Intra-Transaction Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System

Michael Rys Moira C. Norrie Hans-Jörg Schek

Dept. of Computer Science, Swiss Federal Institute of Technology (ETH)
CH–8092 Zürich, Switzerland
{rys, norrie, schek}@inf.ethz.ch

Abstract

We map an object model to a commercial relational multi-processor database system using replication and view materialisation to provide fast retrieval. To speed up complex update operations, we exploit intra-transaction parallelism by breaking such an operation down into shorter relational operations which are executed as parallel subtransactions of the update transaction. To ensure the correctness and recoverability of the operation's execution, we use multi-level transactions. In addition, we minimise the resulting overhead for the logging of the compensating inverse operation required by the multi-level concept by logging the compensation for non-derived data only. In particular, we concentrate on the novel application of multi-level transaction management to efficiently maintain the replicated data and materialised views. We present a prototype implementation and give a performance evaluation of an exemplary set-oriented update statement.

1 Introduction

During the past decade, research in data modelling and database system architecture has led to two new classes of database management systems: object database management systems (ODBMS) and parallel database management systems (PDBMS). ODBMS are based on semantically rich and powerful object models which feature concepts such as object classification and generalisation/specialisation. Such an object model is accompanied by an object algebra which describes the model's generic query and manipulation operations [BM93].

PDBMS support the natural parallelism of database systems based on the concurrent execution of different user transactions on a common database (inter-transaction parallelism) by exploiting the infrastructure of multi-processor and/or multi-disk computer systems. In addition, they increase the performance of the database system by executing operations within a transaction in parallel (intra-transaction parallelism) and by parallelising the execution of these operations (intra-operation parallelism). The performance increase manifests itself either by the capability to process larger datasets in the same amount of time (scale-up) or by the reduction of the response time of a single transaction (speedup). The reduction in response time either can be used to improve the response time for a single user or to improve the overall system transaction throughput [Gra95]. While several commercial systems of either variety are now available, the combination of parallel and object database management systems is still a research topic [Val93].

Many applications which use ODBMS feature rich classification structures and typically execute significantly more queries than updates on the database. A typical example of such an application is a decision support system of a life insurance company [RRSM93]. We combine an ODBMS and a PDBMS to support such a knowledge based decision support system with a parallel object database management system. In particular, a frame based knowledge representation model FRM [Rei89] is mapped to the COCOON object data model and its associated object algebra COOL [SLR+94] which are themselves implemented on top of a commercial relational database management system [RLNR95]. We use a commercial relational database

Proceedings of the 22nd VLDB Conference Mumbai(Bombay), India, 1996

management system as an "intelligent" storage server in order to exploit the mature technology provided by the relational database management system for query optimisation, concurrency control and physical design [NRL+94]. The main load of a decision support system is retrievals. Therefore we increase the access efficiency through the extensive use of replication (see [SC89] for a discussion of its benefits) and view materialisation (see [Han87, GM95, BE96]). Update operations become more complex due to the added redundancy in the mapping of the large classification structures. In order to speed them up, we exploit intra-transaction parallelism by breaking the updates into shorter relational operations. These are executed as ordinary independent parallel transactions on the relational storage server.

This works builds on our previous experience with multi-level transactions [Wei91, WS92a, SSW95]. In particular, we explore and evaluate the key idea of exploiting inter-transaction parallelism at the low level for intra-transaction parallelism at the high level in the context of multi-level transaction management as described in [WH93].

Using parallelism to update derived data is also investigated in the Cactis project [HK89]. In contrast to our approach, Cactis deals with derived attributes of a few, large objects typically encountered in engineering design applications. It does not use an existing parallel database system, nor does it apply a multi-level transaction management concept.

Many object database management systems use a relational database management system as a storage server, examples are: COCOON [TS91], SQL/XNF [MPP+93], OMT [RBP+91], ERIC [LC91], Iris [LK91], HP's OpenODB [AD92], EXTREM [HHRW92], and OPM [CM95]. They choose a "traditional" mapping where the resulting relational schemata are normalised and avoid object replication by using either horizontal or vertical partitioning when mapping the generalisation hierarchy [RBP+91, LV87, HH91]. The disadvantage of the normalised, non-replicated mapping is that retrieve operations need to recompose the objects from the different tables with expensive join operations which we want to avoid. The idea of introducing extensive replication in the mapping of an object model to a relational parallel database (storage) system has, to our knowledge, not been investigated so far, although it seems to be a straightforward approach.

The main contributions of this paper are the following:

- The investigation of extensive replication in the mapping of an object model to a relational parallel database (storage) system.
- The use of multi-level transaction management to efficiently maintain materialised views and replicated data by parallelising the execution of updates is a novel application of the multi-level transaction concept.
- In order to drastically reduce the inherent logging overhead of a two-level transaction manager, we introduce a new logging approach which uses the mapping information from object operations to storage operations, i.e. SQL in our case.

The evaluation of our approach is based on a real prototype implementation.

In this paper, we give an overview of our approach and discuss the parallelisation of the generic update operation. Section 2 introduces our approach and the architecture of our system and gives a short introduction to multi-level transactions. Section 3 gives an overview of the object model COCOON and its mapping to the relational database management system. Section 4 presents the mapping for the object algebra COOL and also introduces our logging strategy. Section 5 discusses evaluation results for the execution of the update operation and discusses the potential and limitations of the approach before we conclude with a discussion of ongoing and future work.

2 General Architecture and Approach

Our work is embedded in a larger project called HYWIBAS [RRSM93] which exploits database technology to support efficient retrieval and update of large frame-based knowledge systems. The general architecture concerning the database support is shown in figure 1. FRM is the frame based knowledge representation language which is being used to build the life insurance decision support systems. FRM is a terminological logic [WS92b] which features two kinds of relations, namely properties and semantic relationships, and a powerful classification mechanism. It is mapped to the ODBMS layer with a data model featuring specialisation and classification structures. In our case, COCOON is the data model used. Object operations and transactions are mapped to SQL transactions of the parallel database system (PDBMS). In our case this is Oracle.

