Building Middleware for Higher-Order Object Data Management

Helmut Kaufmann   Michael Rys   Hans-Jörg Schek

Abstract

Future object managers will utilise existing relational database systems as primitive building blocks replacing today's file systems. SQL, similar to assembler language today, will be used to implement higher-order object managers. Current middleware layers will be extended to a coordination layer that uses and controls different database systems used as storage systems and as index managers. We substantiate this claim by presenting two middleware prototype systems as first steps into this visionary direction. One provides a generic object data model layer to the clients. It is mapped to a relational multiprocessor database system. The other is a textual document object manager that is implemented using a TP-monitor together with a relational multi-processor system for storage management. A key aspect in both prototypes is high-level parallelism for high performance in terms of response times and throughput. We achieve this by a synthesis of research results in the area of multi-level transactions with existing transaction processing technology in relational database systems. This synthesis allows to transform intra-transaction parallelism to inter-transaction parallelism with the advantage of the latter being well supported by today's database systems. Practical evaluations show significant improvements compared to conventional solutions.

1 Introduction

At the client side a higher-order object manager is an application object service interface that provides its tools and services to many clients. On the other end, at the storage side, it utilises one or several database systems as storage managers. In many aspects a higher order object management system is a database of databases: It maps the client object model to the data models of "its" databases and it maps client transactions to transactions that are executed by its database transaction managers. The higher-order object system therefore is a mediator or a coordinator between the clients and the underlying database systems and possibly underlying file systems that are used as storage managers. Towards the clients it behaves like a sophisticated object database system. It supports asynchronous, parallel executions of high- level object operations, hides distribution and replication, necessary for performance and availability, and supports various execution guarantees (e.g. spheres of atomicity, spheres of isolation) in terms of modern transaction management.

The exciting aspect is the realization of such a system. In view of the technology in hardware and communication systems development as well is in view of the mature technology of relational database systems we can afford to utilise relational databases on multiprocessor workstation system or on many high-end PCs as our primitive building block for the implementation. We construct sophisticated services by the repeated usage of the more primitive services of relational database systems. Therefore, also from a realization point of view, we call such layers "higher-order" or "upper-level" object managers.

We have built two such coordinator middleware prototype systems. One is TPM/TEXT, a document management system, the other is BUTTERFLY, a generic object manager that provides an interface similar to the ODMG standard. Both systems were built by extending a relational storage service with an additional level on top of the relational systems which provides the semantics of these applications. First, we will present the common architecture and aspects of these two systems and outline two different approaches to realise these middleware components. Next, we present the architecture of BUTTERFLY in section 3 and of TPM/TEXT in section 4. In addition some results of the performance evaluation are discussed to verify our approach.

The two prototype systems were developed within two doctoral theses that will be available in early 1997 [Kau97, Rys97]. This short paper summarises the two directions under the common framework of higher-order object management, the research area of the DBS Group at ETH Zurich [SABT].

2 Common Architecture

Both systems form middleware components which provide their specific services to the clients and utilise a relational multi-processor database system as their storage and execution server (see figure 1). In addition, the middleware components implement internal system services which are used to implement the client services on the relational resource manager using the resource manager's language (i.e. SQL). These internal services are not directly accessible from the client applications, and differ in their semantics and implementation between the different systems. However, due to the common goals and approaches presented in this section, they fulfill similar needs.

