An Approach to Transaction Processing for Distributed Object-Oriented Databases

Mikhail M. Gilula
gilula@4ds.com

Jacob Gluz
jacob@4ds.com

Alexei P. Stolboushkin
aps@4ds.com

Anatoly Volkhover
anatoly@4ds.com

Fourth Dimension Software
555 Twin Dolphin Dr.
Redwood City, CA 94065-2102

April 5, 1999

Abstract

We discuss design and implementation issues related to online transaction processing (OLTP) for object-oriented databases in heterogeneous environment. In the described two-layered scheme, nested transactions are controlled at the lower level of a distributed file system, uniformly for all types of data and indexes. The paper focuses on the advantages of this scheme, as well as on the algorithms of its efficient support.

The approach has been implemented in *COOL*-NT (Common Object-Oriented Language–New Technology)—an object-oriented application development platform for medium to large-scale OLTP systems.

1 Introduction

Most existing OLTP systems have been developed for relational databases. As systems supporting transactions for object-oriented databases have more recently started to appear, several of them relied on the same techniques and solutions. It has become clear, however, that objects do not generally map well into the relational design; specifically, as far as transaction control is concerned, the number of input/output operations becomes prohibitively high. Distributed databases only make things worse, inasmuch as individual objects lose their identity and get clustered inadvertently.

At the same time, providing native transaction support for object-oriented databases presents a challenging problem, unless severe restrictions are put on the transactional objects. For example, in the Argus programming language [5], the developer needs to explicitly consider synchronization and recovery issues for complex types.

We pursue an approach in which transactions are controlled at the level of a distributed file system, and then transactions for the object-oriented database are implemented through the lower-level transaction control, uniformly for all types of data and indexes. This layering allows implementation of full-scale nested transaction support in a distributed environment without compromising the strengths of object-oriented database design. As an added benefit, multi-platform support is simplified under this approach.

The idea of transactional file systems is not novel—see [2] and the bibliography therein. However, the design philosophy of *COOL*-NT is different in that the partitioning of data is encapsulated in the file system layer. Beyond this level, we can think of a *COOL*-NT cluster as of a single server. Besides, *COOL*-NT provides an object-oriented programming language, which supports an object-oriented database with nested transactions.

In this presentation we want to share our experience in carrying out these ideas. We start off by trying to sell the reader on the idea of transaction processing on clusters of machines, specifically, clusters of lower-cost computers.

2 OLTP on Computer Clusters

With the processing power of personal computers doubling every 2 years, personal computers have clearly entered the age of puberty and are soon to meet challenges once seen as exclusively reserved for mainframes. For many applications, OLTP systems may be accommodated on clusters of lower-performance machines as successfully as on mainframes. This is especially true when the crucial performance factor is the number of *transactions per second,* rather than the length of any single transaction. Of course, for time-critical OLTP, expensive high performance computers will continue to be superior.

Generally, clustered OLTP systems do scale well, and it is not unreasonable to expect that the average system performance will grow linearly in the size of the cluster of machines.[1] This scalability relies on the partitioning of the database across the cluster to allow several database operations to run concurrently. In *COOL*-NT this is achieved by running a database server on each machine in the cluster. This server is responsible for performing operations on the local partitions, as well as for automatically routing database requests to the other machines in the cluster. Transactions on the database also are supported at this level.

In implementing this approach, one faces challenges not usually encountered when dealing with existing OLTP environments such as Tandem's TMF (Transaction Monitoring Facility). For one thing, because of the large variety of hardware and software solutions used on smaller machines, the system should be prepared to readily handle these numerous architectures. Also, the hardware here is less reliable, and transaction processing generally lacks hardware support. It may realistically be assumed that any node in the cluster, as well as the networking hardware, may fail at any moment. At the software level, extra care should be taken to guarantee that if this is the case, uncommitted transactions are rolled back completely upon fixing the problem.

In the framework of *COOL*-NT's multi-layered architecture, across-the-cluster

[1] except going from 1 to 2 machines is sublinear because of the one-time communication charge.

partitioning and database transaction processing are encapsulated in the lowest layer which is called SMF (Stream Manipulation Facility). None of the higher layers deal with the issues of hardware failure recovery or cross-platform data accessibility.

3 Object-oriented Nested Transactions

One of our main goals has been to fully support true object-oriented databases. The object repositories, called *containers,* are each capable of storing objects of a certain specified class, *as well as of any class derived from it.* Indexing of the objects in a container, too, is supported in object-oriented terms. This means that indexed keys are defined using data and/or (possibly, virtual) function members of the base class. Several special features of indexes that may require nontrivial additional work under other designs, fit here naturally. These features include:

1. **Multiple entry indexes**: For example, a customer can be found using either one of his telephone numbers.

2. **Parameterized orderings**: For example, points on a surface can be indexed by their distance from a given point (parameter).

Handling of containers is supported by the CMF (Container Manipulation Facility). Partitioning of data across the cluster and distributed nested transaction control (cf. [3]), as was said earlier, are isolated from CMF and supported at a lower level by the SMF, which is unaware of objects or containers.

