

ObjectStore®

Technical White Paper

Designing C++ Applications for Concurrency

ObjectStore
A Division of Progress Software Corporation
14 Oak Park
Bedford, Massachusetts 01730
Phone: +1-781-280-4000
www.objectstore.net

Copyright © 1989-2004 Progress Software Corporation. All rights reserved. ObjectStore is a registered trademark, and Real Time Event Engine is a trademark, of Progress Software Corporation in the U.S. and other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Any other trademarks or service marks contained herein are the property of their respective owners.

April 2004

Contents

Introduction	1
ObjectStore's Concurrency Control	2
Transactions	2
Locking	3
Two-Phase Locking	3
Lock Contention	3
Deadlocks	4
Ownership	5
An Example	6
Maximizing Concurrency	7
Clustering for Concurrency	8
Clustering for Single-Process Performance vs. Concurrent Performance	8
Clustering and Spurious Lock Conflicts	9
Avoiding Deadlocks	9
Using Nonblocking Reads	10
Example of an MVCC Read	11
Snapshots	11
Reducing Lock Contention in Data Structures	11
Separating Update Operations from Read Operations	12
Using Indexes and Dictionaries	12
Partitioning	13
Avoiding Unnecessarily Long Transactions	15
Lock Contention and Schema Data	16
Libraries for Developing Highly Concurrent Applications	17
C++ Middle-Tier Library (CMTL)	17
Real-Time Event Engine (RTEE) Library	18
Conclusion	18
Bibliography	19

Introduction

Essential to any database management system (DBMS) that is used in a distributed application is *concurrency control* — the ability of the DBMS to allow concurrent processes to access the same data without violating data integrity. (Note that, as used in this white paper, the terms *client* and *process* are interchangeable. *Client* is generally used to refer to the role of a process, as distinguished from the role of the *server*.)

Consider, for example, a banking application in which two users simultaneously attempt to withdraw the remaining balance from the same account. How does the DBMS ensure that only one of the users is successful and that the other gets an “Insufficient Funds” message? In other words, how does the DBMS ensure that the users don’t access the account at the same time and debit the same balance?

Concurrency control can also impact performance. In the banking example, a DBMS would typically *serialize* the withdrawal operations, forcing one of them to wait until the other is finished and, to that extent, losing the performance benefit of true concurrent execution. The challenge posed by concurrency control is how to ensure data integrity without sacrificing concurrent performance.

This white paper discusses concurrency in applications that use ObjectStore®. It is divided into two parts:

- “ObjectStore’s Concurrency Control” on page 2 explains how concurrency control is implemented in ObjectStore.
- “Maximizing Concurrency” on page 7 discusses how to design ObjectStore applications for maximum concurrent performance.

NOTE: This white paper is a companion piece to *Designing C++ Applications for Scalability and Performance*. If you are unfamiliar with the ObjectStore architecture, it is recommended that you read the other white paper first.

ObjectStore's Concurrency Control

The purpose of concurrency control is to allow multiple users to access the same data at the same time without interfering with each other or corrupting the database. ObjectStore's implementation of concurrency control consists of three components:

- Transactions
- Locking
- Ownership

The following sections describe each component.

Transactions

The basic safeguard used by most DBMSs (including ObjectStore) to protect persistent data from concurrent access is the *transaction*. A transaction is a sequence of statements in a user application that represents a unit of work on persistent data. The user defines the boundaries of a transaction, and ObjectStore guarantees that it will preserve certain properties of each transaction during execution. The properties are known by the acronym *ACID* (*atomicity*, *consistency*, *isolation*, and *durability*) and have the following meanings:

- *Atomicity* means that the code within a transaction executes in its entirety or not at all. If the transaction *commits*, all changes to persistent data within the transaction become permanent and visible in the database. If the transaction *aborts*, all changes are rolled back to the state of the database before the transaction began.
- *Consistency* means that, when a transaction is finished, the database satisfies all of its constraints that ensure data integrity. If the database is in an internally consistent state before the transaction has begun and the transaction observes the constraints on data integrity, then the database should be in an internally consistent state when the transaction is finished.
- *Isolation* means that a transaction must not have unintended side-effects on another transaction, that a transaction's intermediate state must not be visible to another transaction. Transactions are isolated from each other when they can be executed concurrently and produce the same results as when they are executed serially.
- *Durability* means that, when a transaction commits, its changes to the database are permanently recorded and recoverable, even if the system fails.

Although all four properties are essential for ensuring correctness and data integrity, *isolation* is uniquely important for concurrent execution. If multiple transactions are executing concurrently against the same data, it is essential to the correctness of the application that each transaction is isolated from interference by the other transactions.

Locking

ObjectStore uses locks to isolate transactions from the effects of other transactions acting on the same data. The following sections describe the *two-phase locking* protocol used by ObjectStore and two conditions that result from locking: *lock contention* and *deadlock*.

Two-Phase Locking

ObjectStore uses strict two-phase locking for controlling concurrent access. This locking protocol has three requirements:

- Before a client can read persistent data, it must acquire a *read lock* or a *write lock*; and before it can write data, it must acquire a *write lock*. The lock is acquired at the point of access within a transaction.
- A client can acquire a read lock only if the data is not write locked by any other client. Also, a client can acquire a write lock only if the data is not read locked or write locked by any other client. To put it another way, a client can lock data only if the data is not incompatibly locked by another client. The following table shows which locks are compatible or incompatible:

	Read Lock	Write Lock
Read Lock	Compatible	Incompatible
Write Lock	Incompatible	Incompatible

- All locks are released at the end of the transaction in which they are acquired. If the transaction is nested in another transaction and the nested transaction commits, its locks are released at the end of the outermost transaction. If the nested transaction aborts, the client releases its locks immediately.

