In: Proc. of the Int. Symposium on Databases in Parallel and Distributed System, Austin, 1988, pp. 131-143.

Parallelism in Processing Queries on Complex Objects

T. Härder H. Schöning A. Sikeler

University Kaiserslautern, Department of Computer Science,
P.O. Box 3049 , D-6750 Kaiserslautern, West Germany

Abstract

Complex objects to support non-standard database applications require the use of substantial computing resources because their powerful operations and their related integrity constraints must be performed and maintained in an interactive environment. Since the exploitation of parallelism within such operations seems to be promising, we investigate the principal approaches for processing a query on complex objects (molecules) in parallel. A number of arguments favor methods based on inter-molecule parallelism as against intra-molecule parallelism. Retrieval of molecules may be optimized by multiple storage structures and access paths. Hence, maintenance of such storage redundancy seems to be another good application area to explore the use of parallelism. Deferred update seems to be a bad idea, whereas concurrent update strategies incorporate salient application features. For performance reasons, we have chosen a multiprocessor system sharing an instruction addressable common memory which is used for buffer management, synchronization, and logging/ recovery. Activation of concurrent requests is supported by a nested transaction concept which allows a safe and effective execution control within parallel actions of an operation.

1. Introduction

Non-standard database applications such as 3D-modeling for workpieces or VLSI chip design [1] require adequate modeling facilities for their application objects for various reasons. If the object representation is adjusted to the needs of a particular application area, the intended object handling (e.g. composition or transformation of objects) may be performed in a natural and smooth way, thereby saving a lot of ponderous deviation steps. Data models supporting such applications embody some degree of object orientation (towards the application objects). The notion of complex objects is used to indicate that such objects have an internal structure (structured components) maintained by the database management system (DBMS) and that access is provided to the object as a whole as well as to its components (structural object orientation). To enhance integrity control and semantic expressiveness, more object properties beyond the structural relationships have to be specified and preserved by the data model (behavioural object orientation). Such a rich data model supports appropriate forms of data abstraction and encapsulation (e.g. ADTs) which relieve the application from the burden of maintaining intricate object representations and checking complex integrity constraints.

On the other hand, DBMS requests using such a high-level object-oriented DBMS interface (incorporated by a set of powerful ADTs) imply very long execution path lengths since all aspects of complex object handling have to be performed inside the DBMS. When applying processing concepts known from conventional DBMS, serious performance problems may occur in terms of response time, e.g. in an interactive construction environment. Although an operation is decomposed into a tree of suboperations (Fig. 1b), the classical DBMS processing strategy observes a synchronous activation of each suboperation and its strictly serial execution. Only some conventional systems deviate marginally from this processing strategy by using low-level parallelism for certain house-keeping tasks, e.g. writing modified pages to disk. In general, however, concurrent execution on behalf of a user operation is not achieved [2].

The use of intra-transaction parallelism for higher-level operations was investigated in a number of database machine projects [3]. These approaches focus on the exploitation of parallelism in the framework of the relational model. Complex relational queries are transformed into an operator tree of relational operations in which subtrees are executed concurrently (evaluation of subqueries on different relations) [4]. Other approaches utilize special storage allocation schemes by distributing relations across multiple disks. Parallelism is achieved by evaluating the same subquery on the various partitions of a relation [5, 6].

We are going to investigate possible strategies to exploit parallelism when processing complex objects. In order to be specific, we have to identify our concepts and solutions in the framework of a particular data model and a system design facilitating the use of parallelism. Therefore, we refer to the molecule-atom data model (MAD model [7]) which is implemented by an NDBS kernel system called PRIMA [8]. We use the term NDBS to describe a database management system tailored to the support of non-standard applications.

For this purpose, we introduce the essential concepts of an NDBS architecture and a model of NDBS operations. We focus on the principal ways to process a query on complex objects in parallel. Furthermore, we consider the use of parallelism when redundant storage structures kept for performance reasons have to be maintained. In order to achieve a safe and effective execution control for parallel actions, we tailor the concept of nested transactions to our distributed processing strategies in a server complex carrying the PRIMA code. Finally, we conclude with a summary of our design proposals.

2. A Model of NDBS Operations

In order to describe our concepts of supporting parallelism in the framework of NDBS processing, we introduce a multi-layered

architecture with well-defined interfaces. A major design objective was the provision of modular system layers which directly facilitate the decomposition of operations and, as a consequence, foster concurrent processing. On the other hand, our architecture as shown in Fig. 1a reflects a high degree of data independence maintained by explicit interfaces among layers.

The overall architecture consists of a so-called NDBS kernel and a number of different application layers, which map particular applications to the data model interface of the kernel. Our kernel which is considered application-independent is divided into three layers:

- The storage system provides a powerful interface between main memory and disk. It maintains a database buffer and enables access to sets of pages organized in segments [8].
- The access system manages storage structures for basic objects called atoms and their related access paths. For performance reasons, multiple access paths and redundant storage structures may be defined for atoms.
- The data system dynamically builds the objects available at the data model interface. In our case, the kernel interface is characterized by the MAD model and its language called MQL (molecule query language) [7]. Hence, the data system performs composition and decomposition of complex (structured) objects called molecules.

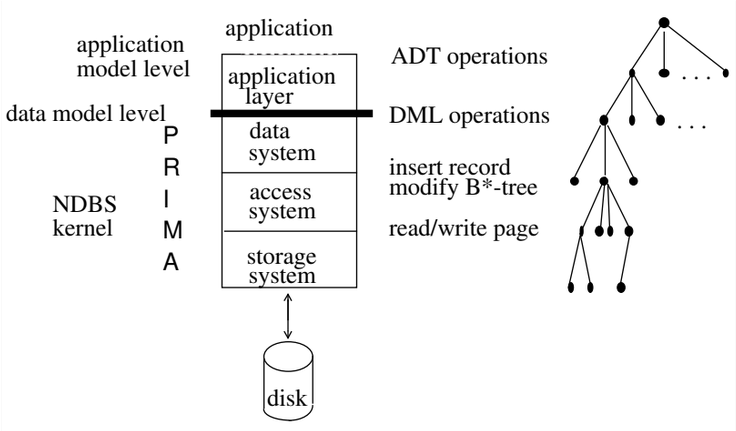


Figure 1: Model of operation

The application layer uses the complex objects and tailors them to (even more complex) objects according to the application model of a given application. This mapping is specific for each application area (e.g. 3D-CAD, VLSI design, geographic information management). Hence, for each application area a different application layer exists which provides a tailored interface (e.g. in form of a set of ADT operations) for the corresponding application.

The NDBS architecture as exhibited in Fig. 1a lends itself to a workstation-server environment in a smooth and natural way. The application and the corresponding application layer are dedicated to a workstation, whereas the NDBS kernel is assigned either to a single server processor or to a server complex consisting of multiple processors. This architectural subdivision is strongly facilitated by the properties of the MAD model: Sets of molecules which consist of sets of heterogeneous atoms may be specified as processing units.

Thus far, we have sketched the static mapping of NDBS objects and operations organized in hierarchical layers. The dynamic processing of user operations may be explained by a tree of suboperations as illustrated in Fig. 1b. This operation tree reflects the hierarchical decomposition of a user operation according to our architectural model. The operations call up services from the underlying layer which are decomposed, in turn, into more elementary operations: “Each call to a subroutine is an example of a primitive at one level of implementation invoking a set of primitives at a lower level of control [9]”.

In conventional DBMS, all operations in such an operation tree are called synchronously and are executed serially (left-most depth-first traversal). Given an appropriate run-time environment, operations at a certain level may be called in parallel, i.e. the corresponding subtrees as shown in Fig. 1b are executed (traversed) concurrently. In principle, such a decomposition and parallel execution is conceivable at every level of operation. However, certain prerequisites such as sufficient operation granules are necessary for successful application of concurrency in processing a user operation. In this paper, we want to focus on concepts to exploit parallelism within the NDBS kernel, that is, how concurrent and asynchronous actions should be organized in the server complex carrying the NDBS kernel code.

2.1 The Data System Interface

In order to describe the concepts for achieving parallelism in sufficient detail, we have to refine our view of the kernel architecture and the interfaces involved. It is obvious that the data model plays the major role and determines many essential factors which enable reasonable parallelism: sufficiently large data granules, set orientation of request, dynamic construction of objects (result sets), flexible selection of processing sequences, etc.

