In: LNCS 503, PRISMA Workshop, Nordwijk, Holland, Springer, 1990, pp. 71-98..

An Approach to Implement Dynamically Defined Complex Objects

T. Härder
University of Kaiserslautern, Dept. of Computer Sciences, D-6750 Kaiserslautern, West Germany

Abstract

Conventional data models embodied by current database management systems (DBMS) do not provide satisfactory support for emerging applications. A major reason for this deficiency is the absense of concepts for complex object processing. In this paper, we explain the motivation and key properties of a new data model explicitly designed for the management of complex objects. Furthermore, the most important design decisions and implementation concepts for complex objects are discussed, as far as they were realized in the PRIMA project. Finally, we describe a nested transaction concept enabling intra-transaction parallelism when complex objects have to be retrieved or manipulated.

1. Introduction

Today's DBMS are unable to meet the increasing requirements of emerging applications that would like to use a DBMS. Such applications including CAD, VLSI design, geographic information management, etc. are often called non-standard applications. To improve this situation, a new generation of DBMS architectures adjusted to the demands of the applications has to be developed.

For this purpose, many researchers have analysed the data management needs of a spectrum consisting mainly of engineering applications and have encountered both a modeling and a processing problem in today's DBMS [Da86, DD86, DE84, LK84, SR86, WSSH88]. An important reason for both problems is the lack of adequate support for complex objects [". . . support for molecular objects should be an integrate part of future DBMSs . . ." [BB84]). For our discussion, the notion of complex objects (molecules) is used to indicate that such objects have an internal structure maintained by the DBMS and that access to the object is provided as a whole as well as to its components.

[BB84] has classified the 'molecular' objects according to their structure, leading to disjoint/non-disjoint and recursive/non-recursive complex objects. We have argued elsewhere [HMMS87, Hä89, Mi89] that the least restrictive or most general classification properties **non-disjoint**/**recursive** are indispensable for a data model sufficiently useful for the broad class of non-standard applications. In order to achieve refined and accurate modeling as well as efficient processing for complex objects, the data model and its implementation should offer [HMS90]

- genuine and symmetric support for **network structures** (sharing of sub-objects in contrast to hierarchical structures, which are just special cases thereof), or even recursive structures,
- support for **dynamic** object definition in combination with
- **powerful**, yet **efficient manipulation** facilities.

Before we outline the molecule-atom data model (MAD model) as our solution to the handling of complex objects in sect. 3, we explain why the relational model fails to provide the above mentioned capabilities. Sect. 4 discusses the major implementation problems related to complex objects within the architectural framework of the PRIMA project [Hä88]. The main properties of a nested transaction concept are sketched in sect. 5, which supports parallelism in operations on complex objects motivated in sect. 6. Finally, we present some conclusions and give an outlook as to our current application areas.

2. Why not the relational model ?

In order to investigate the weaknesses of the relational model for engineering applications, we start with a simplified example from solid modeling. Fig. 1 illustrates the Entity/Relationship diagram [Ch76] and a mapping example for the representation of 3D-objects. Note, we have skipped a number of additional entity types such as track and relationship types (between Point and Face or Point and Volume) to keep the example tractable. The recursive relationship on Volume is only used to demonstrate the mapping of recursive structures to the data model. In the following, we mainly focus on the non-recursive part (from Volume to Point).

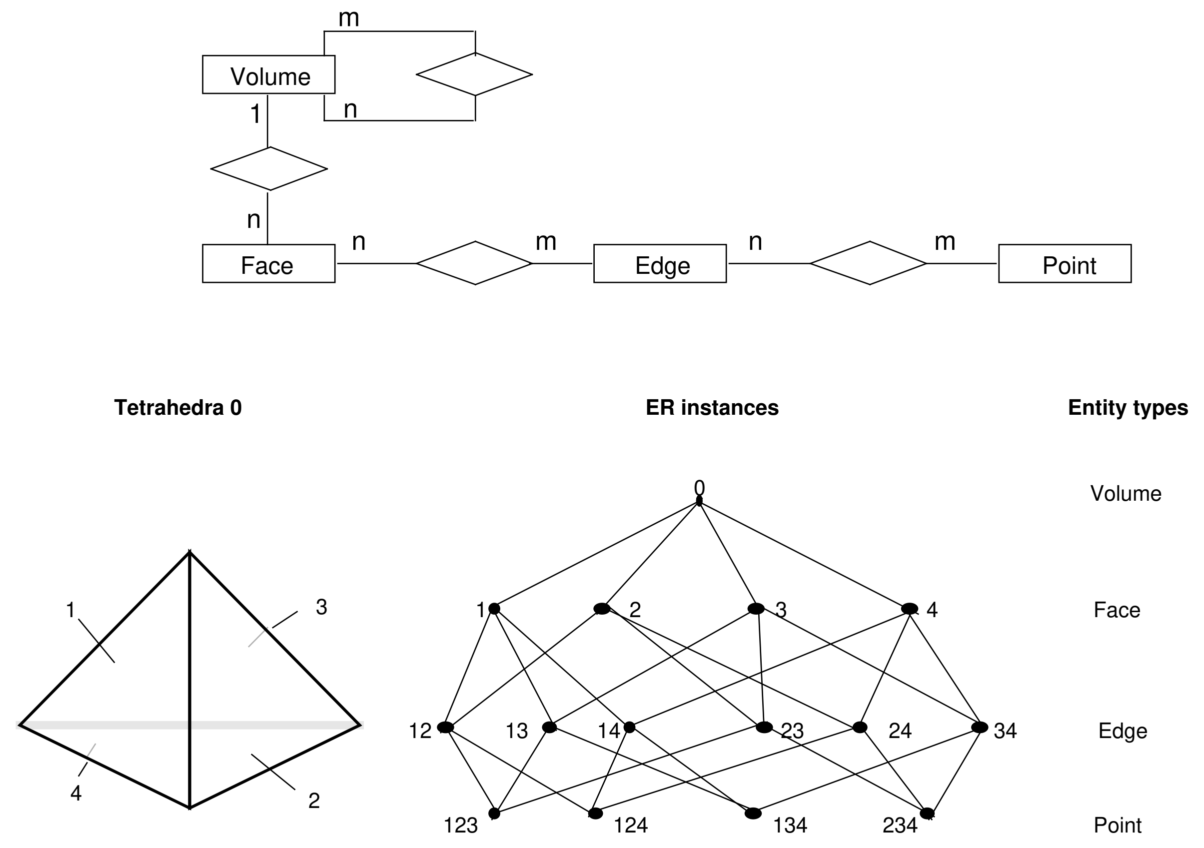


Fig. 1: Reference example: ER diagram and ER instances for a simple solid

The relational model offers the concept of normalized relations, primary key, and foreign key to map entity and relationship types. Primary/foreign key pairs are used to represent the relationships in the model. Since only functional relationships may be covered by primary/foreign key pairs, every (n:m)-relationships has to be replaced by two (artificial) (1:n)-relationships. Such keys are always symbolic values (user-defined) which may have some critical impact on join, integrity checking, or search operations. As far as model-inherent integrity preservation is concerned, the so-called relational invariants should be guaranteed by the system.

Our reference example of Fig. 1 may be syntactically translated into a relation DB schema by assigning each entity type and each (n:m)-relationship type to a separate relation and by representing each (1:n)-relationship type by a pair of primary/foreign keys. The result is shown in Fig. 2 together with a sample database for the representation of the simple solid Tetrahedra 0. To indicate the functional relationships established by primary/foreign keys, we have used dotted lines at the instance and at the type level.

