Achieving High Concurrency In Object Oriented Databases

Robert Bretl
GemStone Systems, Inc.
1260 N.W. Waterhouse Ave., Suite 200
Beaverton OR 97006
bretlb@gemstone.com

Introduction

Today object oriented databases (OODB) are being used in large scale applications in a variety of industries including telecommunications, banking, manufacturing, insurance, and shipping. These applications are characterized by having complex data, that is, data which is represented in the highly interconnected graphs of an object model.

Object databases are very good at storing the complex data model, but it is generally up to the application developer to figure out how to scale the application so that it runs efficiently with many concurrent users. The purpose of this paper is to explore various techniques that can be used in object oriented databases to achieve high concurrency.

This paper first provides background information about transactions and several concurrency control techniques, then the difference between physical consistency and logical consistency are explored. Finally, several specialized container classes are introduced to show how logical (behavioral) consistency can be implemented and how they can be used to improve concurrency and simplify application development.

Transactions

Any discussion of transactions begins with an understanding of their ACID properties:

- **Atomicity.** A transaction allows for the grouping of one or more changes to objects in a database to form an atomic or indivisible operation. That is, either all of the changes occur or none of them do. If for any reason a transaction cannot be completed, everything this transaction changed can be restored to the state it was in prior to the start of the transaction with an abort or rollback operation.
- **Consistency.** Transactions always operate on a consistent view of the database and when they end always leave the database in a consistent state. While changes are being made to a database, inconsistent state that occurs during the process of performing updates are hidden so that whether the transaction commits or aborts the database is always left in a consistent state.

- **Isolation.** To a given transaction, it should appear as though it is running all by itself on the database. The effects of other concurrent transactions are invisible to this transaction, and the effects of this transaction are invisible to others until the transaction is committed.
- **Durability.** Once a transaction is committed, its effects are guaranteed to persist even in the event of subsequent system failures. Until the transaction commits, not only are changes made by that transaction not durable, but are guaranteed not to persist in the face of a system failure.

Transactions provide a programming model for databases that define a starting point, indicated by a “transactionBegin” command, followed by a sequence of operations against the database, and terminated with either a “transactionCommit” or “transactionAbort” command.

A transactional model that supports all of the ACID properties provides a simplified framework in which a programmer has a clear understanding of what to expect from the behavior of the system.

Concurrency Control

Concurrency control is the technique used to maintain the consistency and isolation properties of transactions and is required when two concurrent transactions try to simultaneously perform read or write operations on the same objects. Read consistency is defined as requiring that all reads in a transaction are performed against the same state of the database, while write (update) consistency guarantees that the order of the operations on objects in the database doesn’t effect the final outcome.

Databases may implement various forms of consistency support and concurrency control mechanisms. To insure read consistency two basic mechanisms are typically used:

- **Guaranteed view –** all transactions are guaranteed a consistent view of the object repository based on the state of the database at the point in time that the transaction begins. This is a characteristic of the database and may not apply to all databases.

- **Read locks –** the programmer must acquire locks on objects as they are read to insure that other users cannot modify them.

If an application requires a consistent view for certain read-only transactions, such as generating reports, or gathering data to be presented on a customer display, then a database that supports a guaranteed view is generally better because:

- The application is easier to develop and maintain because the programmer doesn’t need to be concerned about managing the read locks and the possible deadlocks that can occur when interacting with transactions using write locks.

- The application runs more efficiently because databases that implement the guaranteed view usually provide very efficient read access to the data. This is possible because they do not need to consult a lock manager and are able to access the data by following a pointer to the specific view.
- A higher level of concurrency (thus an increase in overall throughput) can be attained because there is no conflict between the concurrent readers and writers.

If an application is written in such a way that all of the update or write operations in concurrent transactions are on disjoint sets of objects, then no conflicts exist and the transactions can commit successfully without any additional concurrency control to slow down their performance. This however, is rarely the case and in most applications some form of concurrency control must be used to insure that transactions are able to commit successfully.