In our system, the data model interface is embodied by the MAD model and its language MQL which is similar to the well-known SQL language. Here, we cannot introduce this model with all its complex details, but only illustrate the most important concepts necessary for our discussion. In the MAD model, atoms are used as a kind of basic element (or building block) in order to represent entities of the real world. In a similar way to tuples in the relational model, they consist of an arbitrary number of attributes. The attributes' data types can, however, be chosen from a richer selection than in the relational model, i.e. apart from the conventional types the type concept includes

- the structured types RECORD and ARRAY,
- the repeating group types SET and LIST, both yielding a powerful structuring capability at the attribute level as well as
- the special types IDENTIFIER (serving as surrogates) for identification purposes and REF_TO for the connection of atoms.

Atoms are grouped to atom types. Relationships between atoms are expressed by so-called links and are defined as link types between atom types. Link types are treated in a symmetric way, i.e. links may be used in either direction in the same manner. Such link types directly map all types of relationships (1:1, 1:n, n:m). The flexibility of the data model is greatly increased by this direct and symmetric mapping. Link types are represented by a pair of REF_TO attributes (reference and “back-reference”) one in either involved atom type. For example, a link type may be specified as follows:

- FIDs: SET_OF (REF_TO(Face.EIDs)) in an atom type *Edge*
- EIDs: SET_OF (REF_TO(Edge.FIDs)) in an atom type *Face*.

In the database, all atoms connected by links form meshed structures (atom networks) as illustrated in Fig. 2.:

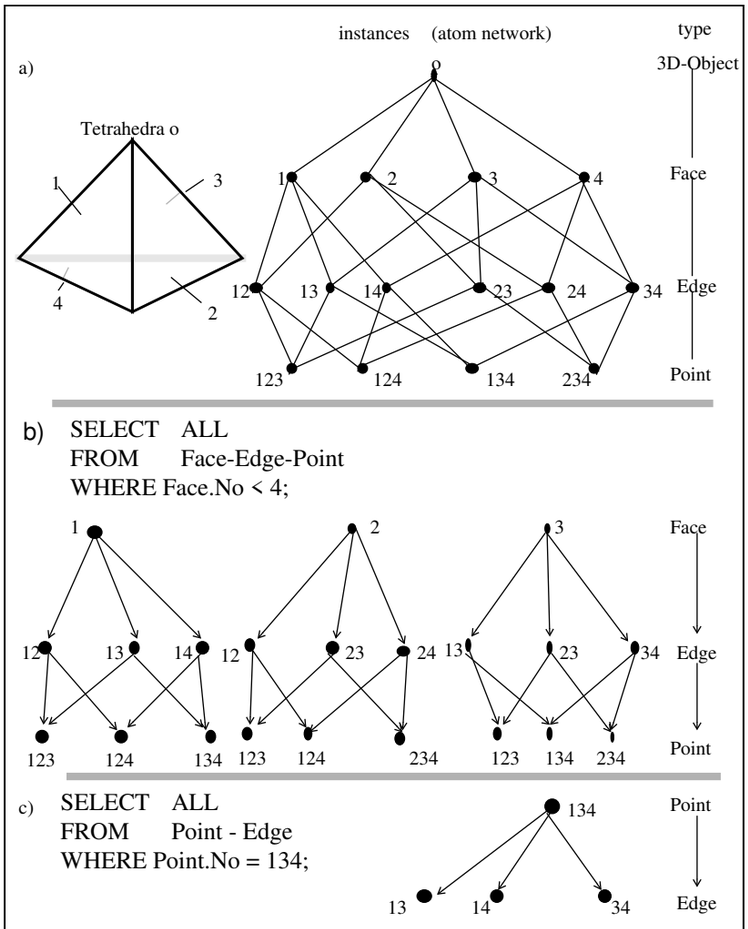


Figure 2: Dynamic construction of molecules from a sample geometric object

Molecules are defined dynamically by using MQL statements and have to be derived at run-time. Each molecule belongs to a molecule type (which is specified in the FROM clause). The type description establishes a connected, directed and acyclic type graph (cycles occur when recursive types are specified), in that a starting point (i.e. root atom type) and all participating atom and link types (for short "-") are specified. A particular example of a molecule type is Face-Edge-Point. Such a molecule type determines both the molecule structure as well as the molecule set which groups all molecules with the same structure. At the conceptual level, the dynamic construction of molecules proceeds in a straight-forward manner using the molecule type description as a template: For each atom belonging to the root atom type all children, grandchildren and so on are connected to the molecule structure, terminating after all leaves of the molecule structure are reached. Connecting children to the molecule structure means performing the hierarchical join which is supported by the link concept. Hence, for each root atom a single molecule is constructed. Fig. 2b shows the result of a molecule construction for Face-Edge-Point molecules, where the set of molecules was restricted. Furthermore, it illustrates the most important properties of the MAD interface

- An MQL request handles a set of molecules.
- The molecules as complex objects consist of sets of atoms of different type, i.e., they are embodied by sets of interrelated heterogeneous record structures.
- Molecule construction is dynamic and allows symmetric use of the atom networks (e.g. *Point -Edge* (Fig. 2c)).

2.2 The Access System Interface

The access system provides an atom-oriented interface which allows for direct and navigational retrieval as well as for the manipulation of atoms. To satisfy the retrieval requirements of the data system, it supports direct access to a single atom as well as atom-by-atom access to either homogeneous or heterogeneous atom sets.

Manipulation operations (insert, modify, and delete) and direct access operate on single atoms identified by their logical address (or surrogate) which is used to implement the IDENTIFIER attribute as well as the REF_TO attributes. Performing manipulation operations, the access system is responsible for the automatic maintenance of the referential integrity defined by the REF_TO attributes. Thus, a manipulation operation on such an attribute requires implicit manipulation operations on other atoms in order to adjust the appropriate back references. These operations however, are triggered by a special consistency manager (section 3.3).

Different kinds of scan operations are introduced as a concept to manage a dynamically defined set of atoms, to hold a current position in such a set, and to successively deliver single atoms. Some scan operations, however, are added in order to optimize retrieval access. Therefore, they may depend on the existence of a certain storage structure (defined by the database administrator):

- The atom-type scan delivers all atoms in a system-defined order utilizing the basic storage structure which exists for each atom type.
- The access-path scan provides appropriate means for fast value-dependent access based on different access path structures such as B-trees, grid files, and R-trees.
- The sort scan processes all atoms following a specified sort criterion also utilizing the basic storage structure of an atom type. However, sorting an entire atom type is expensive and time consuming. Therefore, a sort scan may be supported by an additional storage structure, namely the sort order.
- The atom-cluster scan speeds up the construction of frequently used molecules by allocating all atoms of a corresponding molecule in physical contiguity using a tailored storage structure as a so-called atom cluster. For example, in Fig. 2 each *Face* atom and all its associated *Edge* and *Point* atoms may be organized to form an atom cluster (Fig. 3). On a logical level, an atom cluster corresponds to a molecule. It is described by a special so-called characteristic atom which consists of references to all atoms belonging to the molecule. This characteristic atom together with all the referenced atoms is mapped onto a single physical record which in turn is stored in a set of pages.

The underlying concept is to make storage redundancy available outside the access system by offering appropriate retrieval operations (i.e. the choice of several different scans for a particular access decision by the optimizer of the data system), whereas in the case of update operations storage redundancy has to be hidden by the access system. As a consequence, maintaining storage redundancy in an efficient way is a major task of the access system. However, sequential update of all storage structures existing for a corresponding atom results in a lack of efficiency which is not acceptable. Therefore, exploiting parallelism seems to be a natural way to speed up a single manipulation operation.