Schema definition statements

```
CREATE TABLE Volume
   (vid            :  INTEGER,
   descriptor      :  CHAR(20),
   numfaces        :  INTEGER,
   usage           :  CHAR(25),
   PRIMARY KEY (vid));

CREATE TABLE Face
   (fid            :  INTEGER,
   forientation    :  INTEGER,
   numedges        :  INTEGER,
   vref            :  INTEGER,
   PRIMARY KEY (fid),
   FOREIGN KEY (vref) REFERENCES Volume);

CREATE TABLE Edge
   (eid            :  INTEGER,
   etype           :  CHAR(5),
   PRIMARY KEY (eid));

CREATE TABLE Point
   (pid            :  INTEGER,
   x, y, z         :  INTEGER,
   PRIMARY KEY (pid));
```

```
CREATE TABLE VolStructure
   (u_vid          :  INTEGER,
   l_vid           :  INTEGER,
   PRIMARY KEY  (u_vid, l_vid),
   FOREIGN KEY (u_vid) REFERENCES Volume,
   FOREIGN KEY (l_vid) REFERENCES Volume);

CREATE TABLE FE_Rel
   (fid            :  INTEGER,
   eid             :  INTEGER,
   PRIMARY KEY  (fid, eid),
   FOREIGN KEY (fid) REFERENCES Face,
   FOREIGN KEY (eid) REFERENCES Edge);

CREATE TABLE EP_Rel
   (eid            :  INTEGER,
   pid             :  INTEGER,
   PRIMARY KEY  (eid, pid),
   FOREIGN KEY (eid) REFERENCES Edge,
   FOREIGN KEY (pid) REFERENCES Point);
```

Representation of Tetrahedra 0 **Relations**

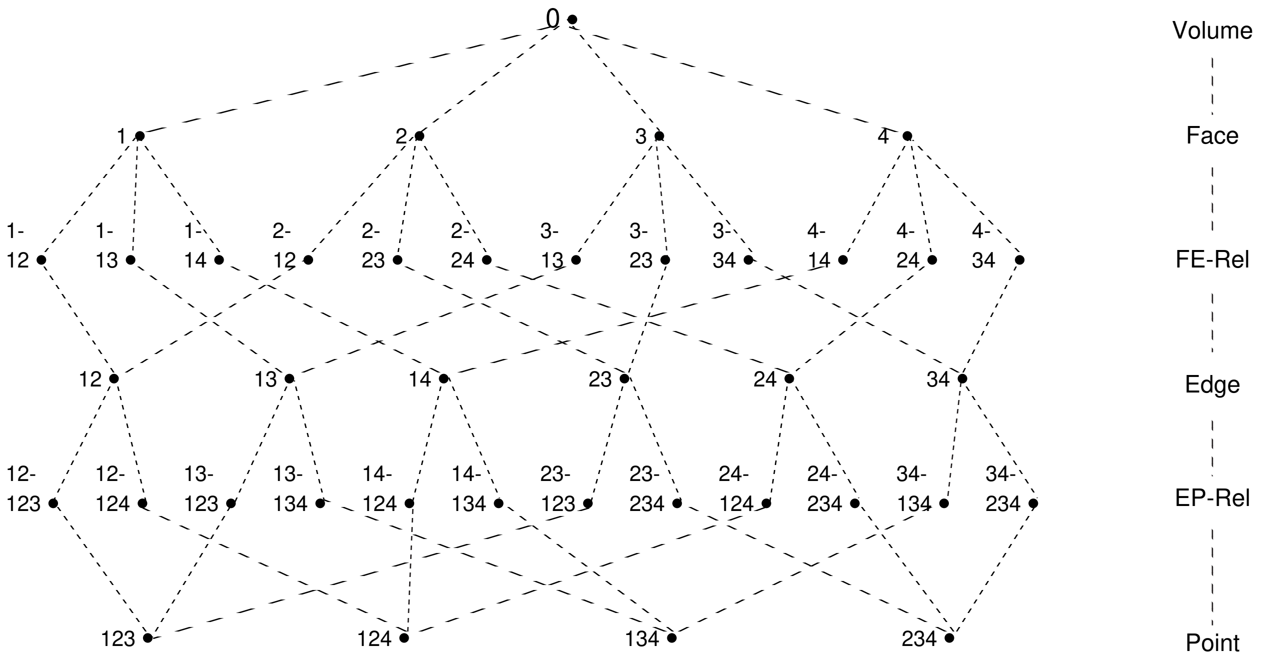


Fig. 2: Reference example represented in the relational model

As one can see even in this simplified example, the tuples representing Tetrahedra 0 are spread across many (six) relations. More complex CAD objects would require the use of about 20 or more relations [Hä89]. Since the data model knows only relations as objects, more complex structures cannot be dealt with. It can only provide an 'atomized' view of the application entities, that is, all model-inherent operations deal with tuples and relations (closure property), but cannot derive structured objects. For example, the view of the solid as an integral object including its representation, manipulation, and integrity preservation has been lost. When complex objects have to be handled, a component above the data model interface (e.g. application) has to perform this task using relational operations.

For this reason, access to complex objects has to be simulated by means of SQL operations applied to sets of independent relations. Let us assume that the Face object with fid < 3 together with the related Edges and Points has to be fetched. Since the result of an SQL operation is a (homogeneous) relation, such a request is not feasible (without loosing the object structure). But we may, for example, obtain all Points belonging to Face objects (with F.fid < 3), thereby explicitly reconstructing the complex Face object by means of user-specified joins:

```
SELECT    F.fid, F.forientation, F.numedges P.x, P.y, P.z
FROM      Point P, EP-Rel S, Edge E, FE_Rel T, Face F
WHERE     F.fid < 3
    AND   P.pid = S.pid          /*  reconstruction of  */
    AND   S.eid  = E.eid         /*  complex objects    */
    AND   E.eid  = T.eid         /*  at                 */
    AND   T.fid  =  F.fid;       /*  run time           */
```

The relational model permits the use of relationships in both directions. If we want to access all Faces associated with Point (50,44,75), we cannot refer to the corresponding Point object with all related Edges and Faces (a complex object with inverse nesting as compared to the previous case), but we may again derive the object structure explicitly:

```
SELECT    F.fid, F.forientation, F.numedges
FROM      Point P, EP-Rel S, Edge E, FE_Rel T, Face F
WHERE     P.x = 50 AND P.y = 44 AND P.z = 75
    AND   P.pid = S.pid          /*  reconstruction of  */
    AND   S.eid  = E.eid         /*  complex objects    */
    AND   E.eid  = T.eid         /*  at                 */
    AND   T.fid  =  F.fid;       /*  run time           */
```

It is interesting to note that the same sequence of join operations has to be used in both cases, although both object structures (Face-Edge-Point and Point-Edge-Face nesting) are quite different.

More arguments and observations may be found for modeling and processing weaknesses of the relational model. To shorten the discussion, let us summarize our criticism:

- The structure of complex objects is not preserved when mapped to relations; hence, it cannot be used for retrieval, update, or integrity checking.
- (n:m)-relationships cannot be directly represented; their replacement by two (1:n)-relationships leads to ponderous modeling and increase of type-crossing operations.
- Referential integrity checking may be expensive due to missing operational support.
- Explicit reconstruction of complex objects at run-time causes significant overhead due to the number and type of join-operations (based on symbolic values).
- Neither recursive object structures nor recursive search operations (e.g. transitive closure) are supported.

As a consequence, the application has to do almost everything in a very expensive way as far as complex object handling is concerned.

3. A Data Model for Complex Objects

So far we have explained why complex object management is not possible in the relational model because of inadequate operations and missing structures to support object orientation. Simulation of an equivalent behavior, however, is incomplete (lack of structure) and neither effective nor efficient as shown in [Hä89]. For this reason, we advocate a powerful data model supporting complex objects; its key properties were encountered by several prototype studies which were directed towards database requirements for engineering applications.