There are three basic techniques used to manage update concurrency in a database:

- **Last in Wins** – The last concurrent writer of the data "wins" in the sense that their changes are preserved in the database, but this can lead to unexpected results. For example: User A fetches the value of object O and sees the value 3. Then concurrent user B changes the value of object O to 7 and commits. Meanwhile User A adds 1 to the value of O (that it read) and stores the resulting value 4 into object O. The result is that User B's updates were lost. Most applications will not be able to use this technique.
- **Optimistic –** objects are accessed and updated in a private view as needed during the transaction. Before the transaction commits, the transaction must perform checking to determine if its operations are consistent with the operations of other transactions that committed since it started. If the consistency checks fail, the commit is not successful and the changes made during the transaction must be aborted. The updaters are "optimistic" in that they don't expect that there will be a conflict very often and in the case of failure they can try the update again. This is also referred to as "First in Wins" since the second attempt to commit an object can fail in the commit with transaction conflicts.
- **Pessimistic** – a lock or timestamp ordering is used to prevent other transactions from accessing or updating an object while the transaction is active. Two-phase locking is a frequently used because it is simple and avoids the problem of cascading aborts.

In many object oriented applications predicting which objects are going to be read or written can be tricky because the behavior of a method may depend on a value found. For example, a method may discover that a stock item has a low inventory and then invoke a reorder method. The reorder method may be very simple, but it may need to access a number of supplier objects and determine which is best in the current situation. This unpredictability of the behavior generally makes it more difficult to develop applications using only pessimistic concurrency control mechanisms because lock management becomes more difficult and the probability of deadlock is increased. Thus, in object oriented systems optimistic concurrency techniques are more frequently used.

One of the keys to the optimistic approach is the detection of conflicts. Consistency failures are usually classified as either Read or Write failures. Read consistency insures that data read during a transaction hasn't been changed by another transaction during the time the transaction was active (between the start of a transaction and when it is committed). These are referred to as Read/Write conflicts because the current transaction's reads conflict with the writes of other transactions. Write consistency insures that an object written during a transaction was not written by another transaction during the time the transaction was active. These conflicts are referred to as Write/Write conflicts because the current transaction's writes conflict with the writes of others.

Some databases can provide multiple levels of consistency enforcement:

- Full consistency checks require that all objects read or written by a transaction must not have been written by a concurrent transaction that committed while a transaction was active.
- Concurrency can be increased (commit failures reduced) by only performing write checks, but this should only be done when the application can tolerate the fact that its update may be made based on data that may have changed since it was read. Workflow applications like the one described in [Beck and Hartley, 1994] take advantage of this capability.

It is also important to consider the granularity of the consistency checking mechanism. Some systems may attempt to optimize the checking by performing checks on courser grained units such as the pages on which the objects reside. This can improve performance in cases where the objects are clustered so that objects that would cause conflicts are on different pages. While possible, this is often very cumbersome and in general finer grained consistency checking on individual objects will allow more concurrent updates to be processed.

Hybrid Concurrency

In general, fine grained locking or concurrency checks provide more concurrency. However, in object oriented database systems that support both optimistic and pessimistic mechanisms applications can be developed that take advantage of both. One approach is to identify the specific operations that must succeed (cannot suffer the cost of being aborted and retried) and to use "guardian objects" to control their access to a group of objects. Operations that must succeed then use the pessimistic features of the database system to acquire/release locks on the guardian objects, thereby insuring their success. One major advantage of using the "guardian objects" pattern is that the total number of objects that you need to get locks on is greatly reduced, which decreases the number of the locking combinations that need to be analyzed to prevent deadlocks. In general, throughput can be increased because only the transactions that need to lock these groups of objects are strictly serialized by the locking mechanism and the other optimistic transactions can proceed unimpeded.

Behavioral Concurrency

Applications that are able to take advantage of hybrid concurrency models are often limited by the cost of analyzing the possible interactions and the complexity of developing the appropriate mix of concurrency controls to satisfy the application requirements. Because the operations are not encapsulated, maintaining a hybrid system is very difficult. Someone updating an existing application must understand all of the possible interactions to know whether the change will preserve the desired behavior.