The main components for the mapping are the *Schema Definition Translator* which does the physical design on the relational tables (see also section 3), the *Update Operation Translator* and the *Query Operation Translator* which both translate the COOL operations based on the physical design. The COCOON object layer acts as a client to the relational storage server. In order to be able to execute several transactions in parallel from a single transaction, we have to build

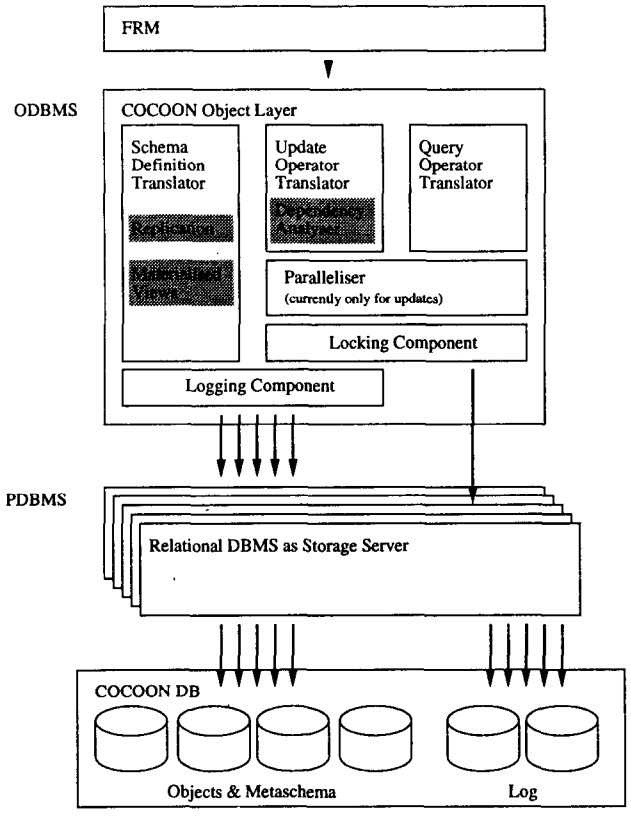

Figure 1: The overall system architecture

an extra layer on top of the client/server interface of the relational database system, because current systems do not support several parallel transactions from a single client. This subcomponent (the *Paralleliser*) is similar to a transaction processing monitor in that it accepts a set of transactions and a description of their dependencies, executes them in parallel on the underlying relational system and returns a single return code to the actual client. Finally, the *Logging* and *Locking Components* implement the level specific multi-level transaction management. In this paper, we concentrate on the schema and update operation translators and on the logging component. The locking component is similar to the lock managers described in [Has95, SSW95]. Details about the query translator, the paralleliser and the lock manager are beyond the scope of this paper.

As we mentioned above, the reduction in retrieval costs resulting from our mapping is achieved at the expense of increased complexity of update operations due to the data replication and view materialisation. A COOL update operation now has to update and reclassify the different replicated and materialised copies of the updated objects in all the different locations introduced by the mapping. Instead of deferring the updates to the retrievals and slowing them down, we immediately propagate the changes to the replicated

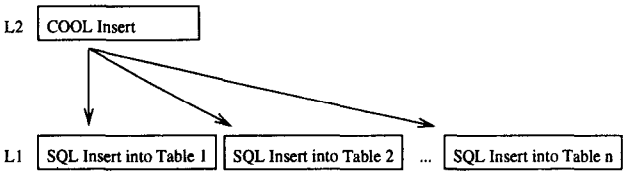

Figure 2: A multi-level transaction hierarchy

objects. However, to avoid long transactions with lots of updates which are not only slow, but may also hinder queries because of lock contention, we apply intra-transaction parallelism by executing the many resulting relational updates in parallel [Has95].

As an example for a COOL update operation, consider the insertion of objects into classes on the object model level. These objects now have to be inserted into all the SQL tables where they belong because of the replication of the objects within the classification structure and the materialisation of the classes. All SQL insert operations have to be executed, however their execution order does not matter. In particular, these inserts can be performed in parallel as individual SQL transactions. The available concurrency control mechanisms of the advanced storage server (i.e. the relational database system) can be used to execute the SQL operations successfully on the lower level. However, in order to assure the correctness of the main insert operation, additional concurrency control mechanisms have to be used on that higher level. By applying multi-level transaction management [WS92a], we can use its formal foundation to correctly and efficiently parallelise the COOL update operations.

Multi-level transactions are a variant of open nested transactions where the subtransactions correspond to operations at particular levels of abstraction in a layered system (see figure 2 for the hierarchy for the given example). Each level exploits the semantics of the level specific operations for concurrency control by reflecting the commutativity (compatibility) of the operations in level-specific conflict relations. In the example, the COOL insert operation is on level L2. It commutes with another COOL update operation if the set of inserted objects is disjoint with the other set of updated objects. Similarly, the SQL operations on level L1, where they form level L1 transactions, can commute with other SQL transactions with which they do not conflict. Such a conflict test can be performed by a predicate based lock manager such as the ones described in [Has95, SSW95]. In our case, the level L0 operations are hidden within the relational database system. In short, the operations of a transaction of level L*i* form transactions on level L*i-1*. Multi-level transaction management allows more concurrency compared to conventional (single-level) transaction management. However, transaction aborts

can no longer be implemented by restoring the before-image of the modified low-level objects, since certain modifications become visible to concurrent transactions at the end of each subtransaction. The solution to this problem is to perform the aborts by means of high-level inverse operations that compensate complete subtransactions.

Thus, the intra-transaction parallelism at L2 is transformed into inter-transaction parallelism on level L1, the parallel relational database management system. Therefore we can exploit the capability of the multi-processor database system to efficiently execute many short transactions in parallel.

3 Mapping of the Object Model to SQL

In this section, we describe the basic principles of the COCOON mapping to a relational database management system, and discuss our physical design.[1] Note that the approach is general and could also be adopted for other object data models as well.

3.1 The COCOON Object Model

The COCOON data model differentiates between the structural description of objects and the semantic grouping of objects. The structure is described by a type system, while classes are used to group the objects. In both cases an object can be multiply instantiated, i.e. it can be of several different types and it can be member of several different classes at the same time. If objects change properties during updates which influence their classification, they are dynamically reclassified. Thus, COCOON features objects, property functions, types and classes as main concepts. Using the example given in figure 3, we now describe how these concepts occur in COCOON.