3.1 Desired Data Model Properties

Our data model should exhibit structural object orientation allowing the system to utilize the structure information to derive or manipulate the complex object as a unit and to maintain the referential integrity of the structure. The integration of application-oriented (behavioral) semantics and integrity into the model was not considered in order to avoid overloading the model with application-specific aspects; it could be added by a so-called application layer operating on top of the data model interface [HM90]. The result of our prototype studies revealed important access and processing characteristics of engineering applications [Hä89] which influenced the design objectives of our data model:

- **Non-disjoint object representation:** Complex objects may share common subobjects. Whenever a relationship is of type (n:m), such component sharing occurs. This type of relationship is frequent in engineering applications and, therefore, critical for modeling tasks. If relationship representation is restricted to functional ones (1:1, n:1), object representation is disjoint; however, modeling accuracy and completeness are by no means satisfying the requirements of advanced applications. Hence, non-disjoint object representation is essential.
- **Recursive objects:** Complex objects are called recursive if they are composed of objects of the same type, e.g. solids are built using previously constructed solids. Besides (n:m)-relationships recursiveness seems to be a distinct characteristic of engineering applications. Hence, recursive object definition as well as object manipulation by the system avoids tedious and ineffective object manipulation by the application.
- **Dynamic definition of complex objects:** Static definition of complex objects means that they are defined in the DB schema (frozen at schema definition time). Such static objects are considered to occur rather infrequently. Typically, each algorithm has tailored views of the complex object it is processing. Some solid-construction algorithm may require a particular face object along with the corresponding edges and points (Face-Edge-Point), whereas another task may refer to just the inverse object nesting, i.e. a point object with its adjacent faces and edges (Point-Edge-Face). Hence, design flexibility is greatly enhanced by means of dynamic object definition, e.g. in the user query.
- **Symmetry of relationship representation:** Dynamic object definition implies dynamic object derivation. Since all relationships specified in the DB schema may be used in either direction for object definition, all of them should be represented in such a way that they can be efficiently traversed in both directions (symmetrical representation). For performance reasons, (n:m)-relationships should be mapped directly (without special connection record) to minimize type-crossing operations (e.g. joins).

It was our main goal to incorporate all these properties in our data model.

3.2 The Molecule-Atom Data Model

In the following, we introduce the essential characteristics of the molecule-atom data model (MAD model [Hä88]) and show how the design objectives were obtained. The concepts of the relational model help us to explain similarities and differences, when mapping entity and relationship types of the real world using the concepts of the data model:

- Relations are named **atom types** and tuples are now called **atoms**, which represent entities of the real world.

- All relevant relationships between entity types are explicitly specified in the DB schema and represented in the DB. As opposed to this, the relational model relies on the foreign key/primary key concept.
- Relationship types, simply called **link types**, are represented in an explicit and symmetrical way. As a result, the DB schema consists of undirected networks of atom types.
- Atoms are connected to one another by **links** according to the link types specified in the DB schema. As an important consequence, the DB can be viewed as an undirected network of atoms.
- Atoms consist of **attributes** of various data types, are uniquely identifiable, and belong to their corresponding atom types. The data types of the attributes, however, can be chosen from a richer selection than those in the relational model. The types **RECORD**, **ARRAY**, and the repeating-group types **SET** and **LIST** yield a powerful structuring facility at the attribute level.
- Two special types serve for the realization of links between atoms. Atom identification is achieved by the **IDENTIFIER** type which is implemented by a system-supplied surrogate concept. Based on this type the **REFERENCE** type was defined and provides a list of identifier values belonging to atoms of exactly one atom type.

Fig. 3 illustrates as an example, the schema definition for solids and an atom network representing the simple solid Tetrahedra 0 (the recursive relationships on Volume atoms are not shown). By comparing it with Fig. 1, it becomes obvious that the MAD model can be perceived as a direct mapping (implementation) of the ER model. As illustrated by the schema definition, each link type (e.g. between atom types Volume and Face) is specified by a pair of REFERENCE_TO attributes, one in each atom type involved (e.g. attribute fref of Volume and attribute vref of Face). A link between two atoms (e.g. between 0 and 3) is represented by the corresponding values of the REFERENCE_TO attributes forming the link type; they contain the IDENTIFIER values of the atoms they are referencing (e.g. vref of atom 3 stores the vid value of atom 0 and, in turn, fref of atom 0 the fid value of atom 3 and, of course, 1,2,4). As indicated by the example, our data model supports the concept of cardinality restrictions for link types. An AT_LEAST and an AT_MOST value specify the minimum and maximum number of references which each atom may contain for the corresponding link type.

Obviously, all kinds of relationships (1:1, 1:n, n:m) as well as recursive relationships on the same entity type (e.g. on Volume) can be directly mapped by this concept. Hence, (n:m)-relationships no longer require a decomposition and two (1:n)-mappings, as is necessary in the relational model. Furthermore, convenient information such as numfaces and numedges in the relational DB schema (see Fig. 2) is redundant in the sense that the number of faces and edges can be extracted by searching the Face and FE-Rel relations respectively. As a consequence of our link representation, we do not need to represent this information by attributes; the operation COUNT applied to V.fref and F.eref quickly delivers the current values thereby avoiding the introduction of hidden redundancy. As far as referential integrity is concerned, the system controls all defined link types (pairs of REFERENCE_TO attributes) together with the associated cardinality restrictions. If a link is inserted, modified or deleted by updating the corresponding REFERENCE_TO attribute value, the appropriate back references are automatically adjusted. The specification of the counterpart attribute in the REFERENCE_TO clause is not necessary if only a single link is involved in an atom type. If multiple links are present, this specification is needed to avoid ambiguities. It permits the fast location of the corresponding counter references necessary for checking the cardinality restrictions and for link modification operations.

Schema definition statements

```
CREATE ATOM_TYPE Volume (V)
   (vid           : IDENTIFIER,
    description   : CHAR(20),
    u_vid         : REFERENCE_TO (V.l_vid (0,*),
    l_vid         : REFERENCE_TO (V.u_vid) (0,*),
    usage         : CHAR(25),
    fref          : REFERENCE_TO (F.vref) (4,*));

CREATE ATOM_TYPE Edge (E)
   (eid           : IDENTIFIER,
    etype         : CHAR(5),
    fref          : REFERENCE_TO (F.eref) (2,*),
    pref          : REFERENCE_TO (P.eref) (2,2));

CREATE ATOM_TYPE Face (F)
   (fid           : IDENTIFIER,
    forientation  : INTEGER,
    vref          : REFERENCE_TO (V.fref) (1,1),
    eref          : REFERENCE_TO (E.fref) (3,*));

CREATE ATOM_TYPE Point (P)
   (pid           : IDENTIFIER,
    eref          : REFERENCE_TO (E.pref) (3,*),
    x, y, z       : REAL);
```

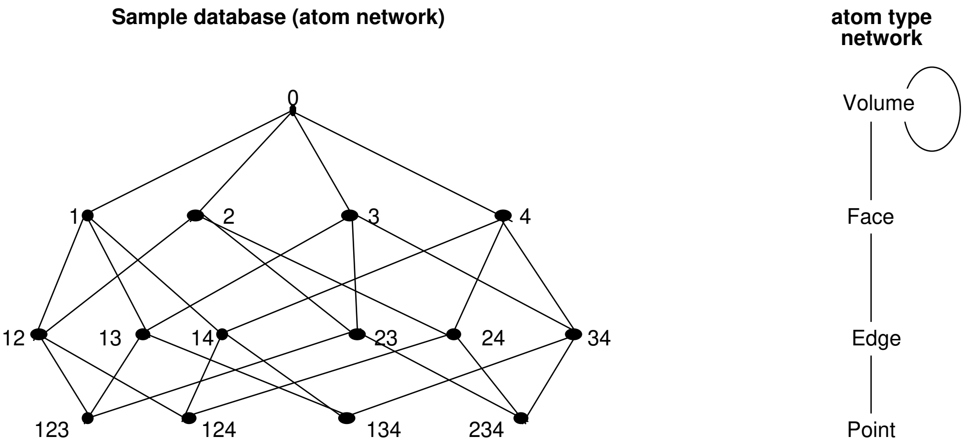


Fig. 3: Reference example represented in the MAD model

The direct and symmetric representation of relationships by bidirectional links establishes the basis for the model's flexibility. Based on the atom networks, complex objects (molecules) are dynamically definable as higher level objects which are viewed as structured sets of interconnected and possibly heterogeneous atoms. Their structure is described by a directed connected subgraph of the DB schema whose nodes are the atom types involved and whose edges are the link types to be used. For example, Face-Edge-Point specifies the structure of molecules where the relationships Face-Edge and Edge-Point are exploited to derive the molecule set. The structure graph must have one designated node (the **root**) from which all other nodes can be reached. Only in the case of recursive molecules is this structure graph allowed to be cyclic (e.g. Bill-of-Materials problem).