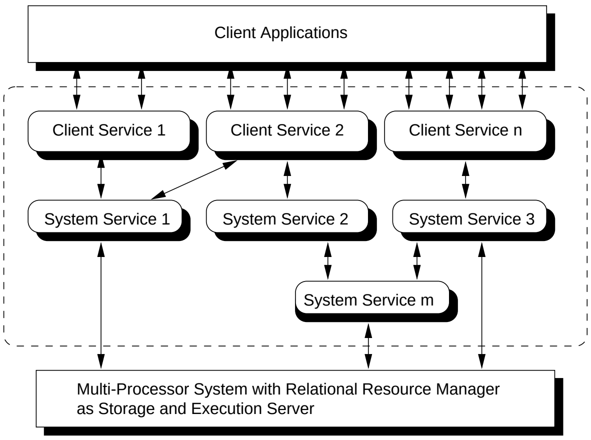

Figure 1: Common system architecture

To achieve the premier goal of efficiency, both systems are replicating data on the relational storage level. TPM/TEXT provides textual index structures, while BUTTERFLY materialises object classes and replicates inherited object properties. This allows fast retrieval of data by avoiding long searches on original data and the runtime object recomposition and view materialisation, respectively. The update operations become more complex, because of the added maintenance of the replicated data items. In order to improve the update performance, the resulting SQL updates of the replicated items are parallelised by the presented systems while preserving the transactional semantic of the user update operation. Thus, both systems apply *intra-transaction* parallelism.

Today's (relational) database systems provide support for *inter*-transaction parallelism, i.e. the concurrent execution of a large number of independent user transactions. Within each of these transactions, individual operations, e.g. SQL statements, are executed strictly sequentially.

Therefore we need other ways to achieve a parallel execution on the SQL statement level. We execute the generated SQL statement(s), which perform such a sub-operation, as a SQL transaction and thus exploit the relational system's capability to execute the SQL transactions in parallel. In other words, we utilise the inter-transaction parallelism provided by the storage server to achieve intra-transaction parallelism on the higher level. In order to assure a correct and efficient execution of the higher-level transactions we employ open-nested transactions [WS92] as the transactional framework for the correct and efficient parallel execution of the sub-transactions.

We investigated two major approaches to build our middleware components: building a specific parallelisation service into the middleware layer or building the whole middleware layer by extending an existing middleware component.

The document manager TPM/TEXT was build by extending an existing middleware component, namely Tuxedo, a transaction processing monitor (TP monitor), which provides some but not all of the required functionality (see [Tre96] for an functional description of TP monitors). It was extended by the client services and by the open-nested transaction management to parallelise the transactions. The advantage of extending a TP monitor is that it provides not only the parallel execution of SQL transactions but also additional features such as different scheduling algorithms, automatic load balancing and – to a certain extent – resource management. The disadvantage is that it may provide too much unneeded functionality and therefore overhead.

On the other hand, a newly build component can exactly be tailored to the particular needs and thus provide better performance by keeping the parallelisation and scheduling overhead to the bare minimum. This approach is used in the implementation of the BUTTERFLY object manager.

3 BUTTERFLY

The object database system BUTTERFLY has been developed to efficiently support query intensive applications such as decision support systems [RNS96]. We achieve that by making extensive use of replication of the object data on a relational storage server as outlined above. Figure 2 gives an overview of the BUTTERFLY specific system architecture.

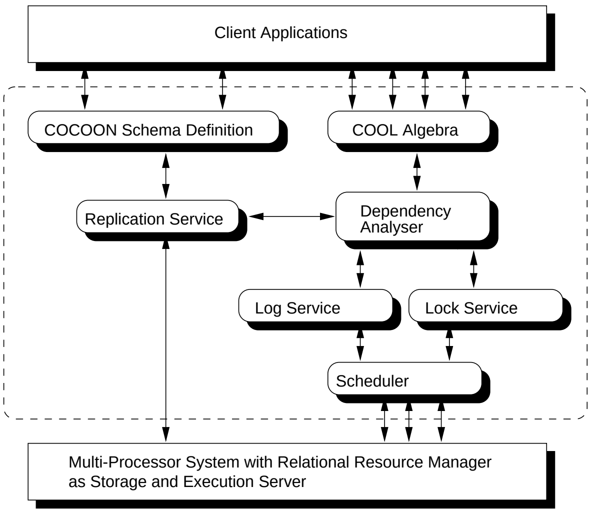

Figure 2: BUTTERFLY System Architecture

A BUTTERFLY user implements its database by using the COCOON object data model and formulates queries and updates using the COOL object algebra [SLR+94] which are the public services of the BUTTERFLY middleware. The COCOON schema concepts such as classes, which group objects of same structure by means of a predicate, are mapped to the relational storage server using replication and materialisation (see figure 3).