COCOON objects have properties which are modelled by functions. The function definitions implicitly introduce the definition of the object types. In the example, the functions *name*, *age*, *sex*, *salary* and *dept* define the object type *employee*.

Subtyping defines a new abstract object type by deriving its definition from one or more existing types (*inheritance*) and adding new functions. For example, *projleader* is a subtype of *employee* and adds the function *projects* to the inherited functions. Each instance of a subtype is also instance of its supertypes (*multiple instantiation*).

In COCOON, types describe the functions which are applicable to an object, but they do not group objects into collections which then can be queried.

[1]For the purpose of this paper, a subset of the actual COCOON data model is presented.

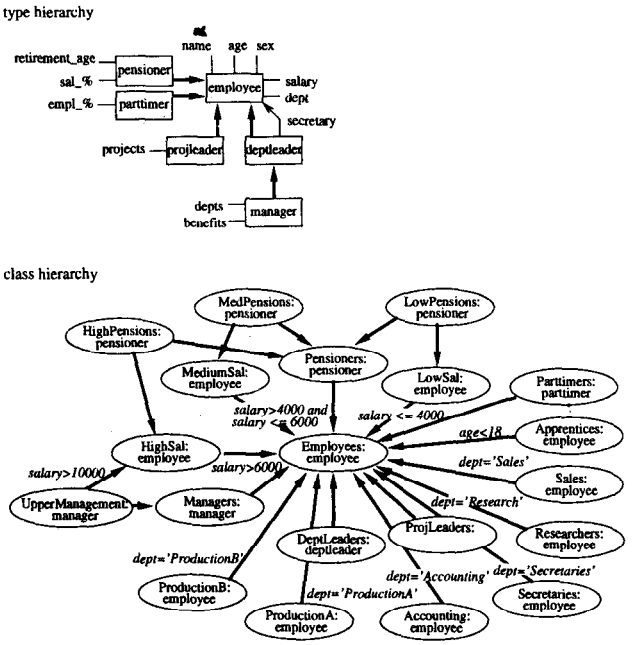

Figure 3: An example schema

To form such collections of objects, COCOON provides a class construct. The *extent* of a class is the set of objects of the associated abstract object type (the *member type*) which fulfil a condition referred to as the *class predicate*. For example, the class *Researchers* has the member type *employee* and the predicate *dept='Research'*. The classes can be considered polymorphic since, due to multiple instantiation, objects contained in the extents are instances of the class member type and its subtypes.

A subclass relationship between two classes implies that the member type of the subclass is a subtype of (or the same type as) the member type of the superclass and the extent of the subclass is a subset of that of the superclass. There are two forms of subclass relationship: **all** classes and **some** classes. **All** classes are subclasses which automatically contain all members of their superclass(es) if they are of the subclass' member type and satisfy the optional associated class predicate; in other words, the subclass membership conditions are both necessary and sufficient. The class *Researchers*, like all other classes in the classification structure of figure 3, is an **all** class and therefore automatically contains all objects of *Employees* if they work in the research department. Note, that **all** classes can be regarded as a partial precomputation of frequently asked queries. **Some** classes are subclasses where the type and the predicate condition are only necessary but not sufficient. Members of such subclasses have to be inserted explicitly. We will concentrate on **all** classes for the remainder of this paper.

There is a predefined root class of the class hierarchy, **Objects**, which contains all existing objects. Thus, the extent of **Objects** is the active domain of the type **object**. All other classes contain only objects which are specified by the subclass relationships and by the class predicates. A class contains all members of its subclasses.

3.2 Mapping of COCOON to a RDBMS

Generally, a COCOON type and class are mapped to a relation schema, while an object is mapped to one or more relation tuples. An object will be represented in all of the relations corresponding to the types of which it is an instance. It will also be represented in all of the relations corresponding to the classes to which it belongs. In this way, direct access to objects of a certain type and of a class is attained. The internal object identifier is implemented by a logical key attribute *COOL_OID* which serves as the primary key in all of the relations representing the types or classes.

Each abstract object type is mapped to a main relation table which besides the logical key attribute, contains all single-valued functions as attributes. Each multi-valued function is stored in a separate binary function table. The subtype hierarchies are mapped with horizontal and vertical replication. This means that all single-valued function attributes are repeated in the subtype's relation table (*vertical replication*) and all instances of a subtype are also stored in the table of the supertype (*horizontal replication*). Using this type mapping, all the attributes of the type table *Employee* are repeated in the type table *Deptleader* (vertical replication) and a *deptleader* object is stored in both tables (horizontal replication).

A class is mapped to a table which in the case of **all** classes is of the same form as its member type's main relation table: In addition, the classification hierarchies are horizontally replicated, i.e. the member objects of the classes are stored in all the tables of the classes of which they are member. Note, that **all** classes can be considered as materialised views, since they could also be mapped as views of their closest **some** class(es) (i.e. their base class(es)). For example, the **all** classes *Researchers* and *UpperManagement* are mapped to tables which are structurally equivalent to the tables of their member types *employee* and *manager*.

In addition, a metaschema contains the necessary meta information such as superclass relationships or a class' member type. Also, the metaschema contains all the information for each class which is necessary to materialise its extent.

Based on this mapping, we can identify the following *derived data*:

- An **all** class table is a materialised view of its member type and base class tables.
- A type table which represents a subtype vertically replicates the inherited attributes.
- The members of a type table which represents a subtype, or of a class table, which represents a subclass, are horizontally replicated in the supertype or superclass tables.

Another important point to consider is the placement of the objects on the disks. Since object tuples are updated in parallel, there will almost certainly be a bottleneck when writing to the disk. In order to minimise the I/O-boundedness of the update operations, the data has to be distributed across different disks such that the object tuples can also be written in parallel. Our current implementation distributes the data using a disk array by horizontally striping the data across several disks. This means that tuples of the same table are on different disks which can improve the I/O access time when accessing objects from the same table. Another way to improve the I/O access time is to allocate the tables on different disks by different placement heuristics. However, our investigations [Blu96], in which we used the anticipated size of the tables and their semantical relationship to achieve load balancing, show that there is no significant gain over the horizontal striping.