Having the isolated SMF layer also facilitates potential extensions of the database by additional indexing mechanisms. By way of example, if one wanted to arm the database with R-tree indexing [4], all one would need to do would be to map the concurrent R-tree algorithms into the existing SMF layer, without worrying about partitioning or transaction control.

Efficient implementation of this scheme relies on several original techniques. The main problem that needed to be resolved was as follows. Because transaction control is not object-oriented, but essentially file-oriented, insertions, deletions, and modifications of the objects under transactions lead to a higher degree of index locking (indexes are implemented via a version of B-trees). The problem is only made more severe by the fact that nested transactions generally lead to a higher degree of transaction isolation.

Our solution for this problem is two-fold. First, to speed up transactions we store all non-committed modifications in cache memory. For nested transactions it means that only the outermost transaction committing will cause any I/O operations, while the inner transactions are fully processed in memory. As a side effect, this approach radically improves efficiency when aborting transactions (or subtransactions) is used as a programming technique.

Second, we reduce isolation of transactions by the following means:

- By using **holds** in addition to locks. A hold is a lock of a special less-restrictive type that may coexist with another hold (but not with a "true" lock!). More specifically, a record may be held by one or more independent transactions at the same time. A hold guarantees that the record may not be changed by any of the concurrent transactions.

- By employing the **dirty read-through-lock** semantics (see [3]) in which even uncommitted changes are visible to other transactions. The effect of this is that searches and retrievals in the database are not normally slowed down by modifications. If necessary, however, an application may gain access to the *clean* (=committed) data by first locking or holding it.

- By using a new page-oriented isolation protocol for B-trees: The main differences from the standard concurrent algorithms for B-trees with page-oriented isolation (cf. [1, 3]) can briefly (and approximately) be described as follows:

  1. In the retrieval case, no locking is used. Instead, consistency of retrieved pages is continuously checked, and in case of a failure the search restarts. This guarantees that retrievals are not held down by modifications
  2. When inserting or deleting, at first locking follows the assumption from [1] that the target leaf will not split, hence, locking is first attempted on the leaf node only. If this fails, however, we further assume that the leaf's parent will not split, and, respectively, attempt to lock the two last nodes in the path, and so on. As compared to the algorithm from [1] (that in the case of failure starts locking from the root node down), our locking method is potentially subject to more failures and restarts, however, it results in the lowest possible degree of locking

We also are taking advantage of the effect that timeouts have on conflicting transactions. When concurrent transactions each involve several locks and changes, increasing the timeouts may actually hurt all the transactions, because of the application deadlocks. We employ an elaborate scheme in which timeouts are sometimes artificially lowered, but the transactions aborted because of the timeouts are repeated. In many cases, this has proved to be efficient.

The syntactical approach to transaction processing in *COOL*-NT may also interest the reader. We adopt the viewpoint that `begin transaction` and `end transaction` are "parentheses", like, say, `while` and `end`. Thus, balancing of these parentheses becomes a syntactical problem. One implication is that, although *COOL*-NT is multi-threaded, no independent thread may start within an ongoing transaction. However, a transaction may be made multi-threaded by way of *forking,* which starts several threads and then waits for all of them to complete.

Conclusion

In this presentation, we focus on object-oriented distributed nested transaction control in *COOL*-NT and outline the main ideas and techniques underlying our approach.

There is more to *COOL*-NT than just its database. We generally left aside the elaborate process-scheduling mechanism, and the multi-tiered client/server architecture that may be especially helpful in designing over-the-Internet applications.

Although we tried to highlight distributed transaction processing on lower-performance machines, by its potential, *COOL*-NT is suited equally well for high performance systems. However, we believe it to be *COOL*-NT's unique feature that it can efficiently run on mixed-platform clusters.

COOL-NT is an application development platform aimed at medium to large-scale distributed online transaction processing systems. Among its main features are:

- **Clustering technology** that allows easy database partitioning, as well as dynamic distribution and scheduling of server processes, across a cluster of machines
- **Multi-platform support** that simplifies integration of a variety of hardware and software architectures within a cluster
- **Structured nested transaction control**
- **True object-oriented database**
- **Full-featured object-oriented programming language** with integrated database programming functionality

At present the short-term market orientation is on Windows NT. However, the architectural design of *COOL*-NT greatly facilitates support of virtually all multi-tasking platforms, and other versions of *COOL*-NT are forthcoming.

References

[1] R. Bayer and M. Schkolnick, Concurrency of Operations on B-Trees, *Acta Informatica,* **9**(1), 1977, 1–21.

[2] J.L. Eppinger, L.B. Mummert, and A.Z. Spector, "Camelot and Avalon: A Distributed Transaction Facility", Morgan Kaufmann Publ., Inc., 1991, 505 pp.

[3] J. Gray and A. Reuter, "Transaction Processing: Concepts and Techniques", Morgan Kaufmann Publ., Inc., 1993, 1070 pp.

[4] A. Guttman, R-trees: A Dynamic Index Structure for Spatial Searching, *SIGMOD Record,* **14**(2), 1984, pp. 47-57.

[5] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler, Implementation of Argus. In *Proc. of the 11th Symp. on Operating System Principles,* ACM, November 87, pp. 111–122.