Once a client releases the lock on a page, the page (and all its data) becomes available for locking by other clients.

Lock Contention

Lock contention occurs when a client attempts to access persistent data that is incompatibly locked by another client — for example, when a client attempts to read data that another client has already locked for writing. The effect of lock contention is that one client is blocked and must wait to acquire its lock until the other client releases its lock. Lock contention has no effect on the correctness of the operations involved in lock contention, nor does it in any way compromise data integrity. But it does impact the performance of the blocked clients.

As described in the previous section, if a client locks data for reading, then another client can read but not write to the data because a write lock is incompatible with a read lock. The following schedule of operations for two concurrent transactions illustrates the lock contention that results from the incompatibility:

<i>Transaction 1</i>	<i>Transaction 2</i>
Read D	
	Read D (succeeds)
Commit	Write D (blocked)

Transaction 1 locks data **D** for reading. The read lock does not prevent Transaction 2 from reading the data as scheduled, but it does block the write operation. At this point, Transaction 2 must wait until Transaction 1 commits and releases its lock. The actual schedule of operations looks like this:

<i>Transaction 1</i>	<i>Transaction 2</i>
Read D	
	Read D (succeeds)
Commit	(wait)
	Write D (succeeds)

Deadlocks

A *deadlock* occurs when two concurrent transaction are each waiting for a lock that the other has, and neither can release its lock until the other does. The result is that both transactions enter into a state of deadlock. The following sample schedule of operations illustrates deadlock:

<i>Transaction 1</i>	<i>Transaction 2</i>
Read D (succeeds)	
	Read D (succeeds)
Write D (blocked)	Write D (blocked)

Transaction 1 first acquires a read lock on data **D**. Transaction 2 can also acquires a read lock on the same data because read locks do not conflict with each other. Transaction 1 next tries to write to the same data but is blocked because it cannot acquires a write lock until Transaction 2 releases its read lock. Meanwhile, Transaction 2 also tries to write to data **D** but is blocked because of Transaction 1's read lock. The result is that neither transaction can proceed.

In this example, the transactions are deadlocked over the same data. However, deadlock can occur when clients attempt to access different data and likewise can involve more than two transactions. For example, if two clients have write locks on different data, they will become deadlocked if each attempts to read data incompatibly locked by the other transaction.

ObjectStore can detect deadlock, which it resolves by aborting one of the transactions (the *victim*), allowing the other transaction to proceed. The aborted transaction can later be retried — in some cases, automatically. ObjectStore uses a default algorithm for choosing a victim in the event of a deadlock. You can change this algorithm or prioritize the transactions to specify which transactions ObjectStore should and should not consider victims.

Ownership

ObjectStore transfers data between the client and the server at the minimum level granularity of a page. The application controls the amount and types of data on a page, using any of the clustering strategies described in *Designing C++ Applications for Scalability and Performance*. When a client requests a page from the server, and the page is available, the server transfers the page into the client cache, giving the client *ownership* of the page. Ownership can be for reading or for writing.

Unlike locks, ownership can be retained across transaction boundaries. The advantage of retaining ownership is that it enables the client to lock the same page in more than one transaction without having to communicate with the server.

A client loses ownership in two situations:

- When the client evicts the page from its cache to make room for another page.
- When the client's ownership of the page is *incompatible* with another client's request for the same page. Compatibility is determined as follows:

Ownership	Read Request	Write Request
Read ownership	Compatible	Incompatible
Write ownership	Incompatible	Incompatible

When a client request is incompatible with another client's ownership, the server sends a callback message to the owning client's cache manager. Sending the message to the cache manager avoids interrupting the client. If the page is not locked, the cache manager revokes client ownership and sends a message to the server indicating that the lock has been released. If the page is locked, the cache manager sends a message indicating that the page is still locked. In this case, the requesting client must wait for the owning client to release the lock by ending its transaction. When the server gets word that the lock has been released, it transfers the page to the requesting client and gives it ownership.

ObjectStore uses the ownership mechanism to reduce communication with the server. After the client makes its initial request to the server for a page, it is possible for the client to handle all locks on that page without having to communicate with the server. As long as the client does not lose ownership of the page, it can use the page in multiple transactions without communicating with the server.

An Example

The sample application discussed in this section illustrates how ObjectStore uses locks and ownership to control concurrent access to persistent data. The application consists of two clients and an ObjectStore server. Both clients are on the same machine, and the server is on another; the clients communicate with the server across a network. To illustrate the effects of ownership on lock handling, the following discussion focuses on two points during program execution: before and after one of the clients has gained ownership of a page.

Figure 1 illustrates the program after Client Process 1 has entered a transaction and just before it executes the statement indicated by the angle bracket (>). Client Process 1 does not have the page that contains x in its cache and therefore does not have ownership of the page. Client Process 2 has just committed the transaction that contains the statement " $a = *x$;" and therefore has ownership of the same page and (as shown in the figure) has the page in its cache.