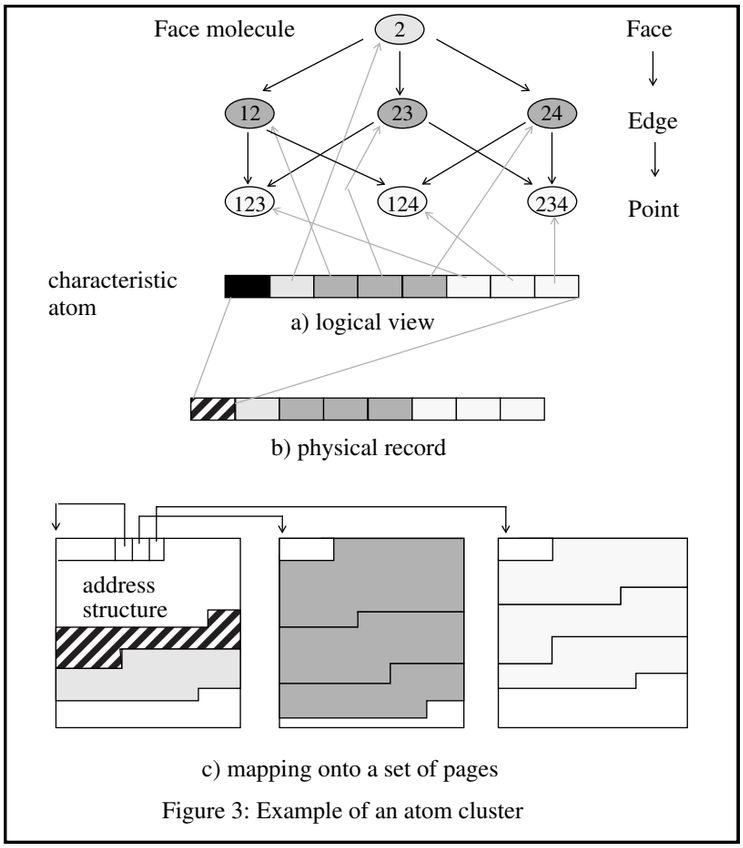


Figure 3: Example of an atom cluster

2.3 Activation of Concurrent Requests

Before we start to evaluate our concepts for achieving parallelism to perform data system and access system functions, we briefly sketch the process (run-time) environment of our DBMS kernel PRIMA. In order to provide suitable computing resources, PRIMA is mapped to a multi-processor system, i.e., the kernel code is allocated to each processor of our server complex (multiple DBMSs). The DB operations to be considered are typically executed on shared (or overlapping) data which requires synchronization of concurrent accesses. Due to the frequency of references (issued from concurrent tasks) accessibility of data and synchronization of access must be solved efficiently.

In a so-called DB-distribution architecture [10] every (loosely coupled) processor handles a partition of the database; such a static data subdivision and allocation is obviously in conflict with our processing requirements. Therefore, DB-sharing architectures [11] where multiple DBMS share the database at the disk level seem to be much more flexible and appropriate for our purpose. In a loosely coupled system, however, each DBMS has its own system buffer creating the need to cope with fully replicated data; e.g. modification of a page in a buffer makes all copies of this page in other buffers obsolete (buffer invalidation problem). Furthermore, the lack of common memory enforces message-based communication for inter-process cooperation (e.g. concurrency control) which seems to be by far too slow for our type of application. As a consequence, use of shared memory for critical functions is mandatory for performance reasons. Hence, we have designed our system to run on a server complex where instruction-addressable common memory [12] is available for buffer management, synchronization, and logging/recovery. Such an architecture (sometimes called closely coupled) avoids buffer invalidation (only one buffer) and provides memory-based message exchange as well as instruction-based synchronization primitives for shared data accesses (e.g. a “compare and swap” instruction). Fig. 4 gives a short illustration of the major architectural issues. Our experimental system for the server complex consists of up to five processors with sufficient private memory, the common memory, an attachment to a file processor, and a high-bandwidth communication system for coupling the server processors to workstations carrying the application layers and the corresponding applications..

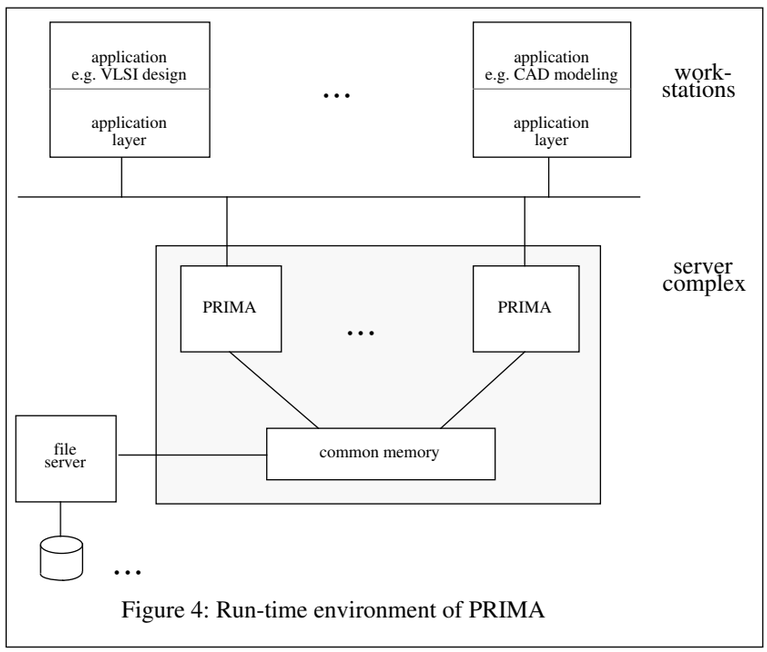


Figure 4: Run-time environment of PRIMA

Each instance of PRIMA (running on a particular processor) is subdivided into a number of processes. Each process may initiate an arbitrary number of tasks which serve as run-units for the execution of single requests. Cooperation among processes is performed by establishing some kind of client-server relationship; the calling task in the client process issues a request to the server process where a task acts upon the request and returns an answer to the caller. In our model, a client invokes a server asynchronously, i.e. it can proceed after the invocation, and hence,

can run concurrently with this server. A typical sequence of concurrent request activations of a task T1 is

```
CALL   service   P2
       . . .
CALL   service   P3
       . . .
CALL   service   P4
       . . .
SUSPEND task T1 (all requests)
```

In this example, task T1 has asynchronously called three processes with multi-tasking capability. With the suspend command, it can deactivate itself to wait for the availability of all requested answers (Selective suspend for any answer is also an option). This short remark concerning our run-time environment and our concurrent request activation is sufficient to follow the discussion in the subsequent chapters. Due to space limitations, we cannot refine our arguments on these system issues.

3. Using Parallelism in Query Processing

Query Processing operates on sets of molecules which are either extracted from the database (retrieval) or updated, inserted, or deleted (manipulation). In either case, the complex operation has to be cut down to one-atom-at-a-time operations [2]. In this chapter we discuss some techniques to exploit parallelism in executing these atom-oriented operations as well as in higher-order operations on intermediate results.

Three phases of query processing can be isolated [13]. The compilation phase checks for the syntactic and semantic correctness of a query and performs some obviously simplifying query transformations. Finally, it derives an operator tree consisting of several operator types, most of which accept and produce sets of molecules. The leaves of this operator tree transform heterogeneous atom sets to molecules (construction of simple molecules for retrieval) and vice versa (molecule modification by atom manipulation). This operator tree is the input of the optimization phase which is expected to reorganize it into a somehow "optimal" form. This includes reordering, combination, and splitting of operator tree nodes. Furthermore, methods have to be chosen for each operator, i.e. evaluation strategies (e.g. nested-loop join), storage structure usage (e.g. B-tree), and amount of parallelism (as described below). Although there is much to be said about these two phases, we concentrate on the discussion of the third phase, i.e. query evaluation. The input of this phase is the operator tree introduced above. For each node, we need an active unit to compute its result. Since there are several operator types, we assume a process for each of them. As a consequence, more than one node of an operator tree may be assigned to the same process, which is supported by the concepts presented in chapter 2.3.

We discuss the operator tree interpretation separately for retrieval and manipulation. In either case, we have to handle a set of molecules, each of which consists of a set of atoms. Thus, we investigate parallel handling of molecules (inter-molecule parallelism) as well as parallel handling of a molecule's components (intra-molecule parallelism).