4 Mapping of COOL Operations

In this section we give an overview of the COCOON object algebra and an introduction to the mapping of a generic update operation of the algebra using intra-transaction parallelism.

4.1 COOL Operations

The operations over COCOON databases are expressed in the language COOL which is founded on a set-oriented, generic, orthogonal and closed algebra formally defined in [SLR+94]. Generally, COOL query operations such as **select** and **project** work on sets of objects (i.e. classes) and have object-preserving semantics, such that their results are subsets of the existing objects in the database.

Object generation in COOL is achieved by operations such as **create** which creates an object but does not explicitly associate it with any classes or **insert** which creates an object and inserts it into the specified class and into all classes where it is classified according to the class predicates. The update operations can be divided into operations for assignment (**set**), for object evolution (**gain**, **lose**, **delete**) and for manipulating the extents of classes and views (**add**, **remove**,

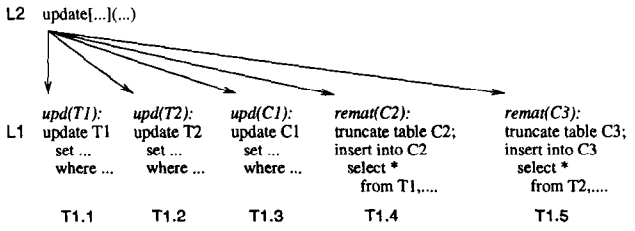

Figure 4: A COOL update transaction hierarchy

update). Due to the dynamic reclassification of objects into **all** classes, an update statement may initiate the automatic propagation of objects within the class hierarchy. In the case of an object evolution operation, objects might also be propagated dynamically within the type hierarchy. For example, the update operation

$$\textbf{update}[f_1(o) := e_1, ..., f_n(o) := e_n](o\text{: } \textit{set-expr});$$

changes the functions $f_1, ..., f_n$ for every object o of the *set-expr* such that the function application $f_i(o)$ results in the value of the expression e_i.

With the physical design introduced in section 3 and the goal to minimise the query overhead, the update operations have to update the data in several tables immediately. A COOL update operation on level L2 consists of several subtransactions, each representing one SQL transaction which performs the necessary update operations on exactly one of the SQL tables on level L1. Since most of the SQL level updates can be performed independently from each other, we would like to speed up the update operations by parallelising the SQL updates. Figure 4 gives an example of such a transaction hierarchy for the **update** statement. We get similar transaction hierarchies for the other set oriented update operations, which then can be parallelised in an analogous way. In this paper, we use the **update** statement as the example to present and evaluate our approach.

4.2 The Update Statement and Rematerialisation Strategies

The mapping of the update statement identifies all types which contain the affected function. It then collects all classes whose member type belongs to the set of the identified types. This results in a set of tables containing all the tables which may contain objects which have to be updated.

If the type table contains a subset of the objects to be updated, the SQL update statement can be expressed by selecting the affected objects without referencing the base table of the set expression. Otherwise a join with the set expression's base has to be performed.

Because classes are mapped as materialised views, the updates of the class tables need to perform the

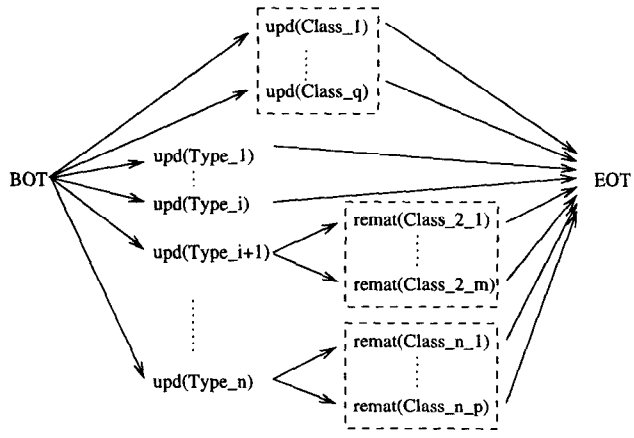

Figure 5: The generic update dependency graph

rematerialisation from the base tables. There are different approaches to implement the rematerialisation [GM95, BE96]. First, we can differentiate between those classes where the update operation may change the extent and those classes where the extent stays the same and only the state of the contained objects may change. In the first case, the classes either have to be incrementally rematerialised by explicitly deleting and inserting the qualifying objects. Otherwise a full rematerialisation has to be performed. If the function affected by the update does not influence the classification, it is sufficient to only update the changed attributes in the class tables.

Thus, these *self-maintainable* class tables can be updated independently (and therefore in parallel) from their base tables. The other class rematerialisations have to be performed after the update of the base class tables and after the member type table on which the classes depend. This execution order assures that the class extents are correct when the views are rematerialised on the changed base tables. In addition, if the extent of the set expression may change due to class maintenance, all updates on tables which reference the set expression's base table, such as an update on a type or self-maintainable class table, have to be performed before the specific class rematerialisation.

Therefore, the execution order of the L1 transactions has to guarantee the dependencies. This is done by the dependency analyser which generates a dependency graph of the L1 transactions (see figure 5) which then is passed to the paralleliser (see figure 1).

4.3 Logging and Recovery Strategy

If a COOL transaction has to be aborted, not yet committed subtransactions will be aborted by the RDBMS. However, since subtransactions are committed on the RDBMS before the COOL transaction is committed, it is necessary to compensate the committed subtransactions. According to the multi-level

transaction theory, compensating inverse operations have to be executed which semantically undo the effects of the subtransactions. To guarantee the correctness of the compensation, atomicity at the SQL operation level is required; this is achieved by executing the inverse operations as compensating subtransactions.

In the naïve approach when we do not make use of our knowledge of the data model's semantics and the execution precedences specified in the COOL transactions dependency graph, the standard inverse operations are as follows (see also [SSW95, Has95]): A committed insert subtransaction has to be undone by deleting the inserted tuples and deleted tuples have to be reinserted. Updated tuples have to be reset to their old values by either deleting and reinserting the changed tuples or, preferably, by updating the tuples to the old values. This naïve approach would cause too much overhead: For each insert, delete or update statement, we would almost double the work by generating another insert for the logging.