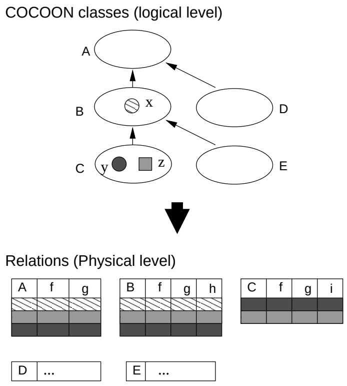

Figure 3: Mapping of BUTTERFLY objects to relations

In this (simplified) example, the most specialised class which contains the objects y and z is C, while x belongs to B. Due to the generalisation hierarchy and the subclass inclusion, these objects also implicitly belong to their superclasses B (for y and z) and A (for all three objects). In the mapping to the relations, full copies of all the objects are explicitly stored as tuples in the relations representing the classes to which they implicitly or explicitly belong. Therefore x is stored twice and y and z are stored three times in our example. As previously noted, this replication makes retrieval much faster, but increases the complexity of the updates which need to keep the replicas consistent.

For example, a COOL update statement of the form

$$\mathbf{update}[f(o) \ := \ expr](o: \ B)$$

which sets the value of the function f of all objects o of the class C to $expr$, is translated to SQL tasks by the internal services using the schema information to determine which classes contain objects with the property f. The replication service differentiates between classes where we only have to propagate the changes on the object copies by a simple update (the so called self-maintenance) and where the update statement may require a change in the class membership. In our example, the update may require that the object z needs to be removed from C and added to E, while the copies of x can be changed in the relations without having to consider a rematerialisation of the class relation. The dependency analyser orders these maintenance tasks in an execution graph similar to the graph shown in figure 4 which then is passed to the transaction management and scheduling components which take care of the efficient and correct concurrent execution. *upd(...)* stands for a simple propagation task, whereas *remat(...)* denotes a rematerialisation operation which depends on the update of its base class. All updates on the self-maintainable classes and classes with a different base class can be performed in parallel which leads to a high degree of parallelism.

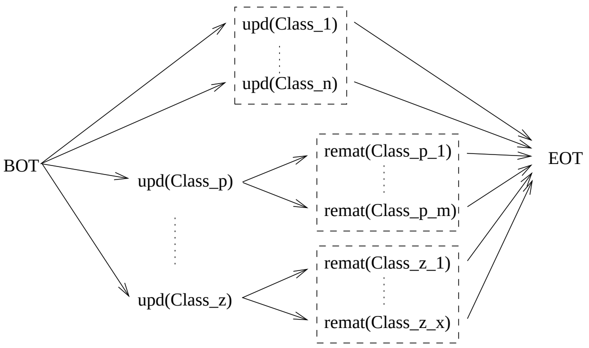

Figure 4: Schema of an update dependency graph

In order to guarantee the correct execution of a COOL transaction, not only the execution order given by the dependency graph has to be guaranteed, but also its atomic and isolated execution. Isolation is achieved by locking the accessed BUTTERFLY objects based on a predicate which is automatically derived from the COCOON schema information and the COOL transaction which requests the lock. Predicates are simplified to a class of predicates which can be handled efficiently by the lock service.

In case that a COOL transaction must be rolled back, some of the tasks executed as transactions on the SQL level might already have been committed. In order to undo the changes which are considered persistent by the relational storage server, so-called *compensating subtransactions* which undo the necessary changes are provided and executed by the log service for each committed task.

The following evaluation was performed on a BUTTERFLY database containing 50'000 objects which were replicated in more than 60 class tables. The measurements were done on a 10 processor Sun SparcCenter 2000 using Oracle 7.1 as the relational database system.

Replication	No Replication
0.3s	6s

Table 1: Typical retrieval performance

The chosen redundant mapping decreases response time of retrievals considerably because less object recomposition and object gathering queries are necessary. For example, table 1 gives the response times for a simple select query on a class for our approach and the standard mapping using vertical partitioning.