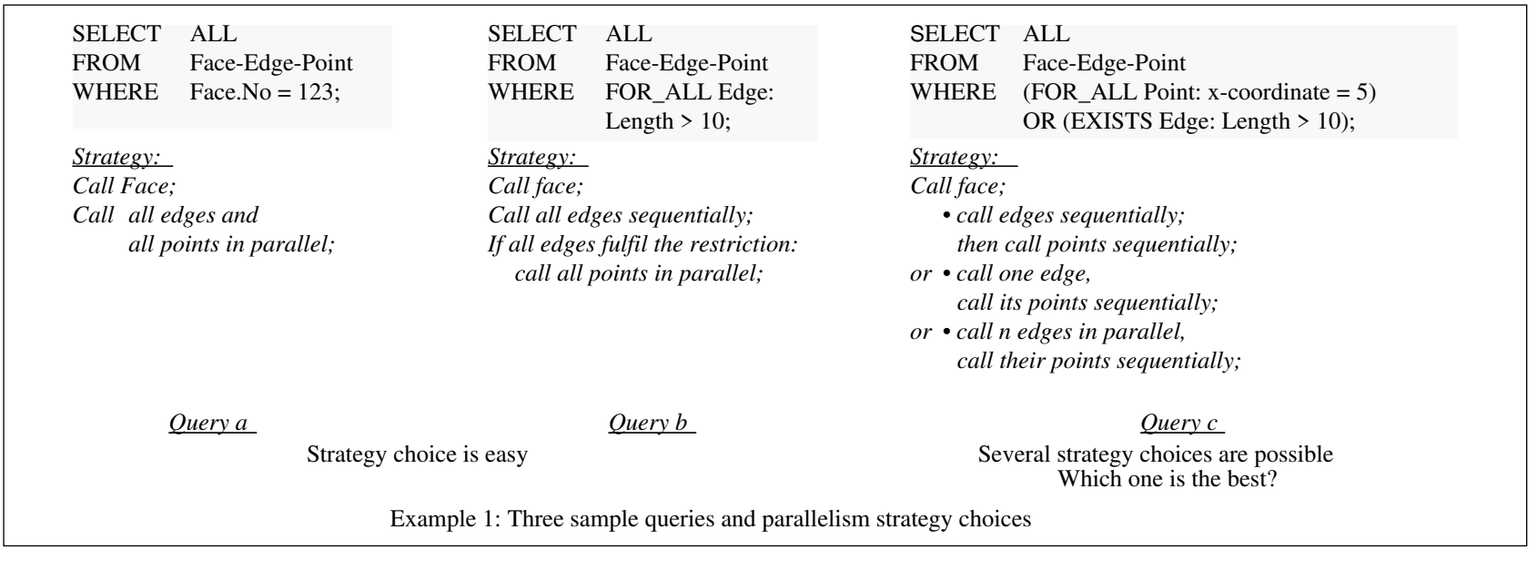
3.1 Parallelism in Retrieval Evaluation

Evaluation starts at the root operator, which needs results from all of its children in the operator tree to compute its own result. Depending on the operator type and the method chosen, children can be evaluated in parallel, e.g. the children of a binary merge-join operator can be accessed concurrently, while those of a nested-loop join cannot. The evaluation of leaf operators requires a lot of access system calls to build up simple molecules (hierarchical, non-recursive molecules). Of course, we assume that the access system is able to handle an arbitrary number of asynchronous calls in parallel. Therefore, construction of simple molecules seems to be a good candidate for using parallelism within the handling of each molecule.

Parallelism Within Construction of Simple Molecules

An obvious strategy is to call the root atom of each molecule via access system scan. Following this all child atoms of the root atom which are identifiable by reference attributes are called, then their children, and so on. These accesses to the children of an atom may be done in parallel (example 1a).

However, in many cases the user is not interested in all molecules of a certain type, but strongly restricts the molecules he wants to see. In this situation, it would be inefficient to search all atoms of all molecules and then throw away most of them by a separated restriction operator. Instead, we want to integrate the restriction facility into the operator "construction of simple molecules" leading to a more efficient evaluation strategy. Restrictions on the root atom are passed on to the access system which allows scan restrictions. All other restrictions on dependent atoms are evaluated as early as possible. As soon as it becomes evident



Example 1: Three sample queries and parallelism strategy choices

during molecule construction that a molecule will be disqualified, none of its atoms has to be fetched any more, thus saving many access system calls (example 1b).

Of course, this approach is contradictory to the parallel molecule construction proposed above, because we want to fetch as few atoms as possible, if a molecule is disqualified. Therefore, we combine both techniques: Atom types that do not contribute to molecule qualification should be treated last. Their atoms can be called in parallel. Atom types restricted by an ALL-quantor should be called sequentially, since each atom of this type can indicate molecule disqualification. While good strategies for these extreme cases are easy to find, much more complicated situations can be thought of (example 1c). They raise the question, if in some situation a compromise on the amount of parallelism and unnecessary atom accesses should be made, e.g. limitation of parallel atom calls to a constant n, thereby limiting unnecessary atom calls to n-1 (third choice in example 1c). We are still investigating this case for generally applicable rules to decide the optimal amount of parallelism for each atom type as well as the best sequence of atom accesses.

The top-down approaches suggested above are sometimes not the most efficient strategies. When highly selective restrictions are defined on some child types, a bottom-up approach may be more promising. In this case, the first step evaluates the qualifying child atoms. Since some of these atoms may be orphans, it is necessary to explicitly check the existence of a related root atom. Finally, the whole molecule is constructed for each of the identified root atoms following the same guidelines as sketched above (example 2b).

So far, we have discussed parallelism within the construction of one molecule. Since queries deal with sets of molecules, we should consider inter-molecule parallelism, too.

Inter-Molecule Parallelism

The most simple model for the computation of a set of molecules is to build up the first molecule completely, then the second and so on, thereby preserving the order of molecules induced by construction of simple molecules. There cannot be any parallelism among an process and its descendants or ancestors. To enable this kind of parallelism, we assume a pipeline mechanism. In other terms, each molecule is handled by an own task, thereby allowing concurrent processing.

In particular, whenever the process for construction of simple molecules finds a root atom for a molecule, it builds up this molecule in a separate task, while another concurrent task calls the next root atom. Example 3 shows a sample evaluation sequence for a simple retrieval based on Fig. 2. The *projection* operator calls the restriction operator, which uses *construction of simple molecules* to build up a molecule of type *Edge-Point*. For this purpose, *construction of simple molecules* initiates an atom-type scan on *Edge* (which gets the scan id 1). In example 3, the first *Edge* atom delivered by this scan is 12. In the subsequent step, the children of 12 (i.e. 123, 124) are requested. Simultaneously, a new task is created to build up the next molecule via the open atom-type scan (i.e. the molecule belonging to edge 13). As soon as 123 and 124 are handed to the first task, the molecule under construction is delivered to *restriction*, which then evaluates the inner query using *aggregation* and *construction of simple*

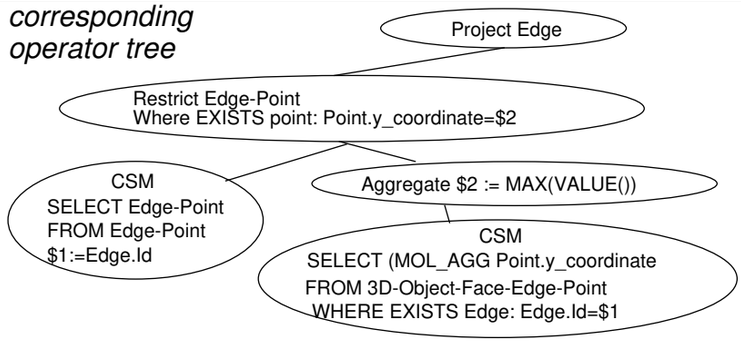
Query:

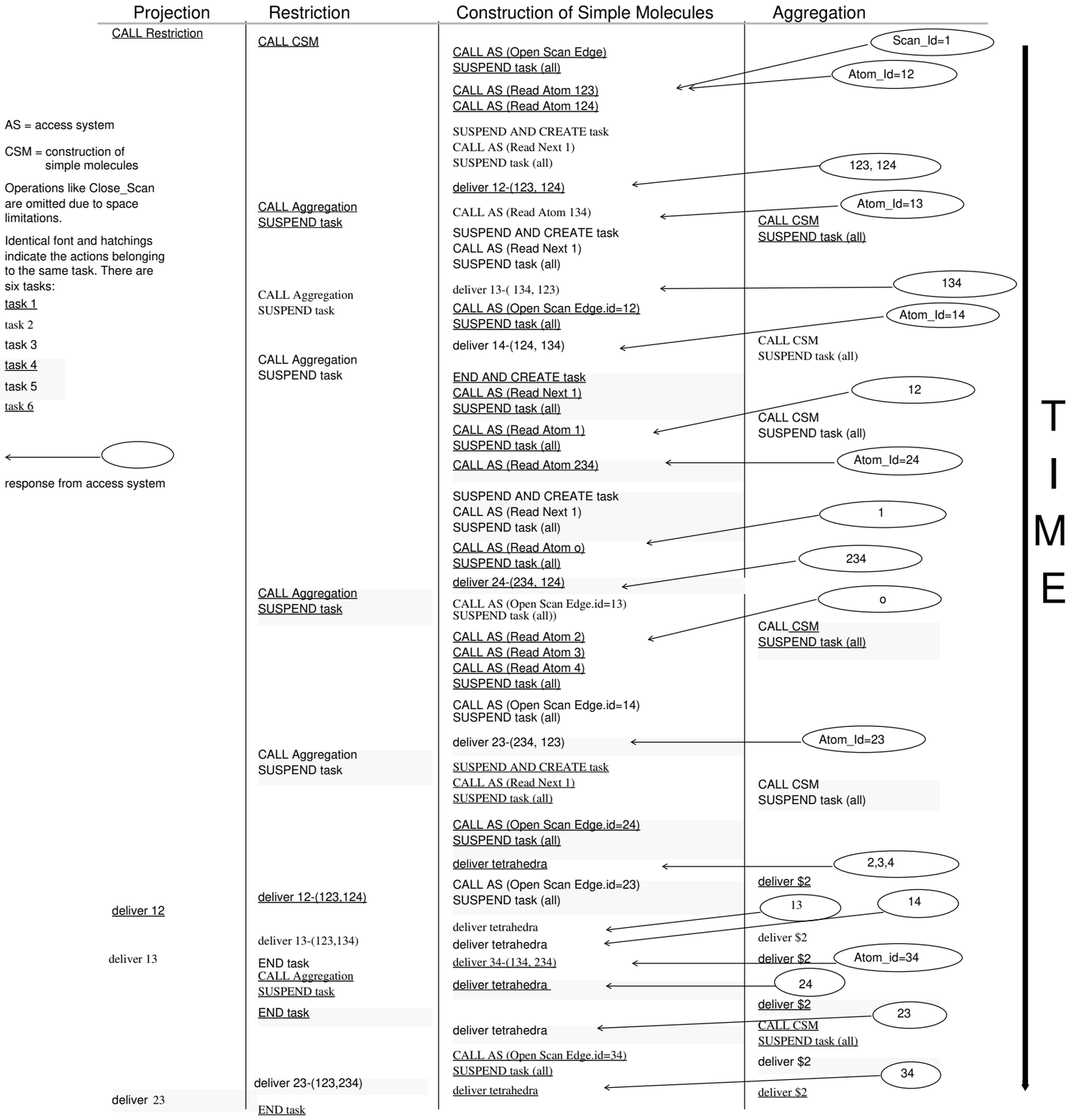
Find all Edges that are top-most in all objects they belong to.

```
SELECT   E
FROM     E(Edge)-Point
WHERE    EXISTS Point: Point.y_coordinate =
         MAX ( VALUE (   SELECT   (MOL_AGG (Point.y_coordinate))
                         FROM     3D-Object-Face-Edge-Point
                         WHERE    EXISTS Edge: Edge.id=E.id))
```

E is a tuple variable for Edge, MOL_AGG combines all values of a certain attribute type within one molecule into a list, VALUE converts a set of molecules into a list (i.e. VALUE is an aggregation function).

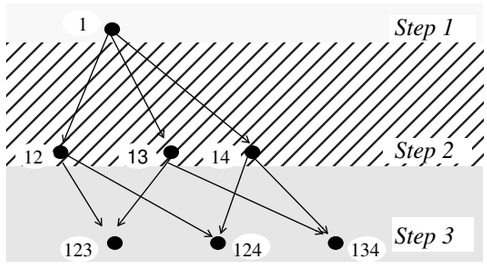
corresponding operator tree





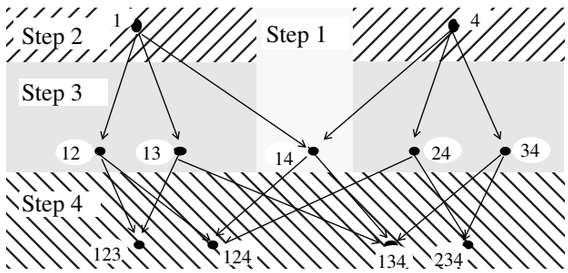
Example 3: Execution sequence of a sample query based on figure 2

```
SELECT   ALL
FROM     Face-Edge-Point
WHERE    Face.No=1;
```



a) Top-down molecule construction

```
SELECT   ALL
FROM     Face-Edge-Point
WHERE    EXISTS Edge: Edge.No=14;
```



b) Bottom-up molecule construction

Example 2: Two ways to construct molecules

molecule s. All activities which are on the same line in example 3 can be performed in parallel, as the vertical time axis indicates.

The pipeline structure defined this way (which at this point of the discussion is introduced as a model of computation and not as schedule for hardware-assignment), is very dynamic and complex, since the number of pipeline stages to run through is data dependent for many operator types and may vary for each molecule. Since this results in varying construction times, order-preservation cannot be guaranteed. As a consequence, there must not be any operator with a varying number of pipeline stages in the operator tree between a sort operator and the corresponding operator that relies on the sort order.

3.2 Parallelism in Manipulation Evaluation

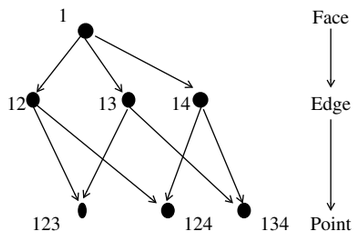
As for retrieval evaluation, we discuss intra- and inter-molecule parallelism. Parallelism among several molecules by creation of a separate task for each of them is possible for manipulation, too. When existing molecules are to be manipulated, tasks emerging from retrieving them can be continued for update. Within one molecule, either a top-down or a bottom-up strategy can be applied, both of them allowing parallelism among most of the atoms of a molecule (example 4).

sample manipulation statement

```
DELETE   ALL
FROM     Face-Edge-Point
WHERE    Face.No=1;
```

This is just a query to show evaluation strategies; it is semantically wrong with respect to figure 2

affected molecule



top-down deletion	*bottom_up deletion*
delete (1)	delete (123), delete (124), delete (134)
delete (13), delete (12), delete (14)	delete (13), delete (12), delete (14)
delete (123), delete (124), delete (134)	delete (1)

all access system calls in the same line can be done in parallel

Example 4: Manipulation of a molecule with top-down and bottom-up strategy

3.3 Consistency Management

Deleting a molecule leads to another problem: Since referential integrity shall be guaranteed by query evaluation, references pointing to deleted atoms have to be removed. Thus, it is quite an evident idea to guarantee referential integrity within each atom operation. Whenever an atom is deleted, all references pointing to it from other atoms are removed immediately by the access system. Thus, if a molecule is deleted atom-by-atom, an atom is updated several times in order to remove invalidated references to atoms of

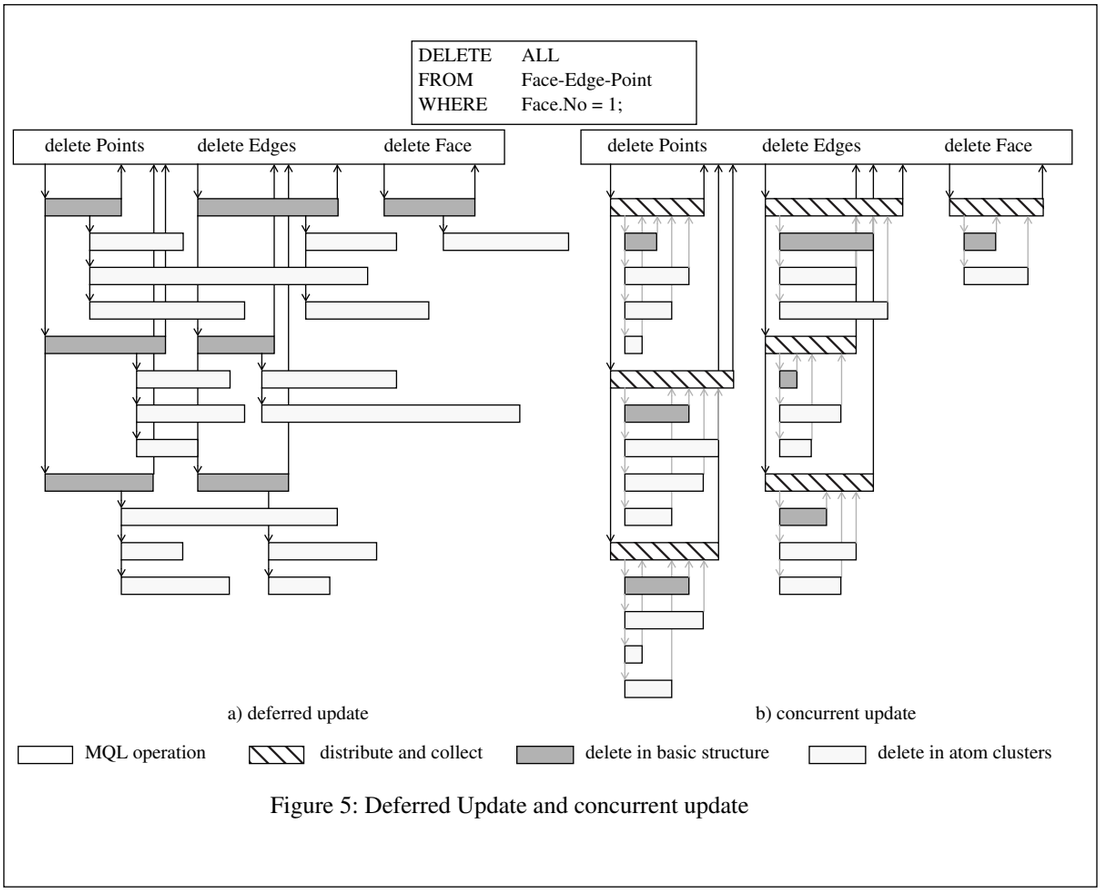


Figure 5: Deferred Update and concurrent update

the same molecule, before it is deleted itself. Obviously, the updates are needless effort, because the unit of consistency was chosen too small.