At least at the conceptual level, the dynamic derivation of the molecules proceeds in a straightforward way using the molecule structure as a kind of template, which is laid over the atom networks: for each atom of the root atom type, a molecule is derived following all links determined by the link types of the molecule structure until the leaves are reached. Hence, for each root atom a single molecule is derived (see Fig. 4). Both the molecule structure together with the corresponding set of molecules are denoted **molecule type**.

3.3 Query Facilities in MQL

The operational power of the MAD model is gained by the molecule query language (MQL) and its facilities for molecule processing. A detailed description of it may be found in [Mi88]. Here, we have to restrict ourselves to a short sum-

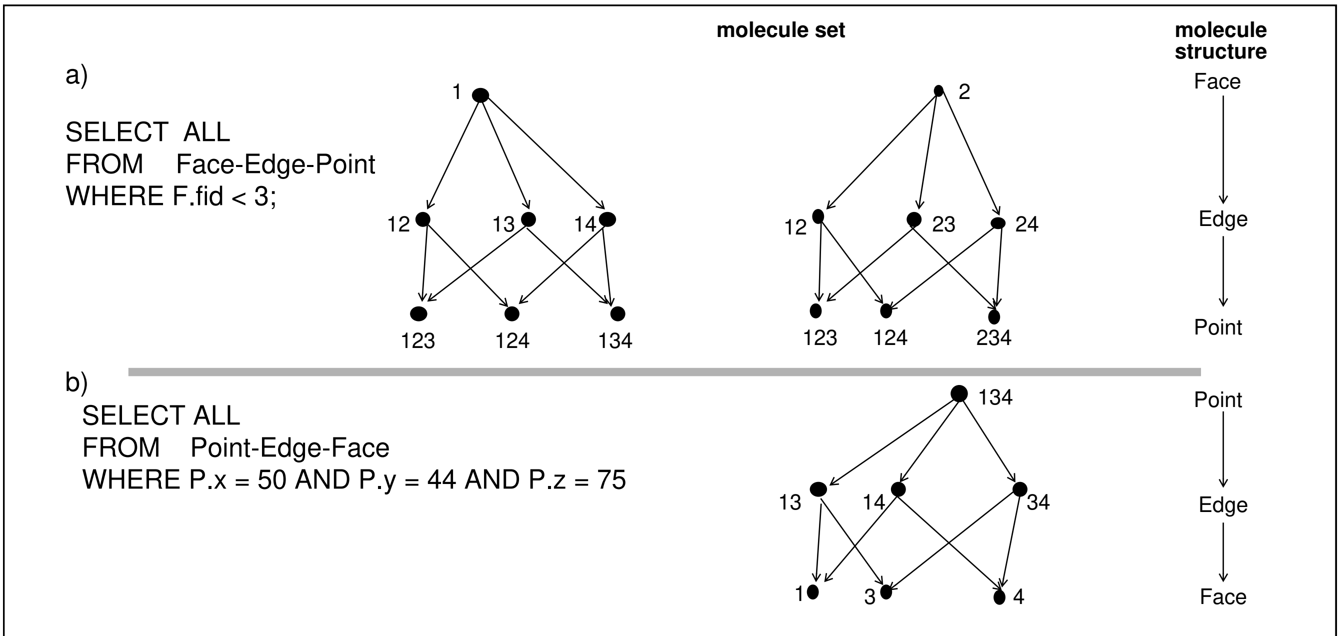


Fig. 4: Sample queries and corresponding molecule sets

mary. Similar to SQL, MQL is subdivided into three parts dedicated to data definition (DDL), load definition (LDL), and data manipulation (DML). To indicate the power of the language, we illustrate some query facilities.

Analogous to SQL, there are three basic language constructs:

- The *FROM* clause specifies the molecule type to be worked with.
- The *WHERE* clause allows for the restriction of the corresponding molecule set.
- The projection clause (i.e. the *SELECT* clause in the case of retrieval statements) defines the set of the molecule's atoms to be retrieved and is responsible for proper molecule projection.

Compared to SQL, these constructs exhibit extended semantics and syntax in accordance to the more complex objects which have to be dealt with. They form the basis of all DML-statements offered. The result of each query is also a molecule type. Thus, it can be shown [Mi89] that the closure of the MAD model under its molecule operations is guaranteed. This is a very important fact, which allows the nesting of molecule queries; each molecule-type specification can be replaced by a molecule query.

In the following, we wish to demonstrate how the key properties of the MAD model are available through MQL statements. Dynamic object definition is achieved by means of the FROM clause which determines the molecule structure to be operated upon. Fig. 4a shows the result of an MQL query which was derived from the atom network of Fig. 3. In addition, Fig. 4b illustrates the aspect of symmetric relationship representation, where Point-Edge-Face was used instead of Face-Edge-Point in the previous example.

The molecule structures in Fig. 4 form (simple) hierarchies; in the same way, network structures may be specified, e.g. the molecule type A-(B,C)-D corresponds to a diamond structure at the type level. Even recursion along a link type can be easily specified by an MQL query. One of the directed links selected for the molecule type definition is used to form a cycle in the molecule structure. Hence, the definition of a recursive molecule type consists of a subgraph called **component molecule type** and the recursion defining relationship (indicated by the keyword RECURSIVE). Molecules of a recursive type are denoted **recursive molecules** [Sch89].

Recursive molecules on atom type Volume may be defined via REFERENCE_TO attributes u_vid or l_vid by using the RECURSIVE clause, e.g.

- Volume RECURSIVE Volume.u_vid-Volume (is-used-in relationships)