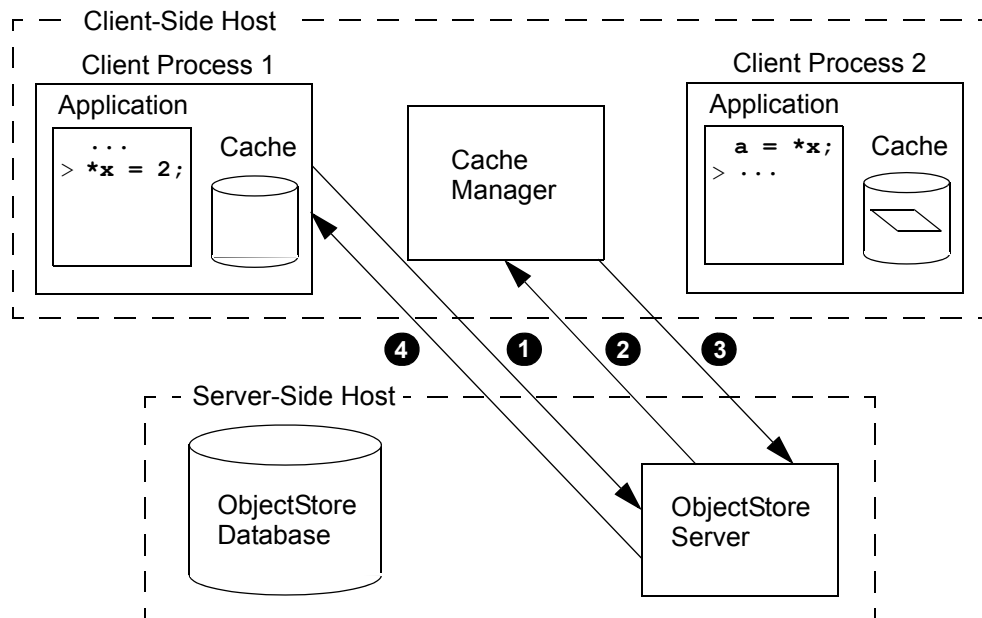


Figure 1. Locking When the Client Does Not Have Page Ownership

As shown in the figure, the following steps must occur before Client Process 1 can execute the statement " $*x = 2$;"

- 1 Client Process 1 sends a request to the server for write ownership of the page that contains x .
- 2 The server determines that Client Process 2 owns the page for reading and that its read ownership is incompatible with Client Process 1's request. The server therefore sends a callback message to the cache manager installed on the host machine with Client Process 2.
- 3 The cache manager determines that the page is not locked. It therefore revokes ownership and notifies the server.
- 4 The server fetches the page from the database and transfers it to Client Process 1, giving the client write ownership and a write lock. At this point, Client Process 1 can execute the statement " $*x = 2$;"

Figure 2 illustrates the same program after Client Process 1 has begun its second transaction and just before it executes the statement indicated by the angle bracket (>). At this point, Client Process 1 retains write ownership of the page that contains `x` and Client Process 2 no longer has read ownership.

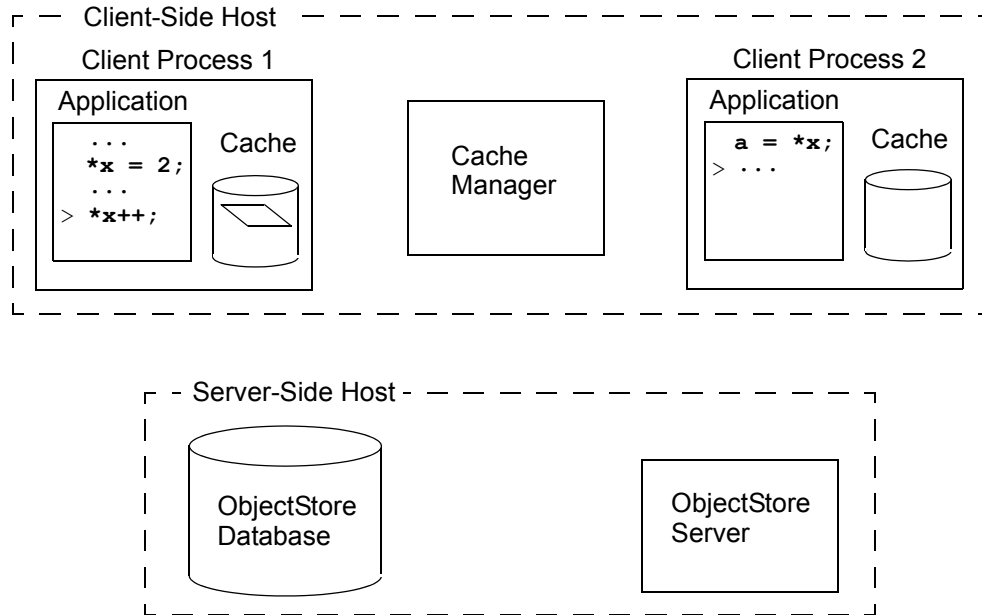


Figure 2. Locking When the Client Has Page Ownership

As shown in figure, Client Process 1 requires no communication with the server to lock the page and execute the statement. Because the client already has ownership, all lock handling occurs in the client at in-memory speed. As long as nothing happens that would cause Client Process 1 to lose ownership, it can continue to access `x` in later transactions, or any other data on the same page, without any communication with the server. This means no requests or transfers across the network, no callback messages, and no disk I/O.

The next section discusses different techniques you can use when designing your application to avoid lock contention and retain page ownership.