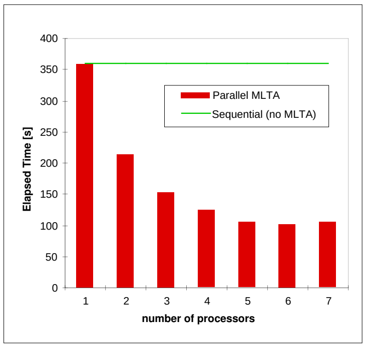

Figure 5: Elapsed time of an update

In the context of large classification structures, the replicated data and materialised views are efficiently maintained with our parallelisation approach. Figure 5 shows the elapsed time for the execution of such an update statement. The horizontal line in figure 5 shows the update time if we use one sequential transaction to keep the replicated objects consistent. If we compare this with the execution using our parallel approach, but only using one processor, we see that the additional overhead (i.e. the logging overhead) is insignificant for the overall performance. If we increase the number of processors, the response time is reduced by a factor of more than 3. Further details on the mapping and the evaluation can be found in [Rys97].

4 TPM/TEXT

TPM/TEXT stands for Transaction Processing Monitor with support for TEXT document management. In this system, textual documents — including the textual index structures, which are implemented using inverted lists physically stored in a binary relation InvList(word, document-ID) — are stored in the underlying relational resource manager. TPM/TEXT extends a conventional transaction processing monitor and provides — among others — client services to insert documents into the system, which indexes the textual attributes on a word–by–word basis, as well as a client service to retrieve documents containing specific words in their textual attributes [KS96]. The architecture of TPM/TEXT is presented in figure 6.

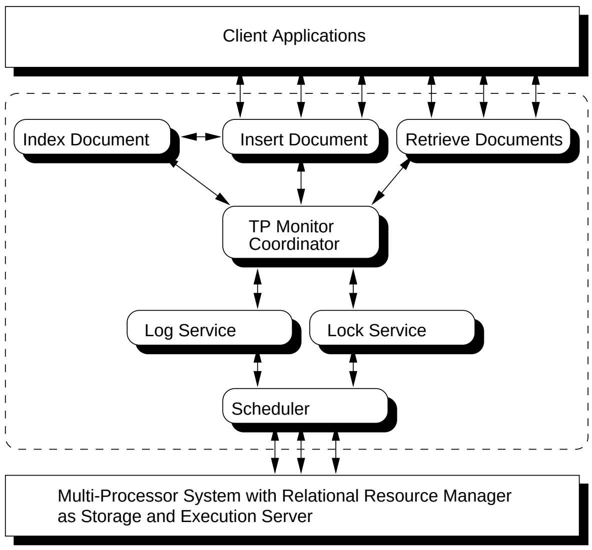

Figure 6: Architecture of TPM/TEXT

All client services and the indexing service are coordinated by the TP monitor. The main difference between a conventional TP monitor and TPM/TEXT is the way client service transactions are executed. In a traditional TP monitor environment, a client service transaction such as the insert of a document consists of a *begin of transaction* call, the execution of the document insert followed by a number of indexing operations and by an *end of transaction* call. All of these operations are executed using a *single* transaction of the underlying resource manager.

Contrary to this, in TPM/TEXT each insert and index service called is executed as an individual transaction of the underlying resource manager. The advantage of this is that these services can be called asynchronously and executed in parallel. As we already noticed in the case of the BUTTERFLY prototype, the unrestricted execution of these services as independent resource manager transactions would result in the violation of the *atomicity* and *isolation* principles: If a transaction fails, one or more index service calls have already been terminated and are committed on the resource level. Changes cannot be undone anymore. Atomicity is violated from a user transaction point of view. As soon as a resource manager transaction commits (i.e. at the end of a service call), changes made by this service are visible to the outside because all locks help by the resource manager transaction are released. Isolation is violated. In our approach we avoid this undesired behavior. In order to further guarantee the ACID properties at the service level, an additional transaction manager has been implemented in a similar way as in the BUTTERFLY prototype.