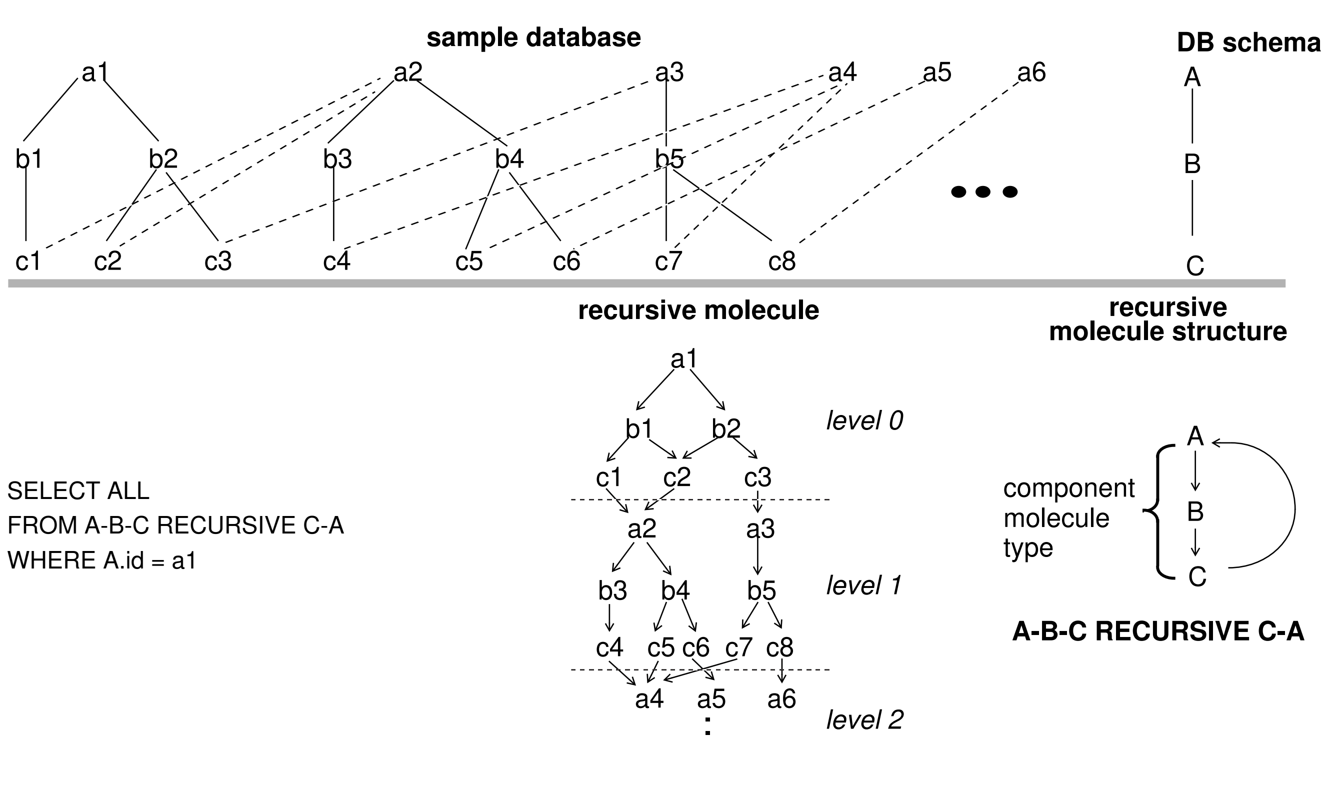


Fig. 5: Example of a recursive molecule type

- Volume RECURSIVE Volume.l_vid-Volume (is-composed-of relationships).

These molecule type definitions specify molecules which are tailored to solve problems of the Bill-of-Material type. They are derived from the atom network (of Volume atoms) in analogy to non-recursive molecules: Starting from the root atom a component molecule is built up (recursion level 0). For each leaf atom involved in the recursion defining relationship, new root atoms may be found. They trigger the derivation of the related component molecules (recursion level 1) which are appended to the resulting molecule. Depending on the network structure, this process allows for an unlimited depth of recursion. In order to guarantee termination, a component molecule belongs to a recursive molecule exactly once; if a component molecule is already in the resulting molecule, it is not included again. Furthermore, recursion depth can be controlled by special keywords referring to the recursion level or stop predicates [Sch89]. This recursive process leads to the realization of a transitive closure computation with a single starting point: A maximal directed acyclic subgraph of the atom network is formed.

Molecules representing transitive closures contain only minimal paths, which are not sufficient for the computation of some path problems (e.g. shortest paths, critical paths, etc.). For this reason, the expressiveness of MQL was extended by the REC_PATH clause and by special operators for aggregation and concatenation to specify and evaluate recursion paths. Path problems are computed by generating all maximal cycle-free paths according to the directed relationship specified in the REC_PATH clause (in place of the RECURSIVE clause). Instead of tracing all values of a REFERENCE_TO attribute at a time to derive the transitive closure, only one REFERENCE_TO value is followed for the path computation. As a result, each of the generated paths forms a molecule of its own. For details see [Sch89].

An example for recursive molecules is outlined in Fig. 5. The definition of the recursive molecule type consists of the component molecule type (A-B-C) and the recursion defining relationship (C-A). Molecule derivation is sketched for a sample DB. In this example, recursion would proceed as long as new root atoms (of type A) are found during molecule materialization.

All examples so far reflect the property of non-disjoint representation of the atom networks and the derived molecules. A molecule set is dynamically obtained as the result of an MQL query: the corresponding sets of interrelated heterogeneous record structures are then passed on to the caller of the query.

Obviously, dynamic object derivation is a performance-critical issue. It should be noted, however, that atom-type crossing operations (hierarchical joins) are less frequent and more efficient than joins in the relational model due to direct (n:m)-relationship representation and system-controlled surrogates. Furthermore, cluster mechanisms at the physical level may speed-up such operations.

4. Implementation of complex objects

So far, we have outlined our view of dynamically derived structured objects. Furthermore, we have sketched some functions of the DML to define and manipulate such objects. With these concepts in mind we can begin discussing the major issues of a DBMS which implements such a powerful data model.

Every DBMS embodies a layered architecture which is in charge of dynamically mapping the physical objects on external storage devices to the objects (molecules) visible at the data model interface. At the bottom, the database is a very long bit string stored on disk which needs to be interpreted by the DBMS code. Proceeding from the bottom up, each layer derives objects containing more structures and allowing more powerful operations. Finally, the uppermost interface supports the objects, operations, and integrity constraints of the data model. Here we refer to a hierarchical architecture with three layers, which is common in many approaches (e.g. System R [As76]). These layers are called storage system, access system and data system which we describe from the bottom up along with their mechanisms related to complex objects implementation.

4.1 Storage system

The storage system is responsible for the management of the DB files on external storage and the DB buffer in main memory as well as for the propagation of updates back to external storage. The objects, or alternatively containers, offered by the storage system are segments divided into pages of equal size. Appropriate adaptation of page size to suit the objects to be stored will reduce I/O overhead, since pages are the unit of physical data transfer. A separate page size per object, however, is unmanageable and would cause a lot of fragmentation in the DB buffer and on disk. A different page size per segment (e.g. 2^n Kbyte, $1 \leq n \leq 5$), on the other hand, may be feasible and may relieve the mapping problem for larger objects. Accordingly, the mapping between pages and disk blocks as well as buffer replacement become more complicated. Buffer management with multiple page sizes [Si88] adds a lot of complexity to replacement decisions, even for restricted page lengths (2^n Kbyte). A single buffer may achieve better memory utilization as compared to schemes with multiple buffers of the same overall size where each buffer is used for a fixed page size. However, buffer fragmentation as well as stronger sensitivity of replacement decisions to reference patterns and page sizes, make the use of such schemes at least debatable. More robust and simple buffer management is therefore gained by a static buffer partitioning, where each of the multiple buffers may be managed by a page replacement algorithm tailored to the expected reference patterns. Lower memory utilization as an argument will become less valid with the growth of memory sizes and a decrease in memory costs.

Typically, the storage system interface offers a single page upon request of the access system. When accessing larger objects or object clusters, it may be advantageous to support the concept of page sequence (or virtual page) consisting of a header page and an arbitrary number of component pages. Reference to such a compound page enables the storage system to accomplish optimization measures when it has to be read or written (e.g. parallel disk access, chained I/O).

4.2 Access system

The access system stores and maintains records (atoms with set-valued variable-length attribute values) and a variety of access path structures to speed-up different types of access. For this purpose, it uses the services of the storage system (e.g. fix and unfix page or page sequence, etc.).

Storage of records

Atoms to be stored need a flexible storage structure supporting variable length, dynamic growth or shrinking, as well as stability of reference. In our context, the following properties for storage schemes are important [SS90]:

- Modification or direct access of atoms is always performed via surrogates. Hence, the corresponding values should make fast access possible.
- Physical movements of an atom should not invalidate its external address used for reference purposes.
- Computation of the position of an attribute value within a record is performance-critical, e.g. to enable efficient sorting.
- Fast access to a single attribute value should be supported independent of whether it has a defined value or not.
- Storage size of records should be kept as small as possible to reduce storage costs as well as I/O overhead.
- Dynamic extensions of existing atom types (e.g. addition of a new attribute) should be possible without immediate modifications in the records affected.

A flexible storage scheme satisfying these requirements was proposed in [SS90]; here, we only sketch the main ideas. The description of an atom type is kept in the meta-data managed by a dedicated component. Hence, attribute description such as name, type, fixed or variable length, single- or set-valued, etc. is available for all attributes, as well as information concerning their sequence, in the atom. In this way, all values belonging to an atom can be stored in form of byte strings (value), as illustrated in Fig. 6.

The first entry of each record contains the surrogate of the associated atom (access to the record via the surrogate value has to be organized efficiently, e.g. by a surrogate translation table). The defined values information implemented by a bitmap permits the skipping of undefined attributes values. Thus, only existing attribute values are stored in the sequence specified in the meta-data. To support position computation within a record, fixed-length attributes are directly stored, whereas variable-length attributes are represented by a management entry (of fixed length) which points to the corresponding value at the 'variable' part of the record. It should be noted that this scheme tries to minimize modification as well as expansion overhead, by using a starting offset (which changes when a new value its attached) and additional relative offsets (which may be influenced when variable-length values are updated). Furthermore, attribute expansions at the atom-type level do not have an impact on the record structure, the corresponding attribute value is considered undefined as long as the record has not received it (the length of the bitmap allows such a decision).