In order to significantly reduce the overhead of the additional log, the logging component optimises the log data by applying its knowledge of the semantics of the COCOON data model and its mapping to the relational database system. The main idea is that undo operations will be logged only on tables containing data which cannot be derived from other tables. Such tables which need to be logged are called primary tables. Compensating operations for operations on tables containing derived data do not need to be logged, but can be derived when needed. The goal is to employ this in such a way that logging is minimised without losing the ability to perform recovery, or a (partial) rollback, of a transaction by compensating (some of) its subtransactions. To achieve this, we need to ensure that we have the logged information on the status of the running and already committed subtransactions to perform the necessary redo (recovery) or undo (compensation) operations on the derived data.

With the vertical and horizontal replication between the tables of type T_i and its supertypes $T_1..T_{i-1}$ and subtypes $T_{i+1}..T_n$, only the operations on T_i have to be logged for operations which primarily affect T_i (i.e. T_i is the member type of the set expression).

In the case of an **all** class table, an undo of a committed change can be done by rematerialising the class table after its member type and base class tables has been undone.[2] If the **all** class is self-maintainable or if we compensate an update on an unlogged type table, we propagate the old object state from either the log or directly from the primary table depending on the status of the subtransaction on the primary table.

If a crash happens after changes on an unlogged table have been committed but the changes of the primary table were lost, the recovery process can simply restart the not yet committed subtransactions.

Therefore, compensating operations only need to be logged for the primary type tables. For the inverse of `insert`, only the deletion based on the primary key attributes (i.e. the object identifier) need to be logged. The inverses of the `update` and `delete` statements have to log the complete undo operation for the affected tuple.

With this logging strategy, the mapping of the update statement looks as follows:

The COOL update operation first operates in parallel on the type tables and on the self-maintainable class tables. The compensation log is only written for the primary type table. A subtransaction of the class rematerialisation phase starts as soon as all of the subtransactions on which it depends have committed. Since there is no dependency between the class rematerialisation subtransactions, they can be executed in parallel.[2]

The information required to perform the inverse operation has to be written to a persistent log using the WAL (write ahead log) rule. We exploit the database functionality for keeping the log data persistent by implementing the log as database tables. One table contains the information about the state of each subtransaction, while the other table contains the undo operations. While in principle all log data can be written to the log table by database triggers which are automatically generated for each operation type and for each table as soon as the table is created, this can lead to a severe performance penalty with the current implementation of triggers in some of the commercially available database systems. Instead, an SQL statement can be generated which writes the log to the table shortly before or after the execution of the operation.

Since the mapping of a COOL operation always results in the same set of subtransactions as long as there were no schema changes, it might appear sufficient to just log the COOL operation and generate the inverses of the subtransactions during the rollback. This approach however can lead to incorrect results, if the inverse of a subtransaction is not defined. For example, if we increment the salary of all employees within a certain salary range, the inverse cannot be derived automatically, since the decrement may also affect objects which were not affected by the increment. Therefore we still need to log the inverse operations of the subtransaction on the primary table. Note that redo information for the derived data also does not need to

[2] For reason of simplicity, we do not discuss the case of classes with predicates referencing other classes or other types (via an inverse function).

be logged on the lower level by the relational system. Unfortunately, no commercial database management system currently allows such non-persistent transactions. In Oracle, **truncate table** operations are not logged, therefore we use them in the rematerialisation transaction.[3] However, the insert statements following the truncation are still logged by the database system.

5 Evaluation

In this section, we present some results of our experimental evaluations for two typical COOL update and two typical retrieve operations. The schema given in figure 3 was augmented by additional classes which partition the existing classes into male and female members. Then a database was generated with this schema which had a size of approximately 75 MB and contained 50'000 objects of the type *employee*.

The measurements were performed on a Sun SparcCenter 2000 shared memory multiprocessor system with ten processors running under Solaris 2.5. The relational database system used was Oracle 7.1.3. The tables of the mapped schema were all allocated on a tablespace which was striped over seven (raw) disks of 50 MB each. The log tables were allocated on a different tablespace which was striped over two different (raw) disks of 100 MB each. The striping granulate was 40 kbytes which corresponds to one disk track. The internal rollback segments were also striped over two disks. The internal redo logs were allocated on different disks while the temporary tablespace shared a disk with the data tablespace. The database buffer was set to 500 blocks with a database block size of 4 kbytes which resulted in an average buffer hit ratio of 98.5%. The size of the shared pool, which is used by Oracle to store session information such as sort areas and triggers, was set to 20MB and the size of the log buffer to 4MB to minimise the influence of Oracle internals on the measurements. The system was running in multi-user mode, however the measurements were performed while no other users were on the system in order to exclude influences from the outside. The measurements were repeated until a 95% confidence level with a confidence interval of ±5% around the average or better was reached.

5.1 Retrieval Performance

In order to investigate the benefit of using replication to speed up the retrieve operations, we investigated the following two simple COOL statements:

#(**select**[*sex* = "*m*"] (*MedPensions*)) (S1)
#(**select**[*sex* = "*m*"] (*MediumSal*)) (S2)

[3]Note that **truncate table** is considered a schema operation by Oracle and therefore forces a commit of the running transaction. Therefore it can only be used in single user investigations.

Table 1: The generated SQL statements

S1	
replicated mapping	**select** count(*) **from** *MedPensions* **where** *sex* = 'm'
vertical partitioning	**select** count(*) **from** *Pensioner N1, Employee N2* **where** *N1.COOL_OID=N2.COOL_OID* **and** *N2.sex* = 'm' **and** *N2.salary* **between** 4001 **and** 6000
S2	
replicated mapping	**select** count(*) **from** *Mediumsal* **where** *sex* = 'm'
horizontal partitioning	**select** count(*) **from** (**select** *COOL_OID* **from** *Employee* **where** *salary* **between** 4001 **and** 6000 **and** *N2.sex* = 'm' **union select** *COOL_OID* **from** *Parttimer* ...)