In these applications the data is generally viewed as a simple type on which the only valid operations are reading and writing. From this viewpoint, data exists as atomic entities apart from the context in which they are used and low level operations either read or write the data. Similarly, the concurrency control mechanism monitors the reads and writes of concurrent transactions to determine if a conflict has occurred.

Object databases can provide an alternative to read/write consistency checking because the objects are manipulated at a higher semantic level than the data that they encapsulate. The operations (behaviors) of objects can be much more sophisticated than simple reads and writes. By carefully defining the semantics of the object behaviors and understanding the visibility of side-effects, concurrency conflicts can be avoided in many situations. In addition, when a conflict does occur, the higher level behavior of objects can be used to resolve a low level read/write conflict while maintaining the consistency of the objects. Objects which support behavioral concurrency can increase transaction throughput by increasing the number of simultaneous operations being performed on the database. By building a reusable toolset of object classes that support concurrent update behaviors, much of the complexity of developing highly concurrent applications can be eliminated.

The initial inspiration for behavioral concurrency comes from research in concurrency control for abstract data types [Herlihy, 1990], [Schwarz and Spector, 1984], [Weihl 1989] and semantic concurrency control [Skarra and Zdonik, 1989]. In [Skarra and Zdonik, 1989], the authors distinguish between transaction semantics and data semantics for determining concurrency properties. Using the transaction approach, the concurrency properties are defined according to the semantics of the transactions and the data they manipulate. Using the data approach, the concurrency semantics on abstract data types can be defined according to the operations on the type. When this data approach to concurrency is integrated into an object database, it can provide a uniform, concurrent behavior for objects that is modular and decentralized. It is modular in that the class encapsulates the complexity of the concurrent behaviors and it is decentralized in that each instance of the class manages the concurrent access to the contained data.

By analyzing the behaviors defined in the class, one can determine which operations can occur concurrently without making the state of the object inconsistent. To aid this analysis, it is helpful to determine whether particular sequences of operations are commutative. Two operations are said to be commutative if they can occur in either order and still lead to the same consistent database state. An example of a commutative operation is a sequence of deposit operations on an account balance object. Two transactions can both deposit into the account and the final account balance is the same, regardless of the order in which the transactions are committed. However, a removal operation is not commutative with a deposit, since it is possible that the removal may fail if it occurs before the deposit.

To analyze the consistency of operations on an object, the interleaved operations by concurrent transactions can be organized into schedules. By definition, commutability of the operations implies that any reordering of commutative operations in a schedule is still valid. For non-commutative operations, however, the order of the operations in the schedule determines the possible outcomes. As mentioned in [Weihl, 1989], the set of all possible schedules is constrained by the storage system model. The typical model assumes an update-in-place storage system in which locking is used to provide consistency. In contrast, a database system can be built upon a storage system model that does not modify objects in place [Bretl, et. al., 1989]. By not modifying in place a read operation in a transaction is repeatable, providing a “guaranteed view”. With optimistic concurrency control, when a transaction attempts to commit, the system validates that the set of objects read and written do not conflict with objects read and written by other transactions that committed in the interim. This is classified as backward validation as defined in [Harder, 1984].

To demonstrate how the different storage system models affect the schedules, consider the following example using a bag, a simple collection of objects. Suppose transactions T1 and T2 start at the same time with an empty bag B in the database. Tl adds an element to B while T2 attempts to remove that element. In Herlihy's analysis, "when a transaction commits or aborts, news, of the event propagates asynchronously through the system. A schedule's commit and abort steps represent the arrival of such news at an object." Thus, the outcome of the concurrent transactions is dependent upon whether T1 commits its addition before T2 attempts the removal. If T1 commits first, then T2 sees the element in B and can successfully perform the removal. This schedule could not occur in a system that is based on a guaranteed view using optimistic concurrency control because the removal operation by T2 would always fail, since in T2's point of view B was empty and did not contain the element.