If the consistency unit is identified with the query evaluation on a molecule, the problem remains for atoms shared among several molecules. Thus we identify the evaluation on the whole set of molecules as unit of consistency enforcement. Since only the access system is aware of all references pointing to an atom, it has to assure appropriate bookkeeping of all consistency violations. This bookkeeping, however, cannot be done within the access system, since it does not see the dependencies of several atom operations. The data system, however, is responsible for molecule processing and therefore should not be burdened with this task.

For these reasons, we delegate this bookkeeping to a separate component called consistency manager. This component is responsible for maintaining all kinds of consistency constraints inherent in the data model, that is, referential integrity, cardinality restrictions, etc. defined in the MAD schema as well as similar constraints in the corresponding meta data. In our implementation, the access system passes consistency information directly to the consistency manager, and units of consistency can be defined in a more flexible manner [14].

Intelligent bookkeeping can identify mutual annulling consistency violations. In particular, the component may work in parallel with all other components. It is left to its decision, when and how consistency is to be enforced, unless consistency is explicitly required (example 5).

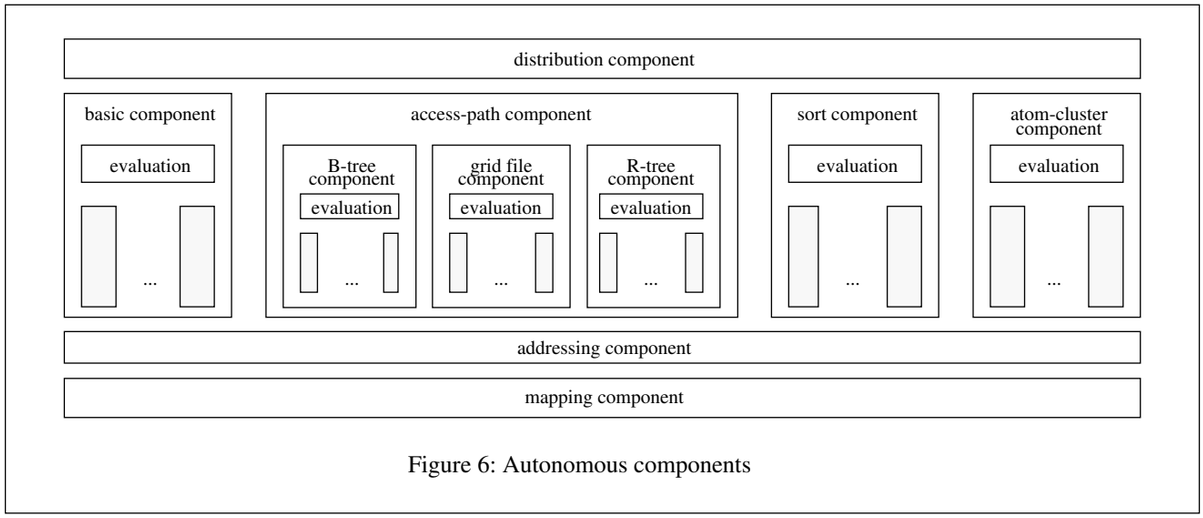


Figure 6: Autonomous components

Operation	reference to atoms	consistency violations
delete (123)	12, 13, 23	12⟶123, 13⟶123, 23⟶123
delete (124)	12, 14, 24	12⟶123, 13⟶123, 23⟶123, 12⟶124, 14⟶124, 24⟶124
delete (134)	13, 14, 34	12⟶123, 13⟶123, 23⟶123, 12⟶124, 14⟶124, 24⟶124, 13⟶134, 14⟶134, 34⟶134
delete (13)	123, 134, 1, 3	12⟶123, 23⟶123, 12⟶124, 14⟶124, 24⟶124, 14⟶134, 34⟶134, 1⟶13, 3⟶13
delete (12)	123, 124, 1, 2	23⟶123, 14⟶124, 24⟶124, 14⟶134, 34⟶134, 1⟶13, 3⟶13, 1⟶12, 2⟶12
delete (14)	124, 134, 1, 4	23⟶123, 24⟶124, 34⟶134, 1⟶13, 3⟶13, 1⟶12, 2⟶12, 1⟶14, 4⟶14
delete (1)	13, 14, 12, o	23⟶123, 24⟶124, 34⟶134, 3⟶13, 2⟶12, 4⟶14, o⟶1
		These references violate the consistency and therefore have to be removed

Example 5: Consistency violation bookkeeping for the operation of example 4 (bottom-up)

Query evaluation offers many opportunities for parallel execution. Intra-molecule parallelism imposes some optimization problems for retrieval, whereas update may be done without further problems. Inter-molecule parallelism can be applied to both update and retrieval, although it requires specific operator tree properties for retrieval. The amount of parallelism achieved by this method directly depends on the cardinality of the molecule set to be operated on. As a final point even consistency enforcement can be done in parallel.

4. Maintaining Redundancy by Parallel Oper-

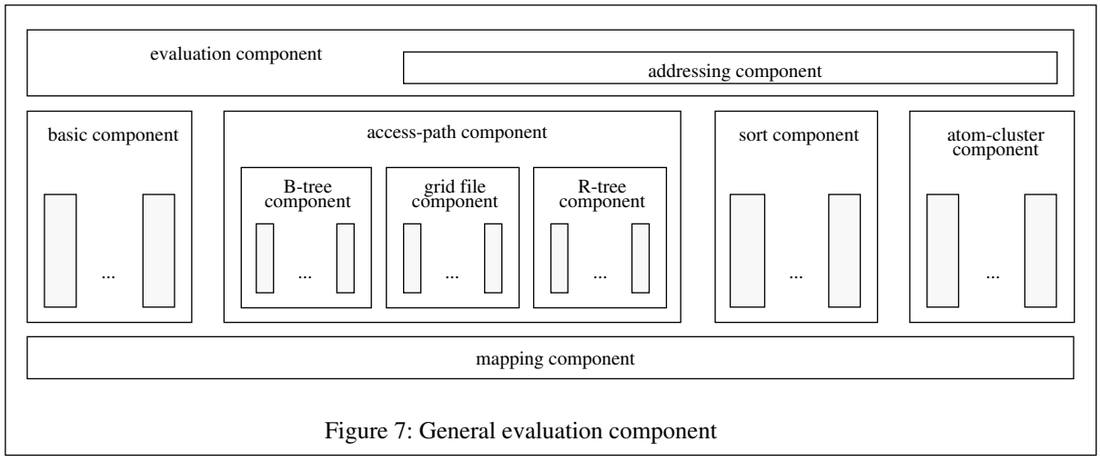


Figure 7: General evaluation component

ations

To speed up data system operations we have introduced some algorithms for the parallel construction/maintenance of complex objects represented as sets of heterogeneous atoms. In the following, we discuss the implementation of concurrent maintenance operations on redundant storage structures used for such atoms. As in the data system, two kinds of parallelism may be distinguished within the access system.

- The inter-operation parallelism allows for the parallel execution of an arbitrary number of independent access system calls. This kind of parallelism is a prerequisite for the parallelism introduced in the data system.
- The intra-operation parallelism however, exploits parallelism in executing a single access system call.

In this chapter, we will concentrate on intra-operation parallelism, since inter-operation parallelism is easily achieved by the underlying processing and transaction concept (section 2.3 and chapter 5). For this purpose, however, the mapping process performed by the access system has to be outlined in some more detail in order to reveal purposeful suboperations to be executed in parallel.