Table 2: Retrieve performance

	Elapsed Time		CPU Time		physical IO	
	repl	nonrepl	repl	nonrepl	repl	nonrepl
S1	0.1s	8.6s	0.1s	7.5s	23	579
S2	1.1s	8.0s	0.7s	6.1s	294	510

S1 retrieves the number of all male pensioners with a medium sized pension while S2 retrieves the number of all male employees with a medium income. Note that these queries could also be formulated directly on the classes which contain the requested employees. S1 is used to compare our replicated mapping with the traditional mapping using vertical partitioning, while S2 is used to compare our approach with the mapping using horizontal partitioning (see Table 1 for the generated SQL statements). Note that each table had a primary key index specified on the *COOL_OID* attribute.

Table 2 gives the measured elapsed and CPU times and the physical reads for a single execution of the statements. Even for such simple statements, our replicated mapping results in performance improvements of one to two orders of magnitude and a significantly lower number of disk accesses. Due to the lower number of disk accesses, the data more likely fits into the database buffer. Therefore, we have to expect a higher buffer miss ratio in addition to the additional computational effort in the non replicated case. When we add an index on the *salary* attribute of *Employee*, S1 in the non-replicated case chooses a different plan, however the resulting elapsed response time improves only by about 15%.

Table 3: Log execution time

	Logged Sub-TA	Log Time	Exec Time of Unlogged Sub-TA
U1	`upd(Pensioner)`	2.3s	3s
U2	`upd(Employee)`	13s	29s

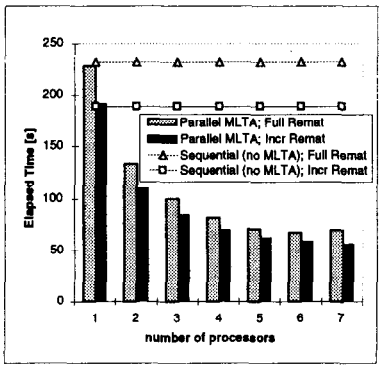

Figure 6: Elapsed time U1

5.2 Update Performance

In the following, we present our results on the main question, namely to what extent updates can be made faster. Based on the statements

$$\textbf{update}[salary := salary + 500](MedPensions) \quad \text{(U1)}$$
$$\textbf{update}[salary := salary + 500](MediumSal) \quad \text{(U2)}$$

we will discuss the performance and applicability of the parallelisation of the update operations and the differences between the two investigated statements. We measured elapsed time and raw CPU time (user and system time) for the execution with multi-level transaction management for 1 to 7 processors. As a reference point, we measured the same statements also on one processor in a single transaction implementation without logging and with the same physical design. In addition, we used an incremental rematerialisation strategy as well as full rematerialisation for both update statements. The statement U1 changes the salary of 5% of the database population while U2 changes 54%.

Table 3 gives the overhead of the log, which is written by the `upd(Employee)` subtransaction for U2 and the `upd(Pensioner)` subtransaction for U1. While the log overhead accounts for a substantial part of the subtransaction's execution time, it is almost negligible if we compare it with the total execution time of the update statements (see discussion of elapsed time below).

Figures 6 and 7 show the elapsed response time of the statements U1 and U2, respectively. The horizontal lines indicate the response times for the sequential executions without multi-level transactions which are compared to the response times achieved with the parallel executions.[4] While one would expect that the execution of the same statement with multi-level transaction support on one processor takes longer because of the logging overhead, the measurements show different. A probable reason for this is that the additional overhead for Oracle to handle a long transaction is large enough to make the difference insignificant, even when no other transaction is present in the system (note that the expected difference due to the additional logging (U1: 1%, U2: 3.6%) was slightly larger than the confidence interval for the measurement of the sequential execution (U1: 0.5%, U2: 2.6%)). Using six processors we are around 70% faster than in the sequential case for all 3 cases.

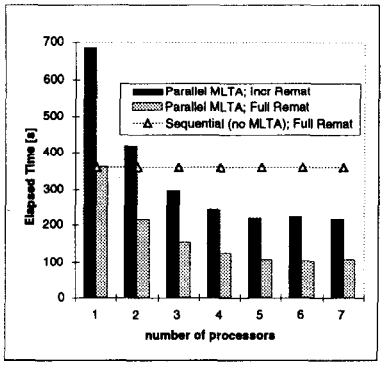

Figure 7: Elapsed time U2

Figure 6 also shows that for a low selectivity of the update statement the incremental rematerialisation strategy must be preferred over full rematerialisation. Figure 7 on the other hand shows that for a high selectivity the incremental approach is unfeasible mostly because of the higher deletion costs. While these measurements alone do not give a conclusive answer to the question of when the incremental strategy is better than the full, it indicates (together with some random samples taken) that the incremental strategy is favourable if less than about 10%–15% of the objects of a class have to be rematerialised.

The increase in performance is also shown in figure 8 which shows the achieved speedup compared to the single processor, single transaction execution for both elapsed and CPU time.[4] In the case of six processors, for example, we get a speedup of 3.53 for the elapsed time and 4.25 for the longest running processor's CPU time for U2. The difference between the elapsed and CPU time speedup is mainly because of

[4]For the single execution of U2 with incremental rematerialisation the measurements could not be performed with the given systems resources because the internal rollback segments were not large enough.

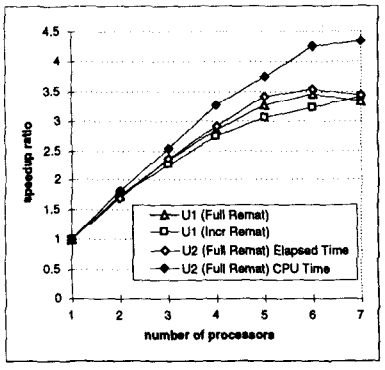

Figure 8: Speedup U1 & U2

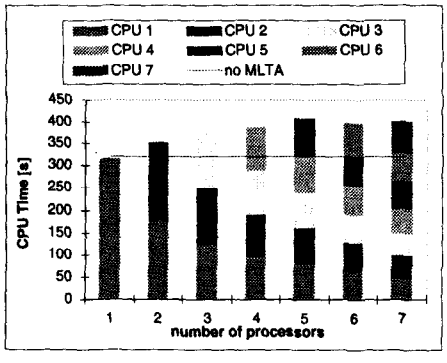

Figure 9: CPU resource usage U2 (Full Remat)

the increased synchronisation overhead on the lower level. Using seven processors does not further improve the speedup in this case because of the increased low-level synchronisation overhead and the sequential execution of the dependent subtransactions. For U2, the sequential execution of the longest subtransaction of the first phase (`upd(Employee)`) and the longest subtransaction of the second phase (`remat(MediumSal)`) increases from 62s for one processor to 97s for seven processors due to the low-level synchronisation. This is only about 7s less than the seven processor execution of the whole update statement. Therefore we can expect that we can still benefit from a higher degree of parallelism for larger classification structures or when we further parallelise the subtransactions. Both rematerialisation strategies result in a similar speedup behaviour for the statement U1.