Designing Concurrent Update Classes

In designing the semantics for classes that support behavioral concurrency, it is important to distinguish between physical and logical conflicts. Physical conflicts are the low level conflicts that are a result of concurrent transactions reading and writing the same objects. Logical conflicts arise due to non-commutative operations performed by concurrent transactions. These conflicts are defined by the high level semantics of the object, and must be detected to prevent the database from becoming inconsistent. To implement behavioral concurrency semantics in Concurrent Update (CU) classes, the underlying physical conflict detection mechanisms must be extended to include the higher level semantics. When transactions attempt to commit their modifications, they may still experience read-write and write-write conflicts based upon access to the internal states of objects. These conflicts may be valid logical conflicts as well, or could be physical conflicts that can be resolved transparently to the end user by using the internal behavior of the object to merge the changes.

In order to extend the physical conflict detection mechanism to implement the concurrent update classes the underlying object database must be active, that is, it must support the execution of object behaviors so that the high level semantics can be exposed during the attempt to commit objects to the database. Passive object databases, ones that only support queries and updates to the underlying data store of objects would be difficult to extend to take advantage of behavioral concurrency because they cannot execute the methods needed to implement the behavioral semantics.

In the design of Concurrent Update classes it is important to use a non-locking approach because the locking of objects forces an ordering of the transactions and decreases the availability of objects, thereby reducing throughput. As a result, an optimistic concurrency control mechanism is required to achieve optimal performance. The general approach to the design of Concurrent Update classes is to design the physical structure of the object so that the operations are less likely to have physical conflicts and to resolve the physical conflicts detected in the commit consistency checks by using the high level semantics to merge the changes of the current transaction into the changes that were previously committed by other concurrent transactions.

Another important consideration for Concurrent Update classes is whether they are used in applications where read-write conflicts are considered important. For applications where read-write conflicts are important, all non-commutative operations must fail, while in applications where read-write conflicts can be tolerated without loss of consistency, then only non-commutative write operations fail, and read-write conflicts can be ignored. For example, a warehouse application may need to keep track of the number of items in a storage bin. Some transactions add to the bin when items are received from the manufacturer and stocked on the warehouse floor. Other transactions remove items from the bin when they are ordered by a retailer and removed from the warehouse floor. An application that maintains a count of the number of items in a bin is characterized by many small decrement operations and a few large increment operations each day, and a single read (accessing the count) operation at the end of each day to determine if additional items need to be purchased. The read operation does not need to prevent concurrent modification, since a guaranteed view provides a sufficiently accurate count for purchasing decisions. Ignoring read-write conflicts in this case provides a performance benefit, since more schedules are considered valid. Also, the implementation of the Concurrent Update classes with these semantics is more efficient, since fewer physical conflicts must be resolved.

In the sections that follow, several CU classes are defined. For each CU class its functional semantics are defined along with the kind of application it is intended to support. The concurrency semantics of each CU class is described by the sequences of operations that are or are not commutative. Finally, an example application is provided where the use of the particular CU class is appropriate.

Semantics of Concurrent Update Counters

A counter object keeps a numerical count (positive or negative) based on the increment and decrement messages it receives. It understands messages to increment or decrement itself by one (or by a given amount), and to answer its current value. Under traditional concurrency semantics, concurrent modification of a counter object results in a physical conflict. Because the increment and decrement operations are commutative, these operations do not logically conflict when performed by concurrent transactions.

At least three kinds of counters with different CU semantics can be defined. The first kind is CuCounter which has the following concurrency semantics: transactions that modify the count value do not conflict with other transactions that modify or read the count value. Thus, two transactions may both increment a CuCounter object and be able to commit successfully, or a transaction that reads the value of the count can commit successfully despite other transactions modifying the count. This CuCounter is intended to support applications such as the warehouse example described previously.

Another kind of counter is a CuPositiveCounter. With this counter, concurrent transactions can modify the counter without conflict as long as the modifications do not cause the value of the counter to become negative. If a transaction were to decrement a CuPositiveCounter such that its value would become negative when other transaction's committed changes become visible, then the transaction is not allowed to commit. Readers and writers do not conflict for a CuPositiveCounter.