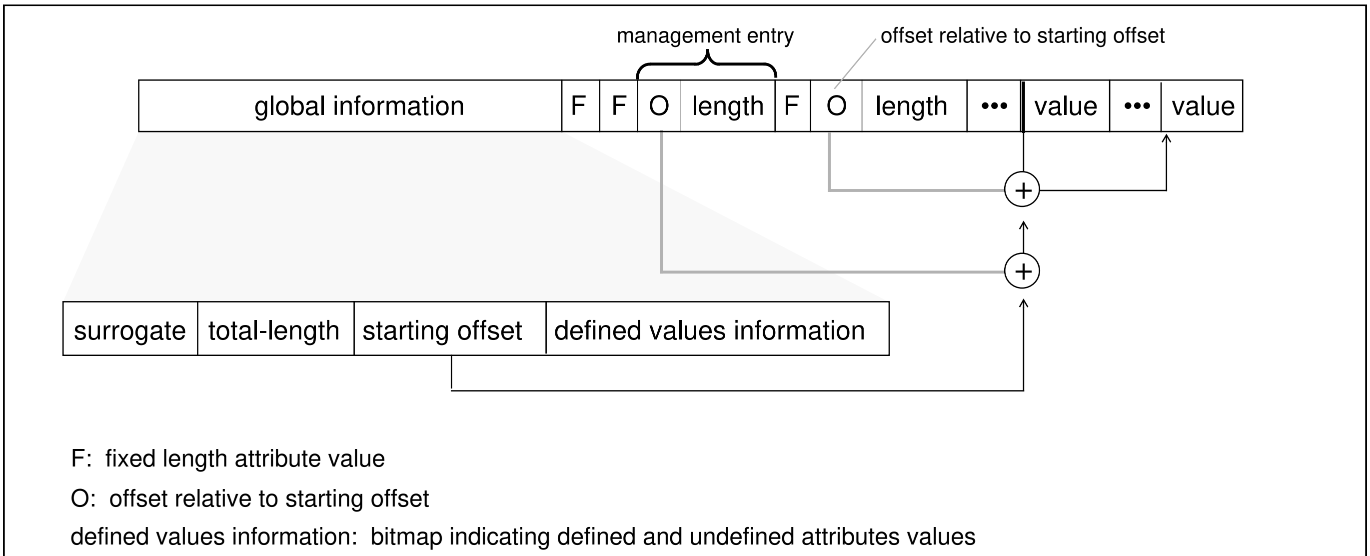


Figure 6: Storage scheme for the representation of atoms

Mapping a record to a page sequence

If the record length is smaller than the page size of the corresponding segment, it is stored within a single page and may share the page with other records. On the other hand, a record may exceed the page size which requires a page sequence to be used as a ’container’ for such a record. A straightforward approach is to consider the page sequence as a single linear address space, i.e., a record is mapped onto the page sequence as if there were no page boundaries (except the intersperse of page headers). In this case, however,

record update may become very cumbersome when displacements across pages are involved. Therefore, a mixed strategy may be beneficial, to avoid excessive modification overhead. A ’fixed’ part contains the global information, all fixed-length attribute values, and all management entries, whereas the ’variable’ part at the end consists of all variable-length attribute values which would frequently cause displacements if represented as a contiguous byte sequence. As indiciated in Fig. 7, the fixed part is consecutively mapped to the page sequence disregarding the page boundaries. The attribute values of the variable part are then attached at the end of the fixed part. Each attribute value, however, is checked as to whether or not it fits entirely into a page. If not, one or sometimes even more new pages are allocated to accommodate the values. The details of this mapping may be found in [SS90].

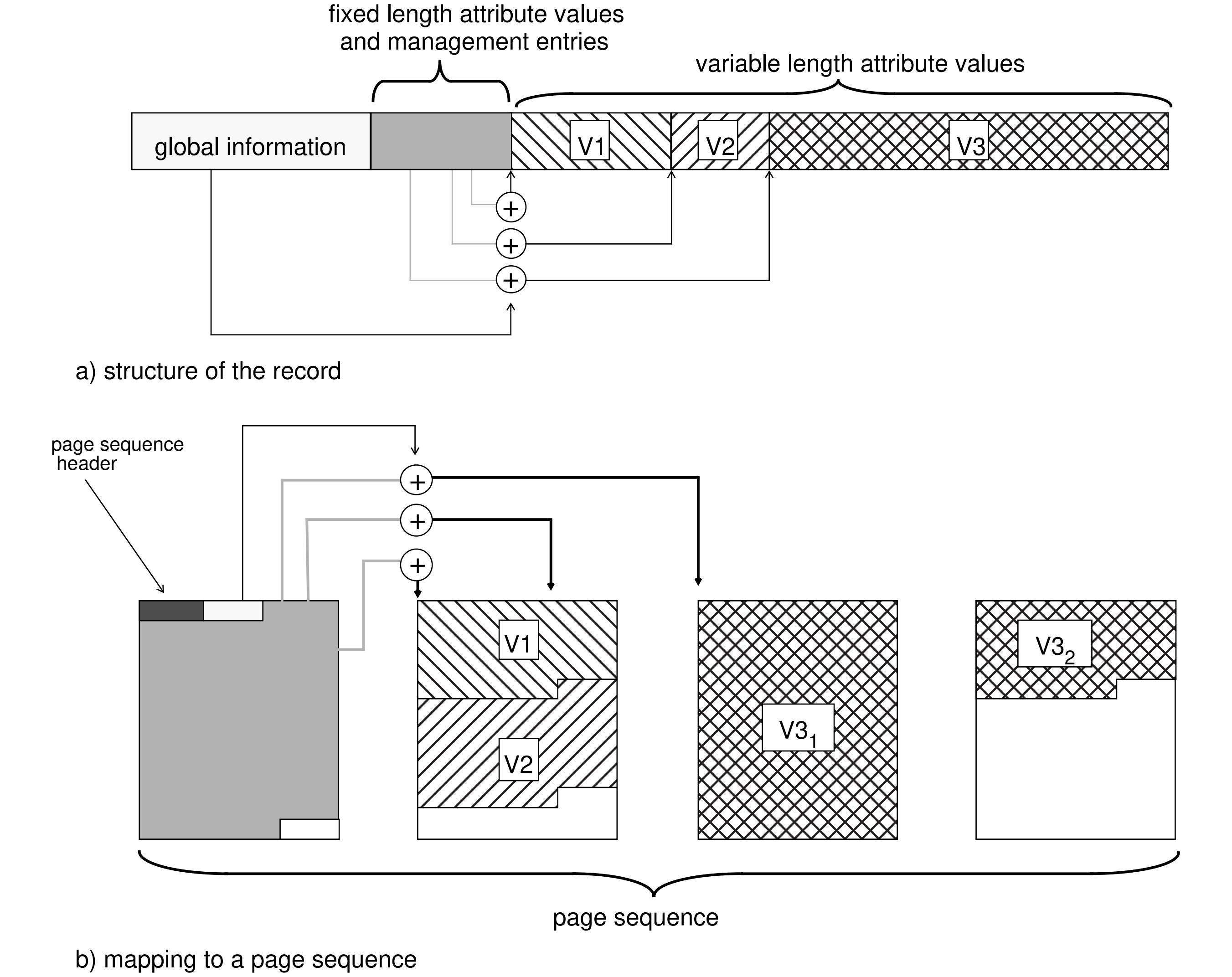


Figure 7: Mapping a record to a page sequence

Note that only one record is stored in a page sequence which strongly simplifies the management of 'long' records, especially their modifications. This storage scheme could also be extended to accommodate 'long fields' [CD86, LL89] which, however, does not directly address the issues of complex objects.