Based on the total CPU time for U2 used by all processors as shown in figure 9, we notice that we have to invest around 22% more CPU time compared to the single processor case because of the additional low-level synchronisation which has to be done by Oracle. Figure 10 shows that our scheduler produces an excellent load balance of the subtransactions up to five processors. After that the distribution of the workload becomes worse because there are not enough subtransactions to balance the longer running subtransactions.

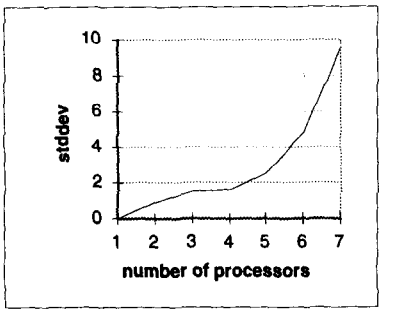

Figure 10: Standard deviation of CPU load balance U2 (Full Remat)

In addition, we can see that Oracle is quite capable to cope with an increased transaction load, because the CPU processing overhead seems to remain stable for an increase in the number of concurrent transactions.

We also investigated for U2 the case where an index was defined on the *salary* attribute of the *employee* type table. This slows down the update of the type table itself because of the index maintenance. On the other hand, the use of the index can speed up the rematerialisation of classes with predicates of high selectivity based on this attribute. In our example, only the *UpperManagement* and the *Pensions* classes have a high enough selectivity to benefit from such an index. Otherwise the cost based optimiser still uses a scan and the index is not used. In our investigated example, the index maintenance costs during the update were too high. Thus, the resulting response times were always worse than the response time without the additional index.

Our results show that there is a benefit in parallelising update statements using multi-level transactions. Especially if objects have to be reclassified in a large classification structure, there is a significant benefit in the parallelisation. As has been shown in [Has95], the faster execution of the update statements also leads to a reduced conflict potential with concurrent retrieves due to shorter locks.

Finally let us compare the execution time of our approach with the traditional, nonreplicated approach using vertical partitioning. In this case only the type table *employee* which contains the changed attribute has to be updated. We measured 16s for this update for U1 and 30s for U2 when executed as a single-level transaction. With our approach, we are about 3 to 4 times slower (see figures 6 and 7). On the other hand, as we have shown above, even a simple retrieve statement benefits from the replication. For example for the statement S2 and the update statement U2 we benefit from the replication approach as long as the number of retrievals is more than 9 times the number of updates.

In summary, we have shown that significant performance improvements can be achieved for retrievals by a mapping using replication and view materialisation. Using our multi-level transaction approach to parallelise the update operations, we can again benefit significantly compared to a nonparallel approach, especially for large and complex classification structures.

6 Conclusion and Outlook

In this paper, we presented an implementation of an object model on a parallel multi-processor system suited to applications such as decision support systems which typically have complex classification structures and many more retrievals than updates. In the implementation we use replication to increase query performance and apply intra-transaction parallelism to execute the updates efficiently. We did not change the relational multiprocessor database system, but used a commercially available system. Nevertheless, by using multi-level transaction management on top of to the existing database transaction manager to map the intra-transaction parallelism to inter-transaction parallelism, we can obtain significant performance improvements for ODBMS update operations. Our experimental investigations show that this approach combined with an "intelligent" logging strategy compares favourably to a single transaction implementation and to the traditional nonreplicated implementation.

In addition, we identified certain characteristics which an "intelligent" storage server should provide in such a layered system. In this paper, we mentioned the benefit of switching off the low-level logging for certain storage server transactions which handle derived data. This avoids unnecessary logging activities on the lower level which improves the performance of our high-level parallelisation. Based on the performance evaluation, we also think that there is a need for more efficient trigger execution if triggers should be used for high performance work.

We were investigating intra-transaction parallelism for only a single high-level transaction. If we also take inter-transaction parallelism at the COCOON level into account, we of course lose some of our benefits due to lock conflicts on the COCOON level and the internal Oracle level. Some investigations concerning the general effects of parallel transactions at the higher levels are reported in [Has95]. We intend to build on this work in future investigations of our approach.

Further investigations are required to generalise the factor for which the replication approach outperforms the traditional approaches. More work also needs to be done in comparing the two mentioned rematerialisation approaches for finding a heuristic approach to chose the better strategy. In the future, we will also investigate the benefits of lazy instead of eager evaluation for the rematerialisation of rarely accessed class tables and try to develop a heuristic to dynamically choose between the eager and the lazy approach based on the current system characteristics.

Our approach of extensive replication for increased retrieval efficiency, together with high-level parallelisation of update operations, is a new promising direction which is more generally applicable. Another area where it could be applied is in distributed environments where replication is a popular approach to increase query performance or in data warehouse systems [LW95].

Acknowledgement

This work was supported by the Swiss Priority Programme in Computer Science under Grant No. 5003-34347. The authors would also like to thank Christof Hasse, Gustavo Alonso and the anonymous referees who gave helpful comments on earlier drafts of the paper.

References

[AD92] R. Ahad and D. Dedo. OpenODB from Hewlett-Packard: A Commercial Object-Oriented Database Management System. *Journal of Object-Oriented Programming*, 5(1):31–35, 1992.

[BE96] D. Botzer and O. Etzion. Optimization of Materialization Strategies for Derived Data Elements. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):260–272, April 1996.