The third kind of counter, a CuAccount, provides semantics similar to the Account defined in [Herlihy, 1990]. For this counter, all non-commutative operations fail. This means that a transaction that reads the account value will conflict with another transaction that increments or decrements the value. As with CuPositiveCounter, concurrent increment and decrement operations succeed as long as the value of the counter remains positive. This type of counter is appropriate in modeling financial accounts where a transaction that reads the count must fail to commit if a concurrent transaction has committed a modification to the account.

Semantics of Concurrent Update Bags

A bag is a container for objects, and more than one occurrence of the same object can reside in a bag. A bag defines messages to add, remove, and query its contents. For the most part, a CuBag behaves like a normal bag; however, CuBag's concurrency semantics are based upon commutability of write operations and readers not conflicting with writers. If the resulting state of a CuBag does not depend upon the order in which transactions commit their modifications, then the operations are logically conflict-free. For example, multiple transactions that only add to the CuBag reach the same state regardless of the order in which the transactions commit. Therefore, multiple adders to a CuBag do not logically conflict.

For some applications, addition and removal of objects to a CuBag are also commutative. If a transaction adds one object to the CuBag, and another transaction removes a different object from the CuBag, it does not matter in which order the transactions commit their changes. Consequently, neither transaction will experience conflict. Transactions that remove from the CuBag do not conflict with transactions that add to the CuBag if the removed objects are disjoint from the added objects.

An interesting case arises when analyzing the addition and removal of the only occurrence of an object in a CuBag. Within a single transaction, addition and removal of the same object are not commutative due to the semantics of removal. For example, if the removal occurs after the addition, then the removal is successful. If the removal instead occurs before the addition of the object, then the removal operation fails because the object is not present in the CuBag. However, when these operations are performed by concurrent transactions, the operations are commutative. The operations are commutative because the removal operation always fails, regardless of the order in which the transactions attempt to commit. This is due to the fact that the addition of the object is not visible to the transaction attempting its removal.

The final case to analyze is when multiple transactions are removing objects from a CuBag. As with additions, a transaction that removes from the CuBag will not conflict with another transaction that removes different objects from the CuBag. The CuBag reaches the same state regardless of the order in which the removals are committed. The one case where the removals are not commutative is when two transactions combined attempt to remove more occurrences of an object than are contained in the CuBag. For example, suppose initially there are four occurrences of an object in the CuBag. One transaction tries to remove three occurrences, while another transaction tries to remove two occurrences. The order in which the transactions attempt to commit determines which one is successful. In this case, to maintain reasonable concurrency semantics, the second transaction that attempts to commit must fail. Otherwise, two transactions would successfully remove more occurrences than there were in the CuBag.

CuBags are intended to support applications where multiple transactions write (add to or remove from) a bag, and concurrent readers are not prohibited from committing due to concurrent modifications to the bag. For example, applications may need to keep a collection of all instances of a particular class. Keeping a collection of all instances is typically done by defining a class variable or static field that holds a collection of the objects, and instances are added to the collection when they are created. CuBags are useful in this situation to prevent multiple instance creators from experiencing conflicts. Another example is an application that collects financial transaction records for statistical calculations at the end of the day. In this scenario, multiple users can add their records to a CuBag throughout the day without experiencing conflict.

Semantics of Concurrent Update Dictionaries

A dictionary is a collection of objects that can be accessed by explicitly assigned keys or names. A hash dictionary is an optimized implementation of a dictionary that utilizes a hashing algorithm to access or update an element based on its key. Users insert or update an entry in the dictionary using the "put(key, value)" method. If the given key is not in the dictionary, a new entry is added. If the key is already present in the dictionary, the new value replaces the existing value for that key. Dictionaries also have behavior to retrieve the value for a given key and to remove a key.