So far, we have outlined the storage structure when a single atom type is allocated to a segment. With appropriate access paths such as B*-trees, fast direct access to single atoms as well as efficient navigation through atom sets according to value-based sort orders may be accomplished. Fast access to a complex object as a whole, however, is not supported. Complex objects with a static structure (restricted to hierarchical relationships) can be easily allocated to enhance efficient access by clustering the object's components along its (unique) structure (e.g. NF^2 tuples [SPSW90]). Dynamic definition of complex objects, in contrast, leaves the selection of a particular object structure open until run-time; hence, the best we can expect is a more or less precise prediction of access characteristics to the database to define and establish appropriate storage structures (at schema definition time).

One concept to speed-up dynamic object derivation is the symmetrical relationship representation by surrogate-based links which allows for direct accessing of the counterparts in join operations; thus, it guarantees effective and relatively efficient hierarchical joins. If the atoms to be joined, however, are spread over multiple segments, quite a substantial I/O overhead has to be taken into account which makes object derivation (a multi-join process) time-comsuming. For this reason, a special storage structure useful for at least frequently requested complex objects should help to improve dynamic object derivation. In [SS89], a cluster mechanism is proposed that is expected to support the required flexibility and dynamism for complex object construction (as needed by the MAD model) with nearly the efficiency of static structure clustering.

A Storage structure for atom clusters

In order to achieve physical clustering for a set of atoms, we have to allocate them in an appropriate physical container, i.e. in a page sequence. If we cluster all atoms belonging to a specific molecule in such a physical container, molecule materialization is a perfectly local operation, that is, the page references for the required hierarchical joins are confined by the page sequence. Such a structure is called atom-cluster type; it obviously minimizes disk access time for the corresponding atom set. Hence, the key idea is to predefine atom-cluster types (by LDL statements) and to allocate the associated (heterogeneous) atom sets to page sequences such that frequently requested molecules are logically contained in these atom clusters. Here, we cannot discuss the question as to which atom-cluster types should be chosen to speed-up molecule materialization; such a decision needs a lot of application knowledge and must be addressed by the DB administrator.

Before we illustrate a storage structure for atom clusters, we wish to point out some consequences of the underlying concept. Fig. 8 and 9 show some examples of molecules which may be derived from our sample database in Fig. 3. For example, we may define an atom-cluster type Volume to Face, which clusters the atom set of Fig. 8. (An atom-cluster type is directed in the sense that the root atoms determine the atom set to be clustered.) Then, the molecule with the structure Volume-Face as well as those of Face-Volume can be derived using the atom cluster. In this case, the functional relationship between Volume and Face allows disjoint representation of atom clusters.

In Fig. 9, the atom-cluster type is again hierarchically structured, however, the relationships between Face and Edge as well as Edge and Point are (n:m). Such relationships to non-disjoint components cause redundant atom representation in the clustered atom sets. As illustrated in our example, molecule sets for molecule types Face-Edge-Point, Edge-Point and others can be materialized from the given atom-cluster type. Apparently, the molecule types Point-Edge or Point-Edge-Face cannot be completely derived by using a single atom cluster per molecule.

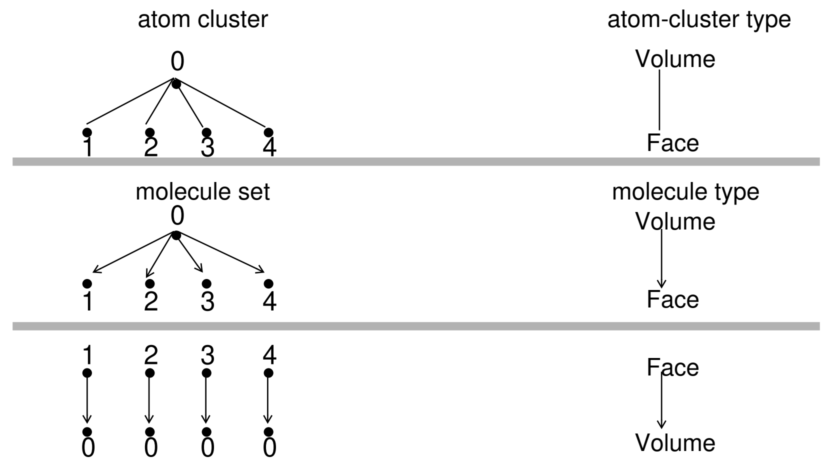


Figure 8: Example of a disjoint atom cluster and molecules derived from it

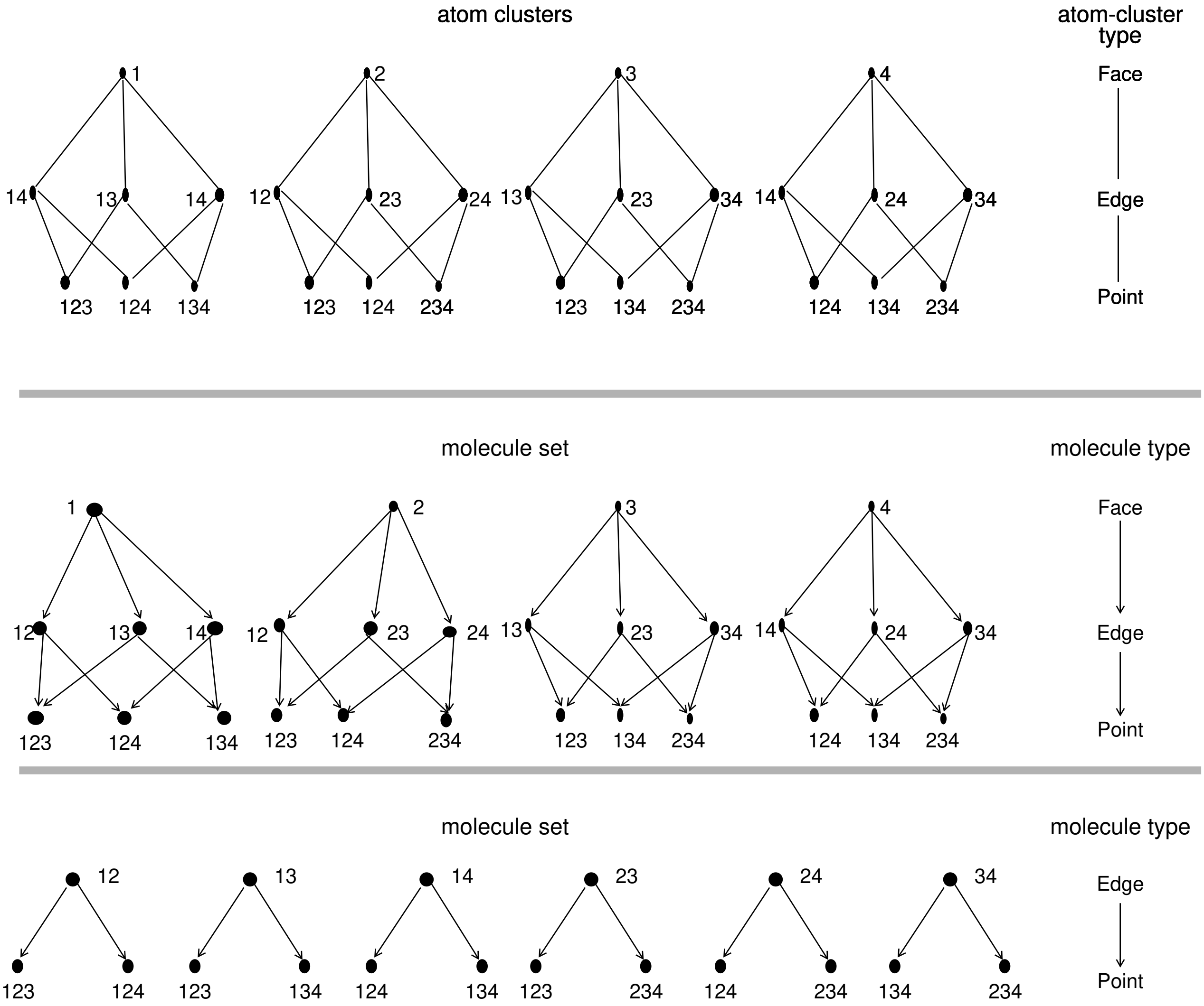


Figure 9: Example of non-disjoint atom clusters and molecule sets derived from them

[SS89] argues that only hierarchically structured atom-cluster types should be allowed; network-like or recursive type structures should be ruled out for operational reasons, complicated maintenance, and semantical interpretation problems. Nevertheless, the price for the cluster concept is redundancy maintenance if (n:m)-relationships are involved. Hence, the benefits of materialization support must outweigh the increased update costs.