Maximizing Concurrency

The following sections discuss how to improve concurrency in an ObjectStore application by identifying and avoiding lock conflicts that can occur when two or more processes attempt to access the same page of persistent data. Although these sections focus on specific techniques for improving concurrency, the last section — “Libraries for Developing Highly Concurrent Applications” on page 17 — discusses two ObjectStore libraries that are useful in developing highly concurrent applications.

Clustering for Concurrency

Clustering refers to the physical arrangement of data in persistent memory so as to optimize its access. Clustering has different meanings depending on whether you are clustering data for access by a single process or by concurrent processes. When you cluster for single-process access, your goal is simply to minimize the number of disk transfers to a client process. You are therefore concerned with how much data you can cluster together on the same page. When you cluster for concurrent access, your goal is to minimize lock conflicts between concurrent processes. Clustering for concurrent access requires knowing what data to *exclude* from a page as well as what data to include.

The following sections discuss issues to consider when you are clustering for concurrency.

Clustering for Single-Process Performance vs. Concurrent Performance

As discussed in *Designing C++ Applications for Scalability and Performance*, one of the goals of good clustering is to make all of the persistent data used by an operation available on the fewest number of pages — ideally, all on the same page. For example, if an operation accesses an object and all of its attributes, then the object and its attributes should all be clustered together.

While this goal can improve the performance of a single process, it can sometimes degrade the performance of concurrent processes. Consider an application that maintains a set of **employee** objects. The application is used by different departments (for example, the payroll and training departments) to make periodic updates by querying objects by employee ID and, when the query is successful, updating the salary or training attributes of the returned objects. Suppose, also, that the application uses a clustering strategy that favors access by any department — that is, each **employee** object is clustered with all of its attributes.

Now, this clustering strategy works well as long as only one department can query and update the database at a time. But if both the payroll and the training departments want to perform updates on the same object at the same time, the first department to lock the page with the object's attributes will block the other department, forcing it to wait until the first department is finished.

Given this scenario, a better clustering strategy might be to allocate all read-access attributes (for example, employee ID) in one cluster, all training attributes in another cluster, and all salary attributes in a third cluster. When the **employee** objects are clustered this way, different departments can update the same object at the same time without blocking each other. It is true that this clustering strategy will affect the performance of any process that needs to access all attributes of each object because of the additional page transfers required to bring all of the attributes for each object into the cache. But the lowered performance of this process is likely to be offset by the increased concurrency — and the increased throughput.

When clustering for concurrency, you should consider how clustering affects the concurrent performance of all processes, not just the performance of each individual process. The increased throughput of a highly concurrent application is often a more significant measure of performance than the efficiency of a single process.

Clustering and Spurious Lock Conflicts

Although lock conflicts often occur when concurrent processes attempt to access the same data item, they can also occur when accessing different data items. This type of lock conflict is called a *spurious lock conflict* and occurs because the same page contains data needed by two different transactions. If the transactions attempt to access the data and acquire incompatible locks, one of the transactions will block the other, even though neither transaction needs the data that the other is attempting to access.

Consider an application in which two transactions are scheduled to run concurrently against different data: **D1** and **D2**. Both transactions can execute without lock contention, as long as **D1** and **D2** are on different pages. But if they are on the same page, Transaction 1 will acquire a read lock on the page ahead of Transaction 2, preventing Transaction 2 from writing to the page until Transaction 1 commits and releases its lock.

Here is the actual schedule of operations when **D1** and **D2** reside on the same page:

<i>Transaction 1</i>	<i>Transaction 2</i>
Read D1	
	Write D2: Blocked
Commit	(wait)
	Write D2 (succeeds)

Reducing spurious lock conflicts requires careful analysis of the access patterns used by the concurrent operations and a clustering strategy that meets the data requirements of the individual transaction without sacrificing the requirements of concurrent transactions. ObjectStore provides the programmer with a considerable degree of flexibility in clustering for managing this tradeoff.

Avoiding Deadlocks

Deadlocks are potentially the most expensive form of lock conflict. When a lock conflict occurs, one of the conflicting processes must wait until the other transaction finishes. When a deadlock occurs, some or all of the aborted transaction must be reexecuted, and the more frequently the aborted transaction is retried, the more expensive the deadlock is.

The most effective way to avoid deadlocks is to allocate the data used by each of the deadlocked transactions so that they are in different clusters; see “Clustering for Concurrency” on page 8. A deadlock is a specific type of lock conflict. Many of the same techniques for preventing lock conflicts can also be used to avoid deadlocks; see “Using Nonblocking Reads” on page 10 and “Reducing Lock Contention in Data Structures” on page 11.

If you cannot eliminate the deadlock by clustering or some other way, and the transaction cannot tolerate deadlocks, you can use the ObjectStore API to do one of the following:

- Designate which transactions should or should not be considered victims. As mentioned in “Deadlocks” on page 4, the API allows you to change the algorithm used by ObjectStore in selecting a victim in the event of a deadlock. You can also use the API to prioritize transactions according to victim eligibility.
- If all of the data involved in the deadlock is on the same page, use the ObjectStore API to explicitly acquire a write lock on the page before reading any of its data. This approach effectively replaces the deadlock with a simple lock conflict, forcing one process to wait until the other finishes its transaction and releases its locks.
- If the data involved in the deadlock is spread across several pages, your application can assert an access pattern requiring that, before any process attempts to access the relevant data, it must acquire an explicit write lock on a special, *empty* cluster used as a gatekeeper. While one process has a lock on the gatekeeper, no other process can access any of the data. The use of a gatekeeper enforces serialization and thus eliminates deadlock. Note that it is important that the **os_cluster** used as the gatekeeper is empty to avoid its involvement in a deadlock.