There are two kinds of Concurrent Update dictionaries with slightly different concurrency semantics. One kind of dictionary, a CuHashDirectory, has the same concurrency semantics as the directories in [Schwarz and Spector, 1984], in which commutability of all read and write operations determines the valid sequences of operations. Another type of dictionary, a CuHashDictionary, relaxes the commutability requirement for readers and writers of the same key. For both kinds of dictionary, addition and removal of different keys are commutative so these operations do not logically conflict. Addition and removal of the same key are not commutative within a single transaction, but are commutative when performed by concurrent transactions. The removal operation of the same key always fails since one transaction does not have visibility of the other transaction's addition. Therefore, addition and removal of the same key in a CuHashDictionary do not logically conflict.

For both kinds of dictionaries, concurrent additions of the same key are not commutative. Therefore, if two transactions add an entry with the same key, the first to commit will succeed and the second one will experience logical conflict. Similarly, concurrent removals of the same key are not commutative so one of the transactions that attempted the removal will experience conflict.

Given the concurrency semantics defined above, a CuHashDictionary (or a CuHashMap) is appropriate in applications where read-write conflicts are not important. For example, a CuHashDictionary can be used as a shared name space where transactions place objects into the dictionary to make them visible to other transactions. In applications where users expect to be adding disjoint entries into the dictionary and do not want to conflict with others who are accessing the dictionary, a CuHashDictionary can increase throughput. A CuHashDirectory is appropriate in applications where a transaction must not succeed if the value read for a key has been modified by another committed transaction.

Semantics of Concurrent Update Queues

A queue is a collection that allows users to add and remove objects in first-in-first-out (FIFO) order. To maintain the absolute ordering, concurrent transactions that modify the queue must conflict. However, if the strict first-in-first-out behavior is relaxed a little, then the concurrency can be increased. The Weakly FIFO Queue in [Schwarz and Spector, 1984], and the Semi-Queue defined in [Weihl, 1989] have similar semantics. The common property between all of these definitions is that entries added to the queue are treated "fairly", i.e., they will not become stuck in the queue but will arrive at the head of the queue at about the same time as other entries committed concurrently.

Whether or not the order of the queue is absolute, addition and removal operations are not commutative with each other. These operations are not commutative because different orderings of the operations lead to different final states. Consequently, in order to increase concurrency, commutability is not used as the basis for logical conflicts for CuQueues. Instead, the concurrency semantics for CuQueues is defined as follows: Transactions that add to the queue will not logically conflict with other transactions that add to the queue. A single transaction that removes objects from the queue will not conflict with other transactions that add to the queue. Logical conflict does occur if more than one transaction attempts to remove from a CuQueue.

This removal behavior distinguishes the CuQueue from Schwarz's Weakly FIFO Queue. The primary reason for this distinction is due to the constraints of the underlying storage system. In a system that maintains a guaranteed view it is not possible for multiple concurrent removers from a CuQueue to see the uncommitted state. Thus, they would all attempt to remove the same element and experience conflict. The CuQueue implementation is thus best for applications that involve multiple producers (transactions that add to the CuQueue) and a single consumer (a transaction that removes items from the CuQueue). It is possible to construct a system that achieves the effect of two consumers by using three CuQueues. The producers initially add their entries to Q1. These entries are then removed from Ql and placed in either Q2 or Q3 depending upon the current backlog of work or other servicing criteria. In this way individual consumers are only removing from a single CuQueue and they don’t experience conflict

To maintain the order in which objects are added to the queue without experiencing concurrency conflicts, a timestamp is included in the queue entry. This allows a transaction performing a removal from the queue to select the element with the oldest timestamp. In systems with a guaranteed view, transactions can see only committed results, so it is possible for a transaction to remove an entry that is committed before one that was actually added to the queue at an earlier time.

CuQueues are currently being used in numerous production applications. In one application, CuQueues are used to pass objects between disjoint servers. This allows distributed databases to share data by placing objects in another server's CuQueue without conflicting with other servers doing the same. Each server is assigned a CuQueue and consumes objects from it. A server can place objects in any number of other servers' CuQueues. In another application, clients place objects in a single CuQueue which is used to gather data for report generation. The report generator removes objects from the CuQueue when a report is requested.