In order to conceal the storage redundancy resulting from the different storage structures we have introduced the concept of a logical record (i.e. atom) made available at the access system interface and physical records stored in the "containers" offered by the storage system, i.e. each physical record represents an atom in either storage structure. As a consequence, an arbitrary number of physical records may be associated with each atom. For example, the creation of an atom cluster for each Face-Edge-Point molecule in Fig. 2 would imply that all Edge atoms belong to two atom clusters and all Point atoms to three (due to the properties of a tetrahedra).

The relationship between a single atom and all its associated physical record is maintained by a sophisticated address structure related to each atom type. This address structure maps the logical address identifying an atom onto a list of physical addresses each indicating the location of a corresponding physical record within the "containers" (page address).

In contrast to the data system, however, the exploitation of parallelism within the access system is limited to the manipulation operations. Although most of the retrieval operations are also decomposed into further suboperations (e.g. in the case of an access-path scan on a tree structure: read the next entry in order to obtain the next logical address, access the address structure for the associated physical addresses, access either physical record), these suboperations cannot be executed in parallel due to the underlying precedence structure. Furthermore, each retrieval operation is tailored to a certain storage structure, thus operating not only on a single atom, but also on a single physical record.

On the other hand, each manipulation operation on an atom may be decomposed in quite a natural way into corresponding manipulation operations on the associated physical records. These lower-level manipulation operations, however, should be executed in parallel due to performance reasons. There exist (at least) two alternatives to perform such a parallel update:

Deferred Update

Deferred update (Fig. 5a) means that during a manipulation operation on an atom initially only one of the associated physical records is altered (e.g. the physical record in the basic storage structure of the atom type). All other physical records as well as the access paths are marked as invalid. Finally, a number of "processes" is initialized which alter the invalid structures in a deferred manner, whereas the update operation itself is finished. Thus, low-level manipulation operations on additional storage structures may still run, although the manipulation operation on the corresponding atom or even each higher-level operation initializing the modification is already finished. This, however, strongly depends on the embedding of deferred update into the underlying transaction concept.

In order to mark a physical record as invalid the address structure may be used to indicate whether or not the corresponding physical record is valid. Therefore, all operations which utilize the address structure in order to locate a physical record may determine the valid records, whereas all operations which do not utilize the address structure will access invalid records with the exception of the appropriate storage structure was already altered by the corresponding "process". Hence, the corresponding storage structures themselves (access paths, sort orders, and atom clusters) have to be marked as invalid and when performing a scan operation on such an invalid structure each physical record has to be checked as to whether or not it is valid. This, however, requires an additional access to the address structure in order to locate a valid record. Consequently, the speed of a scan operation degrades, since each access to the address structure may result in an external disk access. In order to avoid this undesired behaviour, all invalid atoms (or their logical addresses) may be collected in a number of special pages assigned to each storage structure. These pages may be kept in the database buffer throughout the whole scan operation thus avoiding extra disk accesses. Nevertheless, each physical record has to be compared with the atoms collected in these pages. However, this is not sufficient, since each manipulation operation may require a modification of the whole storage structure, e.g. modifying an attribute which establishes a sort criterion requires the rearrangement of the corresponding physical record within the sort order. This fact also has to be considered during a scan operation. As a consequence, some of the scan operations may become rather complex and thus inefficient. For all these reasons, deferred update seems to be a bad idea.

Concurrent Update

The problem of maintaining invalid storage structures, however, is avoided by concurrent update. Concurrent update (Fig. 5b) means that each manipulation operation on an atom invokes a number of "processes" which alter the associated physical records and access paths in parallel. The update operation is finished when all "processes" are completed. When sufficient computing resources are available, concurrent update may not be more expensive, in terms of response time, than (sequential) update of a single physical record if we neglect the cost of organizing parallelism.

Depending on the software structure of the access system, there are different ways to perform a concurrent update:

Autonomous Components

Each manipulation operation on an atom is directly passed to all components maintaining only a single storage structure type (Fig. 6). Each component checks which storage structures of the appropriate type are affected by the manipulation operation. The corresponding storage structures are then modified either sequentially or again in parallel.

As a consequence, all components have to provide a uniform interface including all manipulation operations offered by the access system (i.e. insert, modify, and delete of a single atom identified by its logical address) as well as all retrieval operations. A quite simple distribution component directs each request to all components maintaining a storage structure type and collects their responses. This means, each component initially performs an evaluation phase during which it checks

- whether or not it has to perform the desired operation and if so,
- which storage structures of the appropriate type are really affected.

For this purpose, the addressing component (maintaining the common address structure) and the meta-data system (maintaining all required description information) are utilized. After the evaluation phase the proper operation is performed on each affected storage structure either sequentially or in parallel, thereby again utilizing two common components: the addressing component in order to notify the modification of a physical address and the mapping component in order to transform a logical record (i.e. atom) into a physical record and vice versa (thus achieving a uniform representation of physical records which is mandatory for some retrieval operations which use one of the physical records when accessing an atom (e.g. direct access)).

Thus, it is quite easy to add a new storage structure type (e.g. a dynamic hash structure as an additional access path structure) by simply integrating a corresponding component into the overall access system. However, there may be some drawbacks regarding performance aspects. During each operation, all components have to perform the evaluation phase although in many cases only a few or even only one component are affected. Moreover, the addressing component may become a bottleneck, since access to the address structure has to be synchronized in order to keep it consistent.

General Evaluation Component

These problems, however, may be avoided by a general evaluation component (Fig. 7) which replaces the simple distribution component as well as the evaluation phases in each of the components maintaining a storage structure type. Additionally, it solely maintains the address structure. As a consequence, this general evaluation component becomes much more complex. It requires dedicated information about each component in order to decide whether or not a component is affected by an operation. This means, for example, for the update operations:

- The basic component is always affected.
- The other components are affected as far as there exists a corresponding storage structure:
  - The sort component is particularly affected when an attribute establishing a sort criterion is modified.
  - The access-path component particularly has to be invoked when a key attribute is modified.
  - The atom-cluster component is particularly affected when reference attributes constituting an atom cluster are modified.

Additionally, the general evaluation component has to know the operations offered by either component in order to invoke the corresponding component in the right way. Although these operations may be tailored to the corresponding storage structure type (e.g. insert (key, logical address) in the case of an access path structure), it seems to be useful again to provide a uniform interface to all components in order to allow for a certain degree of extensibility. Such an interface has to consider the different characteristics of each storage structure type and the corresponding component (e.g. maintaining logical addresses instead of physical records) in an appropriate way.

In our initial design, we have decided to implement concurrent update based on a general evaluation component. In our opinion, concurrent update seems to be the better solution due to the invalidation problem. Although both software structures, autonomous components and general evaluation components, have their pros and cons with respect to performance and extensibility aspects [15], we prefer the general evaluation component, since it promises better performance. However, more detailed investigations are still necessary in order to determine the best way which may be a mixture of all. In particular, the influence of the underlying hardware architecture (see chapter 2) has to be investigated in more detail.

5. Dynamic Execution Control for Parallel Actions

So far, we have described various tasks in the framework of DBMS processing which could be executed in parallel. The critical question, however, concerning dynamic execution control was postponed. How can we achieve reliable programming concepts to build the system and a safe and robust run-time environment which enables cooperation and isolation on shared resources despite the presence of failures?

As shown in Fig. 1, the dynamic flow of control for a user operation can be conveniently represented as a tree structure consisting of nested suboperations. Parallelism in a user operation means that suboperations illustrated by subtrees are executed concurrently on the same processor (multi-tasking) or on multiple processors. In such a situation, grouping of a sequence of user operations (or even one) to a conventional transaction [16] would not provide a solution to the question raised above. However, in our case it seems quite natural to refine the transaction concept and to adapt it to the processing structures to be controlled. The concept of nested transactions [17, 18] enables the encapsulation of suboperations (subtrees) thereby providing isolation among concurrent suboperations and failure confinement within a nested subtransaction (isolated recovery [19]). A nested transaction hierarchy may be explained as a collection of nested spheres of control [9] where the outermost sphere is formed by the so-called top-level transaction (TL). The classical transaction properties (atomicity, consistency, isolated execution, durability [16]) remain valid for the TL-transaction; for subtransactions, it is sufficient to provide weaker properties. In short, nested transactions guarantee atomicity and isolation (concurrency transparency); consistency of results may be delegated to ancestor transactions; durability of results (persistence) is subject to the survival of all ancestors, that is, to the commit of the TL-transaction. These properties allow a reliable program design. On the other hand, the parent-child (ancestor) relationships are used to inherit resources (uncommitted data, locks, etc.) in a flexible, but safe way which makes cooperation of subtransactions on shared data possible.