You can also use an object as the gatekeeper, but you must observe the following precautions:

- The gatekeeper object must be the only object on its page.
- The application must use an ObjectStore soft pointer to access the gatekeeper object.

These precautions will ensure that the gatekeeper object itself doesn't fall prey to a deadlock.

Using Nonblocking Reads

As described in “Locking” on page 3, when a process locks a page for reading, concurrent processes can read the same page but cannot write to it. When a process locks a page for writing, concurrent processes cannot access it either for reading or for writing. You can, however, override this locking behavior by opening the database for multiversion concurrency control (MVCC). Opening a database for MVCC allows you to perform *nonblocking reads* on the data. Nonblocking reads do not block concurrent updates, and they never have to wait for the release of write locks to read the database. In other words, a nonblocking read will never have to wait because of an update operation and will never cause an update operation to wait.

You can open a database for MVCC if the transaction does both of the following:

- Performs read-only operations on the database.
- Does not require a completely up-to-date view of the database, but can instead rely on a snapshot of the data. For more information, see “Snapshots” on page 11.

If an application's access patterns meet these conditions, opening a database for MVCC can eliminate lock conflicts with concurrent update transactions.

NOTE: If you are familiar with SQL's isolation levels, reading a database opened for MVCC corresponds to the **READ COMMITTED** isolation level.

Example of an MVCC Read

“Maximizing Concurrency” on page 7 discusses a sample schedule for two transactions. The schedule illustrates lock contention that occurs when a page is locked for reading, preventing a second transaction from writing to the same page. Here is the same example when Transaction 1 reads the page from a database opened for MVCC:

<i>Transaction 1</i>	<i>Transaction 2</i>
MVCC Read P	Read P
	Write P (not blocked)
Commit	

Transaction 2 performs a read operation, which is not blocked in any case, and a write operation, which would normally be blocked by a concurrent read operation. But in this case, the write operation is not blocked because Transaction 1 is reading data from a database that has been opened for MVCC.

Snapshots

When a process reads a database opened for MVCC, it views a *snapshot* of the data. The snapshot has the following characteristics:

- It is internally consistent — that is, it does not contain data in an intermediate state.
- It might not contain changes to the database that were committed by other processes.
- It does contain all changes that were committed before the transaction started.

The snapshot is taken at the time of the first conflict with another process’s update transaction. A conflict occurs when either of the following events happens:

- A process attempts to read a page in a database that it has opened for MVCC, and another process has locked the page for writing.
- A process attempts to update a page in a database, and another process that has opened the same database for MVCC has locked the page for reading.

In both cases, neither process is blocked and both can proceed without having to wait for a lock to be released.

Note that any read operation in any transaction that is started immediately after an update transaction has committed will see the latest value of the data. An MVCC read operation views a snapshot only when a conflict occurs.

Reducing Lock Contention in Data Structures

The use of data structures to access persistent objects can be a major source of lock conflicts and deadlocks. The following sections discuss different techniques for reducing lock contention when accessing data structures.

Separating Update Operations from Read Operations

Database applications often make use of an update transaction that reads through a data structure to find and update objects of interest. When this access pattern is used by multiple concurrent processes, the data structure can become a source of numerous lock conflicts and deadlocks. The problem is that updating an object requires a write lock that blocks concurrent processes, preventing them from reading or writing to the same object or other objects on the same page. And the longer the transaction, the longer that the blocked processes must wait for the lock to be released.

One way to reduce contention is to split up the transaction by breaking out the read and update operations into separate transactions. To access the data structure, a process would first execute a read-only transaction to locate and record the addresses of objects requiring updates. The update operations would occur in a separate, write transaction. ObjectStore provides a retain-address API so that an application can retain addresses of persistent objects across transaction boundaries without their becoming invalid. In other words, this API allows you to locate data in one transaction and update it in another.

The advantage of this approach is that multiple, concurrent processes can execute the read-only transaction without blocking or deadlock. Of course, it is still possible for a process in the read-only transaction to block (or be blocked by) another process in the update transaction, but it is less likely to happen than in the single-transaction case. Even when blocking does occur, the shorter transactions mean that the blocked process has less time to wait for the lock to be released.

One disadvantage of splitting a transaction into shorter transactions is that you lose the ACID properties of the single transaction — in particular, atomicity and isolation; see “Transactions” on page 2. When you read and update a persistent object in the same transaction, ObjectStore guarantees that no other process will make any changes to the object from the point when you first read the object until the end of the transaction. After splitting the transaction, the application must ensure that, during the interval between the two transactions, no other process has changed or deleted the object since its address was read in the read-only transaction. Likewise, the application must decide what to do in the case when one transaction commits and the other aborts. In the single-transaction case, ObjectStore ensures that, in the event of an abort, the entire transaction is rolled back to its pretransaction state. When you split up a transaction, your application must provide a recovery mechanism to “roll back” the effects of the committed transaction after the second transaction aborts.