Implementation Techniques

Once the concurrency semantics of a CU object are defined, there are a number of techniques that can be utilized to implement those semantics. As mentioned earlier, the implementation of CU classes still operates according to the rules of the underlying conflict detection mechanism. That is, concurrent modifications to the same object result in physical conflict at commit time. Through careful design one can avoid the physical conflict in the first place, and when conflicts do occur, the conflicts can be resolved where analysis has determined that concurrent transactions can commit their modifications without making the state of the object inconsistent.

Concurrent Update Read Sets

One technique for avoiding read-write conflict is to notify the underlying conflict detection engine that certain objects are part of a CU object. At commit time, this knowledge is used to ignore conflicts that are known not to upset the higher level semantics of the object. For example, a CuReadSet for a transaction is defined to hold objects on which a read operation should not conflict with other transactions' modifications.   The implementation of the methods for CU classes puts these objects in the CuReadSet. Then at commit time, if any conflicting objects (not including objects that experience write-write conflict) are also in the CuReadSet, they are excluded from the conflict set.

The Redo Log

In implementing CU objects, even though a physical conflict can occur, the higher level semantics may allow us to continue if we can ensure the database remains in a consistent state. To produce a consistent view of the database, the transaction that experienced a conflict must integrate other transactions' modifications with its own modifications.

However, the replaying of operations can fail if another transaction commits a change that invalidates one of the replayed operations. When an operation on a CU object cannot be replayed successfully, it is due to a valid concurrency conflict and the transaction is unable to commit. This is due to the fact that only operations that were initially successful are replayed when a physical conflict is detected. For example, suppose a transaction Tl attempts to remove the last occurrence of an object X in a CuBag. It is possible that another transaction T2 commits its removal of X first. When Tl attempts to commit, it experiences a physical conflict that it attempts to resolve. Tl refreshes its view of the CuBag and replays the operations. When the removal of X is replayed, it fails because X is no longer present in the CuBag, and consequently Tl' s attempt to commit will fail.

To allow for the replaying of operations when a physical conflict is detected, a redo log can be implemented to record the modifications made to certain CU objects. The redo log is similar to intentions described in [Herlihy, 1990]. However, in addition to recording the changes to a CU object, a redo log may also contain the results of read operations. This is done to support CU objects such as CuAccount and CuHashDirectory, where read operations must be repeatable, since complex behavior may have been executed based upon the value that was read.

Partitioning the Solution

An optimization for implementing concurrent update classes is to avoid conflicts in the first place. The goal is to partition an aggregate object into multiple subcomponent objects referenced by a root object which represents the original aggregate. All messages are directed to the root object, which performs at least two duties. (1) For operations that access the contents of the object, the root object is responsible for collecting the contents of the subcomponents to arrive at the aggregate contents. (2) For operations that update the contents of the object, the root determines which subcomponents are actually modified.

With this technique, the possibility of concurrency is reduced by designing the subcomponent selection criteria such that it is unlikely two concurrent transactions will modify the same subcomponent.

One example of partitioning is in the implementation of the CuHashDictionary. In this case the hash table itself provides a natural partitioning of the data. Each entry in the table represents a cluster bucket and only operations on the same cluster bucket need to be analyzed for the CU behavior.

The implementation of the CuCounter class illustrates another way partitioning an object into multiple subcomponents. Rather than implement a counter as a single numeric value, a CuCounter is implemented as multiple values, each encapsulated in its own subcomponent object. The root object of a CuCounter is actually an array of the subcomponents. When the CuCounter is sent the message to answer the current value of the count, the operation answers the sum of the individual subcomponents' values. When the CuCounter is sent the message to increment or decrement its value, the CuCounter modifies the value in only one of the subcomponents. The CuCounter chooses the subcomponent according to the current transaction's unique session identifier, which is used to index into the array of subcomponents. This technique guarantees that the transactions of concurrent sessions do not modify the same subcomponent, and thus never experience write-write conflicts.