On the TPM/TEXT level, a predicate oriented lock manager is used, which locks a minimal number of objects. By this, the potential for lock conflicts is reduced. The *atomicity* is guaranteed in a similar way to BUTTERFLY. To compensate changes caused by a service in case of failure, a so-called *compensating service* must be available for each service. In case a client service transaction is aborted, it is rolled back by executing the compensating service for every service executed on behalf of this client service transaction if the service has already committed its changes on the resource manager level.

We have benchmarked TPM/TEXT against an implementation with flat transactions on a SparcCenter 2000 with ten processors using Sybase as resource manager. Table 2 shows the average insertion time as well as the throughput when 1, 5, and 20 concurrent users continuously insert documents (inserting a document also includes the maintenance of the textual index structures, which is in this case 200 inserts into the inverted lists). Looking at the numbers for flat transactions, one can see that they increase almost linearly with the number of concurrent users. This increase is caused

	concurrent users					
	1		5		20	
	resp	thru	resp	thru	resp	thru
flat	5.8	0.17	27	0.18	102	0.19
TPM/ONT	3.0	0.31	10	0.45	39	0.45

Table 2: Insertion of documents (response time (sec) and thruput (documents/sec)

by a lock contention on the textual index structures. When TPM/TEXT is used, the response times can be decreases by approximately 70% as well as the throughput increased by the same percentage. This is because the locks (Sybase's page level locks) on the textual index structures are not held until the end of a transaction but are released early and can then immediately be acquired by any other concurrently executed inserting transaction.

5 Conclusion and Outlook

In this paper we have presented two middleware prototype components, BUTTERFLY and TPM/TEXT, which both use a commercially available relational multi-processor system and provide higher-order objects to the clients. Both system employ replication of data to achieve the fast retrieval of either text documents or database objects. To improve the insert performance of documents and the update performance on the stored objects, the maintenance of the replicated data items (text index entries, replicated object copies) where executed in parallel on the resource manager. To ensure the ACID properties, a higher-level transaction manager was build into the middleware components. While BUTTERFLY implemented all services from scratch, TPM/TEXT was built by extending a commercial TP monitor.

Neither TPM/TEXT nor BUTTERFLY is limited to be applied to non-distributed database systems. In the future, we will investigate extensions of our prototype systems which allows the efficient execution of client services in a distributed database environment such as a network of workstations with commercially available database systems running on each node.

References

[Kau97] Helmut Kaufmann. *Transaktionsorientierte Verwaltung und Suche von Dokumenten in einer Mehrprozessordatenbankumgebung.* PhD thesis, ETH Zürich, 1997. To appear.

[KS96] Helmut Kaufmann and Hans-Jörg Schek. Extending TP-Monitors for Intra-Transaction Parallelism. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, Florida, December 1996.

[RNS96] Michael Rys, Moira C. Norrie, and Hans-Jörg Schek. Intra-Transaction Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB)*, Mumbai (Bombay), India, September 1996.

[Rys97] Michael Rys. *Materialisation and Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System.* Diss. eth nr. 12282, ETH Zurich, 1997.

[SABT] Hans-Jörg Schek, Gustavo Alonso, Stephen Blott, and Markus Tresch. `http://www-dbs.inf.ethz.ch/`.

[SLR+94] Marc H. Scholl, Christian Laasch, Christian Rich, Hans-Jörg Schek, and Markus Tresch. The COCOON Object Model. Departemental Report 211, ETH Zürich, Institut für ETH Zentrum CH-8092 Zürich, February 1994.

[Tre96] Markus Tresch. Middleware – Schlüsseltechnologie zur Entwicklung verteilter Informationssyseme. *Informatik-Spektrum*, 19(5), October 1996. Springer Verlag, Heidelberg.

[WS92] Gerhard Weikum and Hans-Jörg Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 13. Morgan Kaufmann Publishers, Inc., 1992.