Using Indexes and Dictionaries

Traversing a large extent of objects can result in many lock conflicts and deadlocks that have a cumulative impact on concurrent performance. If the traversal requires a process to read each object in the extent even though it only updates a few, the process will acquire read locks for each page containing an object. The read locks will conflict with a concurrent process that attempts to update any of the locked objects. As a result of the conflict, the blocked process must wait until the blocking process finishes its transaction and releases the read locks. The more objects read by the blocking process, the longer the blocked process must wait.

If you are using one of ObjectStore’s collection types as the extent, an especially effective way to reduce both lock conflicts and deadlocks is to add an index to the collection. As explained in *Designing C++ Applications for Scalability and Performance*, an index is a highly efficient data structure that uses a search algorithm to minimize the number of objects visited by a query operation. Adding an index to a collection not only reduces the number of pages that the operation

must read lock, but it also speeds up the entire operation so that the locks are held for a shorter period of time.

Note, however, that an index can become a concurrency hotspot if the indexed attribute is frequently updated. As an alternative to an index, you can use a dictionary as the extent and achieve the same effect as adding an index. A dictionary is one of the collection types provided by the ObjectStore collections facility. It stores elements as key-value pairs. Given a key, such as an employee ID or a name, the dictionary returns a value, such as the address of the object that contains the key. A dictionary uses an optimized hash table that reduces the number of objects visited by a query operation, thus reducing the number of read locks accumulated during a search. Also, because a dictionary speeds up a query operation, locks are held for a shorter period of time.

Partitioning

As discussed in the previous section, a dictionary can reduce the number of pages that are read locked when traversing an extent. However, it can also be a source of lock conflicts if objects are frequently added to the extent. For accessing and managing a large extent that is frequently accessed by concurrent processes, partitioning can provide a better choice.

Consider a sample application that uses a dictionary to map employee names to the addresses of **employee** objects, as shown in Figure 3.

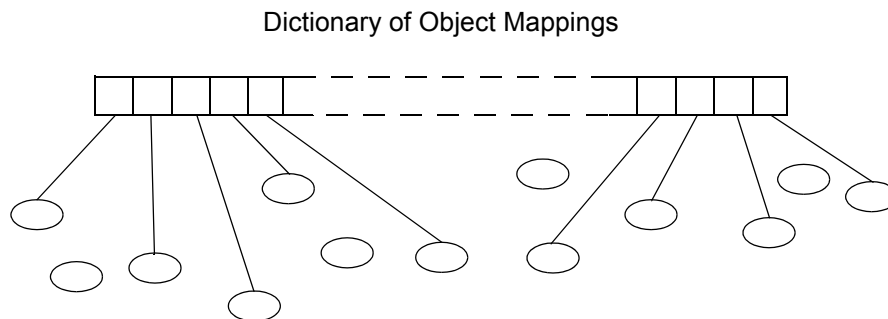


Figure 3. Sample Application Using a Dictionary as an Extent

Each time an object is added to the database, an update transaction must add a new mapping to the dictionary. Other operations enable querying the database and iterating over all objects in the dictionary to perform selective updates. As the number of objects and of processes accessing the data structure increases, the dictionary can become subject to frequent lock conflicts.

Here is a procedure for improving concurrency by redesigning the application to use partitioning:

- 1 Divide the set of objects into partitions, assigning a range of names to each partition. For example, names beginning with the letters *A* through *B* would go in the first partition, *C* through *D* in the second, and so on.
- 2 Map each partition to a different segment of the ObjectStore database.
- 3 Add the objects in each partition to a dictionary that is stored in the same segment. This dictionary maps names to object addresses. Essentially, this partition-level dictionary is just like the dictionary in the original application (see Figure 3), except that it contains a fraction of the number of objects.

- 4 Create a top-level data structure that maps *ranges* of employee names to segments. This data structure could be a dictionary or a simple C++ array — in fact, any structure that can serve as an index for locating the segment that contains an object. The number of entries in the data structure will be the same as the number of segments used as partitions.

Figure 4 shows the structure of the database after partitioning.

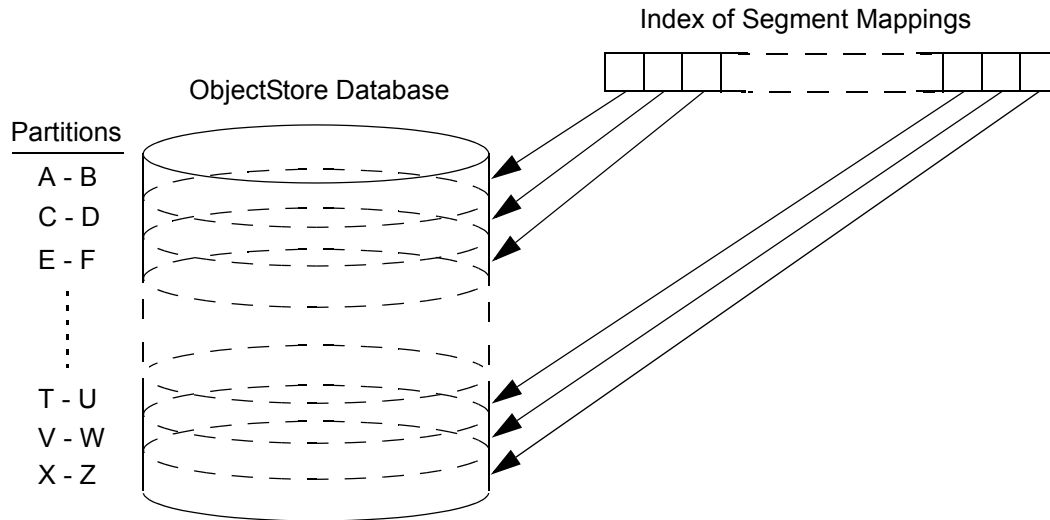


Figure 4. Sample Application After Partitioning

The advantages of partitioning as implemented in this example are:

- Partitioning the data allows the processing to be partitioned so that each data partition can be assigned to its own client process. With partitioning, multiple processes can execute concurrently without lock conflicts because each process is operating on its own data set.
- The top-level data structure is no longer a concurrency problem because updates occur in the partition-level dictionary, not in the top-level data structure (see next item).
- The use of smaller dictionaries to access the subset of objects within a partition reduces the time it takes to iterate over the objects in a partition. Also, fewer objects will be added to any partition, resulting in fewer updates to each partition-level dictionary and therefore in fewer lock conflicts and deadlocks.
- Partitioning can increase performance significantly by making optimum use of each process's client cache. See the discussion of partitioning in *Designing C++ Applications for Scalability and Performance*.

The following are the disadvantages of partitioning as implemented in the example:

- Navigating from the top-level index to an object requires an extra step.
- Top-level iteration and query operations are not as simple as in the original design and require more work on the part the application programmer.

Avoiding Unnecessarily Long Transactions

In general, a long transaction can result in more lock conflicts and deadlocks than a short one because it locks more pages for longer periods of time, making it more intrusive on concurrently executing transactions. Here are some tips for avoiding lock contention in a long transaction:

- Keep transactions as short as possible. Short transactions are less likely to result in blocking and can reduce the time that a blocked process must wait for a lock to be released. Note, however, that dividing a long transaction into a series of short ones risks the loss of a transaction's ACID properties. For example, if an abort occurs in the middle of a long transaction, any intermediate changes to persistent data that were made in the earlier part of the transaction are automatically rolled back to the pretransaction state. Rolling back a transaction avoids the possibility of writing inconsistent data into the database. When you split up a long transaction, your application must ensure that the ACID properties of the original transaction are not violated. For more information about the ACID properties, see "Transactions" on page 2.
- Do not access other systems or wait for user input during transactions. Either of these conditions can needlessly prolong the transaction and increase the likelihood of lock contention with concurrent transactions.
- If possible, execute long transactions outside of peak hours, when they are less likely to intrude on other transactions. For example, a long-running loader transaction can often execute at start-up or initialization time, when it is less intrusive.
- If a long update transaction conflicts with read-only transactions and the logic of the application allows the use of MVCC, open the database for the read-only transactions in MVCC mode. An MVCC database can be read without blocking, or being blocked by, concurrent update transactions. For more information, see "Using Nonblocking Reads" on page 10.

NOTE: Although short transactions can improve concurrency, they can also increase the number of transactions and thus add to the commit overhead. Every transaction incurs overhead when it commits, especially update transactions, which must transfer updated pages back over the network to the server. One way to reduce this overhead is to batch many short update transactions into one long transaction — resulting in fewer transfers. You may need to perform benchmarking tests to determine which has the greater benefit to performance: the improved concurrency of small transactions or the reduced commit overhead of long transactions.

Lock Contention and Schema Data

Thus far, this white paper has examined lock contention as occurring when concurrent applications attempt to access user data. But lock contention can also occur when ObjectStore accesses its own system data — that is, *metadata* that ObjectStore uses to manage the application's data. Because ObjectStore must frequently access metadata while user applications are performing database operations, it can become a source of lock contention. ObjectStore is designed to ensure that metadata does not become a concurrency bottleneck, and in most cases access to metadata occurs transparently to the application. In some cases, however, the application should take into account the impact of metadata on concurrency in order to attain maximum throughput and eliminate any unnecessary blocking.

An example of metadata that can become a source of lock contention is schema data. *Schema data* is persistently stored information about the types of objects in a database. It is analogous to the system tables used by a relational database to manage and access tables containing user data. Similarly, ObjectStore uses schema data to store and retrieve persistent objects.

When you build an ObjectStore application, ObjectStore generates schema data for all persistent objects accessed by the application and stores this data in an application schema database. At run time, when the application accesses a user database, ObjectStore adds schema data from the application schema database to the user database. This operation requires ObjectStore to lock the pages that contain the schema data, thus creating the condition for lock contention. Lock contention involving locked schema pages can occur in either of the following circumstances:

- When ObjectStore adds schema from two applications to the same database concurrently.
- When ObjectStore adds schema from one application while it is concurrently performing schema validation on behalf of another application. ObjectStore uses schema validation to ensure compatibility between the application schema and the database schema. It occurs the first time an application accesses a database and can occur in later transactions if another process subsequently updates the schema.

To reduce the chances of lock contention, ObjectStore by default uses *batch mode* when installing schema in a user database. All schema data in the application schema database is added to the user database when the application first accesses the database — typically, when the application creates the database. Thereafter, no schema needs to be added to the database unless the application has been changed to access persistent objects of a new type.

As an alternative to batch mode, you can specify *incremental mode*. When ObjectStore uses this mode, it installs schema for a particular type only when an application first allocates persistent storage for an object of the type. Incremental mode has two advantages:

- It spreads the cost of schema installation over the lifetime of the database.
- It installs only schema for types that are allocated in the database, thus reducing the size of the schema data in the database.

