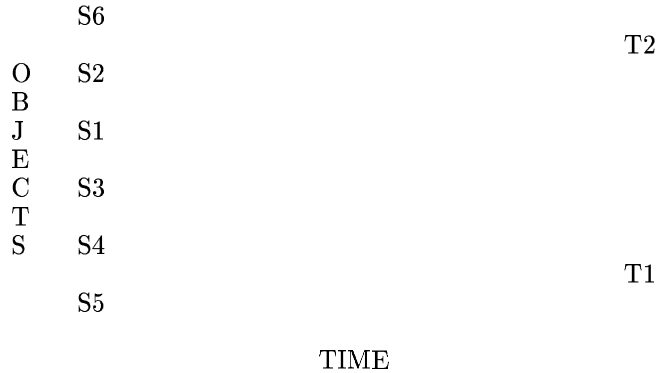

Figure 3: Access Pattern of Transactions T1 and T2

information about the objects, which would be used for validation. Next locks are acquired for the objects (step 10). As it is the master transaction, locks can generally be acquired without any conflict. A conflict might arise in two situations: 1) a balloon transaction is executing and needs an object currently held by another transaction, and 2) a master transaction is executing and needs an object currently held by a balloon transaction. In the first case, we float a balloon until the balloon is handled by the master transaction and the lock granted (steps 12-17). The second case will only arise if a previous master transaction completed before the balloon transaction causing the conflict. Otherwise, the current MT would have control over the object in question. Note that there can be only one master transaction. In such cases, the transaction waits until the lock is released by the BT holding it (steps 18-19). Continuing with the example, T1 next proceeds to stage 2 (steps 22-31), which is nothing but the actual execution of the transaction.

Correspondingly, transaction T2 would be attempting to execute stage 1 and acquire the lock for S2, which has been locked by the master transaction T1. To resolve this conflict, T2 floats a balloon using the **Float_a_balloon** operation (step 13). The master transaction T1, which is already under execution (step 23), checks for the balloon in step 24 and validates against the balloon transaction using the function **Validate_MT_with_BT()** (step 25). In this case, object S2 will not be accessed again by T1, once it has been accessed the first time, so the validation is successful (step 26). The master transaction T1 then provides access permission for S2 by **Provide_lock_access_permission_to_BT()** (step 27), so that the balloon transaction T2 can proceed. Hence, both the transactions can be executed concurrently. Suppose that the validation proved unsuccessful as a result of T1 still needing access to S2. In that case, the balloon transaction would be kept waiting (step 29) until the requested lock is released.

The above scheme can be incorporated into the language in a transparent manner. Transactions are interleaved with the language in the form of calls to methods which manipulate persistent objects. At the time of compilation, necessary support has to be provided for additional information in the form of transaction tree, balloon table and lock table. The compiler has to provide the feature for creation/updating of these tables. A prototype of the above model has been developed in the form of an interpreter. Here, the execution flow of a transaction consists of two stages. The first stage forms the transaction tree, acquires locks, and executes the transaction if such execution is allowed by the master transaction. The second stage will execute the actual transaction if this was not done in the first stage. The beginning of the transaction initiates the first stage. While

acquiring locks, based on the type of transaction, whether MT or BT, the transaction proceeds. There are certain data structures which are visible across all sites in which the persistent objects are stored. Data structures related to the transaction tree and the balloon table should be visible to all active sites.

4 Discussion

The approach described in this paper serves to provide a transparent concurrency control mechanism for CORAL. Depending on the application being developed, a choice of transaction management schemes are available - pessimistic or optimistic. We believe that this work has enhanced the capability of CORAL as a database programming language but may be applied to other object-oriented database management systems as well.