[Blu96] U. Blum. Implementation und Evaluation verschiedener Heuristiken zur Allokation von Tabellen auf verschiedenen Disks bei der Abbildung von COCOON Typen und Klassen. Diploma work, ETH Zürich, March 1996.

[BM93] E. Bertino and L. Martino. *Object-Oriented Database Systems.* Addison-Wesley, 1993.

[CM95] I.A. Chen and V.M. Markowitz. OPM Schema Translator 3.1. Technical report LBL-35582, Lawrence Berkeley Laboratory, March 1995.

[GM95] A. Gupta and I.S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *Bulletin of the Technical Committee on Data Engineering*, 18(2):3–18, June 1995.

[Gra95] J. Gray. A Survey of Parallel Database Techniques and Systems. Tutorial at the 21st International Conference on Very Large Data Bases, September 1995.

[Han87] E. N. Hansón. A Performance Analysis of View Materialisation Strategies. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 440–453. ACM, 1987.

[Has95] C. Hasse. *Inter- und Intratransaktionsparallelität in Datenbanksystemen: Entwurf, Implementierung und Evaluation eines Datenbanksystems mit Inter- und Intratransaktionsparallelität.* Diss ETH Nr. 11045, ETH Zürich, 1995.

[HH91] C. Hörner and A. Heuer. EXTREM - The structural part of an object–oriented database model. Informatik-Bericht 91/5, TU Clausthal, October 1991.

[HHRW92] A. Heuer, C. Hörner, H. Riedel, and U. Wiebking. Syntax, Semantics, and Evaluation of the EXTREM Object Algebra. Informatik-Bericht, TU Clausthal, 1992.

[HK89] S. E. Hudson and R. King. Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System. *ACM Transactions on Database Systems*, 14(3):291–321, September 1989.

[LC91] T. Learmont and R.G.G. Cattell. An Object-Oriented Interface to a Relational Database. In K.R. Dittrich, U. Dayal, and A.P. Buchmann, editors, *On Object-Oriented Database Systems*. Springer, 1991.

[LK91] P. Lyngbaek and W. Kent. A Data Modeling Methodology for the Design and Implementation of Infromation Systems. In K.R. Dittrich, U. Dayal, and A.P. Buchmann, editors, *On Object-Oriented Database Systems*. Springer, 1991.

[LV87] P. Lyngbaek and V. Vianu. Mapping a semantic database model to the relational model. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 132–142. ACM, 1987.

[LW95] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing*, volume 18(2) of *Bulletin of the Technical Committee on Data Engineering*. IEEE Computer Society, June 1995.

[MPP+93] B. Mitschang, H. Pirahesh, P. Pistor, B. Lindsay, and Südkamp N. SQL/XNF – Processing Composite Objects as Abstractions over Relational Data. In *Proceedings of the 9th Data Engineering Conference*, pages 272–282, Vienna, April 1993. IEEE, IEEE.

[NRL+94] M. Norrie, U. Reimer, P. Lippuner, M. Rys, and H.–J. Schek. Frames, Objects and Relations: Three Semantic Levels for Knowledge Base Systems. In *Proceedings of the Workshop "Reasoning about Structured Objects: Knowledge Representation Meets Databases" of the 18. Fachtagung für Künstliche Intelligenz*, Saarbrücken, September 1994.

[RBP+91] J. E. Rumbaugh, M. R. Blaha, W. J. Premerlani, F. Eddy, and W. Lorensen. *Object–Oriented Modelling and Design*. Prentice-Hall International, Inc., 1991.

[Rei89] U. Reimer. *FRM: Ein Frame-Repräsentationsmodell und seine formale Semantik. Zur Integration von Datenbank- und Wissensrepräsentationsansätzen*. Springer, 1989.

[RLNR95] U. Reimer, P. Lippuner, M. C. Norrie, and M. Rys. Terminological Reasoning by Query Evaluation: A Formal Mapping of a Terminological Logic to an Object Data Model. In G. Ellis, R.A. Levinson, A. Fall, and V. Dahl, editors, *Proceedings of the International KRUSE Symposium: Knowledge Retrieval, Use and Storage for Efficiency*, pages 49–53, University of California at Santa Cruz, August 1995.

[RRSM93] U. Reimer, M. Rys, H.–J. Schek, and R. Marti. Datenbankbasierung eines Frame–Modells: Abbildung auf ein Objektmodell und effiziente Unterstützung komplexer Operationen. Technical Report 5/93, Informatik–Forschungsgruppe, Rentenanstalt/Swiss Life, Zürich, August 1993.

[SC89] E. J. Shekita and M. J. Carey. Performance Enhancement Through Replication in an Object-Oriented DBMS. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, volume 18(2) of *SIGMOD Records*, pages 325–336, June 1989.

[SLR+94] M. H. Scholl, C. Laasch, C. Rich, H.–J. Schek, and M. Tresch. The COCOON Object Model. Technical Report 211, ETH Zürich, Departement Informatik, February 1994.

[SSW95] W. Schaad, H.–J. Schek, and G. Weikum. Implementation and Performance of Multi–level Transaction Management in a Multidatabase Environment. In *Proceedings of the 5th Internat'l Workshop on Research Issues on Data Engineering: Distributed Object Management*, Taipeh, Taiwan, 1995.

[TS91] M. Tresch and M. H. Scholl. Implementing an Object Model on Top of Commercial Database Systems (Extended Abstract). In *Proceedings of the 3rd GI Workshop on Foundation of Database Systems*, Volkse, May 1991.

[Val93] P. Valduriez. Parallel Database Systems: Open Problems and New Issues. *International Journal on Distributed and Parallel Databases*, 1(2):137–166, April 1993.

[Wei91] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[WH93] G. Weikum and C. Hasse. Multi-Level Transaction Management for Complex Objects: Implementation, Performance, Parallelism. *The VLDB Journal*, 2(4), October 1993.

[WS92a] G. Weikum and H.–J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, Inc., 1992.

[WS92b] W.A. Woods and J.G. Schmolze. The KL-ONE family. In *Computer and Mathematics with Applications*, volume 23(2–5), pages 133–177. 1992.