Let us quickly discuss an interface for cooperation of transactions. The invocation of a subtransaction establishes some kind of client-server relationship between caller (parent) and callee (child). Each invocation (call) issues a request to the corresponding server which acts upon the request and returns an answer to the caller. The request execution is performed as part of a transaction and is called a work step of this transaction.

To characterize the cooperation structure between parent and child transactions, we introduce the following primitives:

- work: sends a work request to a child transaction
- commit: sends a commit request to a child transaction
- abort: aborts a transaction
- work&commit: issues a work request combined with a commit request to a child transaction.

To establish a transaction (sphere of control), the first work request implicitly initializes an appropriate control structure (begin_of_transaction). For end-of-transaction or abort-transaction, explicit calls are needed. A special, but frequently used case is a so-called single call interface: work&commit provides a mechanism to create a subtransaction and then run it in an atomic manner, that is, the transaction consists of a single work step delivering either the (committed) results or the bad news (notification of abortion).

A sequence of work requests allows for a "conversation" between parent and child. The first invocation creates the transaction and the last answer confirms the commit of the created transaction. Such a conversational interface is necessary to save the state and the resources of a subtransaction across multiple work requests; for example, traversing an access path by means of a scan implies such a state-saving conversation.

These cooperation primitives can be conveniently embedded into our PRIMA run-time environment. They are to be invoked asynchronously and expressed by means of our "CALL service" operation sketched in 2.3.

A key observation determining the structure of the transaction tree is the following: Two or more calls initiated concurrently at a given level of processing must be isolated from each other (together with their included server activities) because they may operate on shared data. Since the transaction concept guarantees isolated execution, it is a good idea to encapsulate such a concurrent call (or a group of subsequent calls) by a subtransaction.

Let us discuss transaction nesting occurring in our system by means of an example. In the data system, a request from the application layer (MQL statement) is decomposed into a number of suboperations which form a subtransaction T1. The delete operation of a molecule (example 4) results in groups of atom operations which may be invoked in parallel (e.g. delete (123), delete (124), delete (134)). Hence, each of these access system requests (execution of atom operations) is encapsulated by a subtransaction. As discussed earlier, the access system may apply a concurrent update strategy in case of redundant atom representations. Therefore, a single delete operation in turn, could provoke multiple parallel operations to remove such redundant structures. The consequence is another level of nesting of the transaction. Fig. 8 illustrates part of the corresponding transaction tree; it does not reflect, however, that the children of a transaction may be executed at the same time

Note that subtransactions need not enforce consistency; in our example, all of them delegate consistency preservation (e.g. referential integrity) to their ancestor transaction T1. Such a flexible cooperation was enabled by our consistency management and the definition of suitable units of consistency.

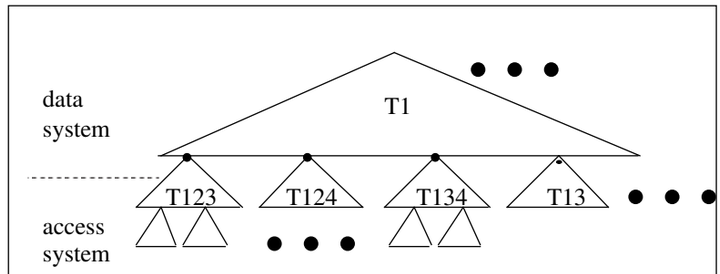


Figure 8: Transaction tree for a single MQL statement

Our implementation of transaction management (TAM) cannot be described in detail. A transaction tree is managed by TAM which creates process for remote subtransactions and handles inheritance of locking and logging resources. More information is found in [20].

6. Conclusion

We have presented a discussion of the essential aspects of parallel query processing on complex objects. The focus of the paper has primarily been on the investigation of a multi-layered NDBS to achieve reasonable degrees of parallelism for a single user query.

For this objective, we have derived several design proposals embodying different concepts for the use of parallelism. In the data system, intra- and inter-molecule parallelism were explored. To exploit the former kind of parallelism seems to be more difficult because it turns out that it is very sensitive to the optimal degree of parallelism which may vary dynamically depending on the complex object characteristics. The latter concept is considered more promising because it allows simpler solutions. Both concepts may be greatly supported by a flexible consistency management. In the access system two approaches were investigated. Deferred update seems to provoke more problems than the solutions it might provide whereas concurrent update on redundant storage structures seems to incorporate a large potential for successful application. A nested transaction concept tailored to the particular client-server structures was proposed as a prerequisite for safe activation and execution of concurrent requests on shared data.

Currently, we have finished the PRIMA implementation (single user version) and are integrating the proposed concepts for achieving parallelism in order to have a testbed for practical experiments. Performance analysis will reveal their strength and weaknesses at a more detailed level.

In the future, we wish to investigate further concepts for exploitation of parallelism. Another possibility of parallel execution on behalf of a single user would be the simultaneous activation of multiple requests within the application; for example, by means of a window system a user could issue several concurrent calls inherently related to the same task in a construction environment. Other possibilities to specify concurrent actions exist in the application layer where a complex ADT operation could be decomposed into compatible (non-conflicting) kernel requests. Usually multiple kernel requests are necessary for the data supply of an ADT operation; hence, these data requests can be expressed by MQL statements and issued concurrently to kernel servers when they do not conflict with each other or do not require a certain precedence structure.

References

[1] Dittrich, K.R., Dayal, U. (eds.): Proc. Int. Workshop on Object-Oriented Database Systems, Pacific Grove, 1986.

[2] Duppel, N., Peinl, P., Reuter, A., Schiele, G., Zeller, H.: Progress Report #2 of PROSPECT, Research Report, University Stuttgart, 1987.

[3] Special Issue on Database Machines, IEEE Transactions On Computers, Vol. C-28, No. 6, 1979.

[4] DeWitt, D., Gerber, R., Graefe, G., Heytens, M., Kumar, K., Muralikrishna, M.: GAMMA - A High Performance Dataflow Database Machine, in: Proc. VLDB 86, pp. 228-237.

[5] Neches, P.: The Anatomy of a Database Computer System, in: Proc. IEEE Spring Complon, San Francisco, Feb. 1985.

[6] Lorie, R., Daudenarde, J., Hallmark, G., Stamos, J., Young, H.: Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience, IBM Research Report, RJ 6165, San Jose, CA, 1988.

[7] Mitschang, B.: Towards a Unified View of Design Data and Knowledge Representation, in: Proc. Second Int. Conf. on Expert Database Systems, Tysons Corner, Virginia, 1988.

[8] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. VLDB 87, pp. 433-442.

[9] Davies, C.T.: Data Processing Spheres of Control, in: IBM Systems Journal, Vol. 17, No. 2, 1978, pp. 179-198.

[10] Tandem Systems Review, some papers on performance aspects of transaction systems, Cupertino, CA, 1986.

[11] Shoens, K.: The AMOEBA Project, Proc. IEEE Spring Comput. Conf., San Francisco, 1985, pp. 102-105.

[12] SEQUENT Solutions: Improving Database Performance, Sequent Computer Systems, Inc., 1987.

[13] Freytag, J.C.: A Rule-Based View of Query Optimization, in: ACM SIGMOD Annual Conference, 1987, pp. 173-186.

[14] Kotz, A., Dittrich, K., Mülle, J.: Supporting Semantic Rules by a Generalized Event/Trigger Mechanism, in: Advances in Database Technology EDBT 1988, Venice, LNCS 303, Springer, 1988, pp. 76-91.

[15] Carey, M. (ed.): Special Issue on Extensible Database Systems, Database Engineering, Vol. 10, No. 2, 1987.

[16] Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery, in: ACM Computing Surveys, Vol. 15, No. 4, 1983, pp. 287-317.

[17] Moss, J.E.B.: Nested Transactions: An Approach to Reliable Computing, M.I.T. Report MIT-LCS-TR-260, M.I.T., Laboratory of Computer Science, 1981.

[18] Weikum, G., Schek, H.J.: Architectural Issues of Transaction Management in Multi-Layered Systems, in: Proc. VLDB 84, pp. 454-465.

[19] Härder, T., Rothermel, K.: Concepts for Transaction Recovery in Nested Transactions, in: Proc. ACM SIGMOD '87 Conf., San Francisco, 1987, pp. 239-248.

[20] Härder, T. (ed.): The PRIMA Project - Design and Implementation of a Non-Standard Database System, Research Report No. 26/88, SFB124, University Kaisers lautern.