The amount of semantic information that we extract to handle concurrent transactions is proportional to the degree of concurrency. Also, the complexity of operations increase with the degree of concurrency. As the application is developed by the user, it may be a reasonable assumption to know the type of application being developed (a priori information). This information, though not used by the concurrency mechanism, helps in scheduling other transactions. Average transaction time could be one such parameter that can be passed as a compilation option. This can be illustrated with an example. Suppose there is a long transaction (LT) being executed. A very short, critical, important transaction, which has a data conflict, comes up. According to the earlier scheme, the short transaction becomes a BT and keeps waiting until the other transaction completes. If the short transaction recognizes that the master transaction is a very long transaction from the parameter supplied by the user, then it can kill MT and it becomes the master. In other words, we are prioritizing transactions. Because of an unexpected aborted transaction, we have to roll back. That is all the changes that have been introduced are undone and the database is returned to the state that existed before the transaction began. So, before the killed MT gives control to the next transaction, it rolls back, remove locks and keeps waiting at the start of the transaction. After the short transaction completes, LT is once again restarted. This may have the effect of allowing many short transactions to execute before the long transaction but as these are given higher priority by the user, this behavior is appropriate. Furthermore, since a long transaction typically consists of a set of subtransactions, it may be the case that not all component subtransactions need to be rolled back. Therefore, the LT will continue to make progress toward completion. Finally, a transaction may take a suitable status based on the status of the on-going transactions and lock tables. If system calls like **get_lock_table_status()**, **get_type_of_transaction()**, etc., are supported by the language, it would help in providing a higher degree of concurrency.

As we provide a data structure for the roll-back mechanism, we can provide an optimistic concurrency control scheme (OCC). The major difference between this scheme and the earlier scheme is the validation phase. If the user, with certainty, knows that he is going to operate in an environment with less collisions, he can choose OCC. From the language implementation, we have all the necessary data structures created. Only the validation scheme would be done at the last. In case of collision, roll-back would take place. The roll-back data can either be stored in the symbol table or in the transaction tree. More details about OCC schemes can be found in [Cell88, Garc87]. Choosing of a scheme can be passed as a compilation parameter.

Our implementation currently exists only in prototype form as an interpreter. The next step is

to integrate the approach into a compiler for CORAL to generate the code for concurrency control at compile-time. Once this is done, we plan to evaluate performance of the transactions under the given methods, including a formal comparison with other optimistic and pessimistic approaches to concurrency control. Additional methods may also be developed for improving the performance where possible. At present we believe we have succeeded in providing a general concurrency control mechanism for CORAL which allows efficient concurrent access to persistent objects without the danger of losing data integrity.

References

[Atki92] Atkins, M. S. and Coady, M. Y., "Adaptable Concurrency Control for Atomic Data Types," *ACM Transactions on Computer Systems 10*, 3 (August 1992), 190-225.

[Barg91] Barghouti, N. S. and Kaiser, G. E., "Concurrency Control in Advanced Database Applications," *ACM Computing Surveys 23*, 3 (September 1991), 269-317.

[Beer89] Beeri, C., Bernstein, P. A., and Goodman, N., "A Model for Concurrency in Nested Transaction Systems," *Journal of the ACM 36*, 2 (April 1989), 230-269.

[Brya93] Bryant, B. R., Chang, D. T., and Lee, T., "Integration of Persistent and Relational Database Objects in CORAL," *Proceedings of VIII Brazilian Symposium on Databases*, 1993, pp. 344-356.

[Cell88] Cellary, W., Gelenbe, E., and Morzy, T.,*Concurrency Control in Distributed Database Systems*, Elsevier Science Publishers, New York, 1988.

[Chan90] Chang, D. T., "CORAL: A Concurrent Object-Oriented System for Constructing and Executing Sequential, Parallel and Distributed Applications," *Proceedings of the 1990 ECOOP-OOPSLA Workshop on Object-Based Concurrent Programming*, 1990, pp. 26-30.

[Chan92] Chang, D. T., "Productivity through Object-Oriented Programming: The CORAL Approach," *Proceedings of the 1992 IBM Software Engineering Intertechnical Liaison Conference*, 1992.

[Garc87] Garcia-Molina, H. and Salem, K., "Sagas," *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, 1987, pp. 249-259.

[Gray93] Gray, J. and Reuter, A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, San Manteo, CA, 1993.

[Kim89] Kim, W. and Lochovsky, F. H., eds., *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, Reading, MA, 1989.

[Moss85] Moss, J. E. B., *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, Cambridge, MA, 1985.

[Naho91] Nahouraii, E. and Petry, F., eds., *Object-Oriented Databases*, IEEE Computer Society Press, Los Alamitos, CA, 1991.

[OMG91] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 1.1, OMG TC Document 91.12.1, December 6, 1991.

[Ozsu91] Ozsu, M. T. and Valduriez, P., *Principles of Distributed Database Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Thom90] Thomasian, A. and Rahm, E., "A New Distributed Optimistic Concurrency Control Method and a Comparison of its Performance with Two-Phase Locking," *Proceedings of 10th International Conference on Distributed Computing Systems*, 1990, pp. 294-301.