A third example of partitioning can be found in the implementation of a CuQueue. Normally a queue would contain a pointer to the head of the queue and a pointer to the tail. Instead, consider a CuQueue which contains a reference to two separate objects, one which encapsulates the reference to the head and another that encapsulates a reference to the tail. With this structure, producers only update the intermediate head object and consumers only update the intermediate tail object. In this way a single producer and consumer can operate on the queue with no underlying write-write conflicts. For multiple producers operating concurrently on the CuQueue, when a physical write-write conflict does occur on the intermediate object, the add operation can be replayed to resolve the conflict and leave the CuQueue in a consistent state.

Considerations for Using Concurrent Update Objects

Using CU objects allows application implementers to build concurrent applications without having to write special code to avoid common concurrency conflicts. Although the functional semantics of CU objects is easily understood and matches the behavior of their non-CU counterparts, implementers need to understand the concurrency semantics of CU objects before blindly applying them in all situations. Sometimes concurrency conflicts are desirable and programmers should carefully consider the requirements of their applications before using CU objects.

Another consideration for using CU objects is the time and space costs of using them. In most cases, using a CU object involves creating and maintaining additional objects that remain hidden from the user. The implementation of a CU object can involve multiple subcomponent objects. These subcomponents may take up space that is proportional to the maximum number of users. Also, time is spent maintaining the subcomponents when the CU object is modified. If a redo log is used, additional temporary objects are created to maintain a history of operations on the CU object.

Despite the space and time costs of CU objects, the advantage is that they allow more concurrency than either optimistic or pessimistic approaches. In some use cases studied, the CU classes have increased throughput from 14% to 37% without impacting the cpu load on the system.

Conclusions

This paper provides an overview of transactions and techniques that can be used to achieve high concurrency in object oriented databases. Database features that make it easier to develop concurrent applications include:

- Guaranteed view – consistent reads
- Optimistic concurrency control – backward validation'
- Support for multiple concurrency approaches in the underlying database.
- Active databases – supports execution of object behaviors.

The "guardian objects" pattern can be used in some applications to combine optimistic and pessimistic concurrency control mechanisms to increase overall concurrency.

Finally, behavioral concurrency is introduced and is shown to be very useful technique for managing concurrency in an object oriented database. It takes advantage of the natural encapsulation of objects to hide the internal structure and complexity while extending the behaviors of the methods so that they can be invoked concurrently. By developing a reusable toolset of object classes that support concurrent update behaviors, much of the complexity developing and maintaining highly concurrent applications can be eliminated.

For more information about an object oriented database that implements a number of the concurrent update classes described in this paper see http://www.facetsodb.com.

Bibliography

Almarode, J. and Bretl, R.: “Reduced-Conflict Objects.” JOOP 10 (8): 40-44 (1998)

Beck, B., and Hartley, S.: “Persistent Storage for a Workflow Tool Implemented in Smalltalk:”, Conference on Object Oriented Programming Systems Languages and Applications, ACM Press, 1994

Bretl, B., et al.: "The GemStone Data Management System" in Object-Oriented Concepts, Databases, and Applications, edited by Kim and Lochovsky, ACM Press, 1989.

Harder, T.: "Observations on Optimistic Concurrency Control Schemes", Information Systems, Vol. 9, June 1984.

Herlihy, M.: "Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types", ACM Transactions on Database Systems, Vol. 15, No.1, March 1990."Indexed Associative Access" in GemStone Programming Guide, Chapter 9, Servio Corporation, 1994.

Maier, D. and Stein, J.: "Development and Implementation of an Object-Oriented DBMS" in Research Directions in Object-Oriented Programming, edited by Shriver and Wegner, MIT Press, 1987.

Maier, D. and Stein, J.: "Indexing in an Object-Oriented DBMS", Proceedings International Workshop on Object-Oriented Database Systems, September 1986.

Schwarz, P. and Spector, A.: "Synchronizing Shared Abstract Types", ACM Transactions on Computing Systems, Vol. 13, No.1, August 1984.

Skarra, A. and Zdonik, S.: "Concurrency Control and Object-Oriented Databases" in Object-Oriented Concepts, Databases, and Applications, edited by Kim and Lochovsky, ACM Press, 1989.

Weihl, W. E.: "Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types", ACM Transactions on Programming Languages and Systems, Vol. 11. No.2, April 1989.