On the other hand, incremental mode can increase the chances of lock contention because it spreads schema installation across the lifetime of the application, rather than confining it to one time. When batch mode is in use, lock contention during schema installation can occur only when a process first accesses a database.

Although batch mode is the default, ObjectStore uses incremental mode when installing schema associated with Dynamic Load Libraries (DLL). Instead of installing all of the DLL schema at the same time, ObjectStore installs schema for each type only when the type is first instantiated, thus reducing the amount of schema data stored in the database. Because component schema is installed incrementally, an application that uses DLLs (such as the collections facility) can experience lock contention when using DLLs in a concurrent environment.

ObjectStore provides simple coding techniques for minimizing lock contention when using component schema, but discussion of these techniques goes beyond the scope of this paper.

Libraries for Developing Highly Concurrent Applications

The following sections describe two ObjectStore libraries that can be used with applications requiring optimum concurrency: the C++ Middle-Tier Library (CMTL) and the Real Time Event Engine (RTEE™) Library.

C++ Middle-Tier Library (CMTL)

Up to now, this white paper has focused on the use of ObjectStore in a two-tier architecture, consisting of ObjectStore clients and servers. But ObjectStore can also be used in a three-tier architecture. The C++ Middle Tier Library (CMTL) makes ObjectStore's transaction processing and management available to the middle tier. CMTL makes it possible for an application that uses middleware software such as an ORB to enact transactions on persistent data with all of the guarantees of the ACID properties and concurrency control of an ObjectStore transaction, without having to communicate with back-end servers.

CMTL provides two levels of concurrency. At the top level, CMTL uses ObjectStore's sessions facility to enable different threads running in the same process to execute multiple transactions concurrently, as if they were separate ObjectStore clients sharing the same process. ObjectStore uses the same locking model for controlling concurrent access in different sessions in the same process as it does in different client processes. In fact, ObjectStore ensures transactional consistency and data integrity across all sessions in all processes. In a multithreaded application server, middleware (for example, CORBA) assigns the data requests to threads, and CMTL routes each thread to a session.

CMTL also provides a fine-grained concurrency at the transaction level of the application. CMTL recognizes two kinds of transactions:

- Virtual transactions
- Physical transactions

Typically in a three-tier system, the middle-tier application must service many short requests submitted by a large number of front-end users. CMTL maps these requests to *virtual transactions*, which are transactions at the application level and are coded by the programmer. At run time, CMTL batches a group of virtual transactions and schedules them to execute as a single *physical transaction*. Batching and scheduling are performed by CMTL, transparently to the programmer. Batching reduces commit overhead by enabling ObjectStore to execute multiple virtual transactions as a single ObjectStore transaction, while at the same time ensuring the ACID properties of each virtual transaction. Based on run-time information about each virtual transaction, CMTL can determine which transactions it is safe to execute concurrently and

Conclusion

schedules them accordingly.

You can configure an application to use CMTL declaratively or programmatically. To use CMTL declaratively, you provide an XML-based configuration file that describes (for example) the number of caches that you want the application to use. The application reads this file at run time, without the programmer's having to change source code and recompile. This means, for example, that you can configure an application to use two sessions and later, to meet an increased number of data requests, reconfigure the application for four sessions just by editing the configuration file.

Real-Time Event Engine (RTEE) Library

The Real-Time Event Engine (RTEE) library is designed for use with multiprocess, distributed database applications that simultaneously collect, process, and query time-based data (*events*). RTEE is especially useful for financial analysis and modeling applications that collect and analyze such events as stock-market trades, and network management applications that collect network event data for billing and operational support.

Such applications must capture streams of event data in real time. Event collection must therefore be capable of processing high volumes of data without interruption. RTEE's feed collector is designed to be highly concurrent with other processes used by the application. For example, on a two-processor Solaris SPARC platform, RTEE can collect over 10,000 events per second in real time without blocking, or being blocked by, any other concurrent process. The RTEE feed collector creates its own transactions automatically and transparently to the programmer.

The following are other RTEE services that can run as separate processes without concurrency conflicts with the feed collector:

- A high-speed query server that provides a flexible and powerful query API for accessing events
- An events editor that can be used to insert, correct, update, remove, or query (for the purpose of editing) events that have been collected
- An archival function that can move and reorganize collected events to another database for more efficient access
- A bulk data loader that can move huge amounts of historical event data into an RTEE database at high speeds

The RTEE library was specially tuned for concurrent performance and uses many of the techniques described in this white paper.

Conclusion

A previous white paper, *Designing C++ Applications for Scalability and Performance*, shows how ObjectStore's client-centric architecture benefits performance by moving most of the data handling out of the server and into the client, thus reducing server traffic and freeing the application from dependence on the performance of server hardware and network connections. The same architectural design underlies ObjectStore's concurrency control. Once data is in the client cache and is owned by the client, the client can make locking decisions without having to wait on costly

trips to the server for locks. Moreover, ObjectStore's architecture gives the application direct control over how multiple processes access data concurrently. By understanding the application's access patterns and by using the techniques described in this white paper, the programmer can ensure that the application's concurrent processes work together with a minimum of conflict and a maximum of throughput.

Bibliography

Bernstein, Philip A., and Eric Newcomer. *Principles of Transaction Processing* (San Francisco CA, 1997).

Gray, Jim, and Andreas Reuter. *Transaction Processing: Concepts and Technique* (San Mateo CA, 1993).

Khoshafian, Setrag. *Object-Oriented Databases* (New York NY, 1993).