Physical contiguity of all atoms belonging to an atom cluster can be effectively obtained by the following mapping (Fig. 10): Each atom cluster is described by a so-called characteristic atom which contains references to all atoms, grouped by atom types, belonging to the resp. atom cluster. Moreover, the characteristic atom keeps information for each reference to the contained atoms which is evaluated to determine update dependencies in the cluster.

Each atom cluster is mapped onto a so-called cluster record, i.e. a byte string of variable length containing the characteristic atom together with all atoms of the cluster. Although they may be referenced several times, all atoms are included only once.

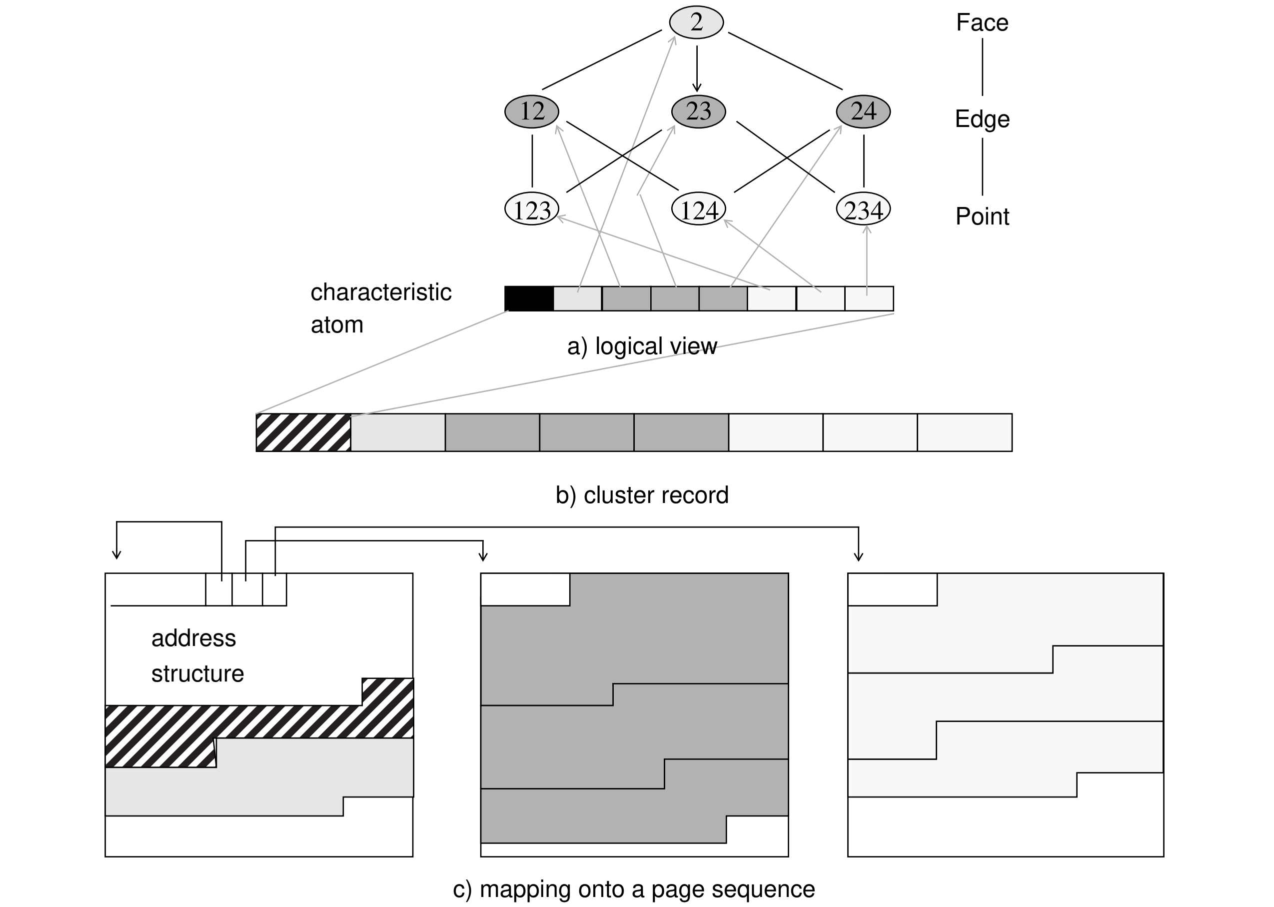


Figure 10: Example of an atom cluster

The cluster record, in turn, is mapped onto a page or a page sequence of the segment to which the atom-cluster type is assigned. If the entire cluster fits into a page, the cluster record can be stored as a byte string containing subrecords for each atom (according to the storage scheme in Fig. 6). Even multiple (small) atom clusters may be stored subsequently in a page, whereas a page sequence always contains only one atom cluster, that is, stores only a single cluster record. Such a mapping is more complicated: All atoms of a single atom type are placed into a subrecord; all subrecords are subsequently mapped onto pages thereby adjusting them to page boundaries, if necessary (see Fig. 10). Since atom clusters (or page sequences) may become large, additional address structures can help to quickly locate an atom. Single-page atom clusters are searched sequentially, whereas direct access to atoms is provided for larger clusters by keeping an address table for the subrecords in the header page of the page sequence. Subrecords which span multiple pages, in turn, have an address table for the atoms to avoid a sequential scan over these pages.

Replicated storage structures

To improve retrieval of dynamically defined sets of atoms, various forms of storage redundancy may be introduced. As explained above, atom-cluster types will cause atom representations to be replicated by a varying degree depending on the relationship structure. Furthermore, access paths such as B*-trees or grid files embody some kind of storage redundancy (at least at the level of attributes and identifiers). However, explicit replication of selected atom types may sometimes be useful to enhance retrieval performance. For example, the specification of a **sort order** provides a special storage structure keeping all atoms of an atom-type sorted according to some of its attributes; such sort orders are fully redundant because the basic storage structure (in system-defined order) always exists for an atom type.

As far as modification operations are concerned, storage redundancy has to be concealed by the access system. Hence, the update of an atom in one particular representation has to be automatically propagated to all its representations in a way that is transparent to all system components outside the access system.

Although the complexities involved in the maintenance of replicated storage structures have to be confined to the access system, the existence of such structures together with appropriate access primitives has to be made known to the data system. Otherwise the optimizer cannot select from the existing choices of access paths which would make storage redundancy useless.

The access system interface

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

Scans are a concept to control a dynamically defined set of atoms, to hold a current position in such a set, and to successively deliver single atoms or only selected attributes thereof for further processing. A scan operation is linked to a certain storage structure or access path which determine sequence and result set of atoms to be retrieved. To increase the flexibility of scans, their result set can be restricted by a simple search argument solvable on each atom and/or start/stop conditions in the case of value-based ordering of atoms. The PRIMA access system supports the following scan operations at its interface:

- the atom-type scan based on a general basic storage structure
- different access path based scans (e.g. a scan based on B*-trees)
- scans guaranteeing a certain sort order, which may either be materialized or dynamically derived
- the atom-cluster scan which operates on clusters of heterogeneous atoms.

Whereas the first three scan types support 'horizontal' access to a homogeneous atom set belonging to one atom type, the last one allows for the 'vertical' access to a heterogeneous atom set across several atom types.

4.3 The data system

The data system has to fill in the gap between the atom-oriented interface of the access system and the data model interface, which deals with complex objects. Its main task is to map the objects and operations of the MAQ model to the primitives available at the access system interface. Hence, it is responsible for the dynamic construction of molecules and for the set-oriented delivery of result sets to the requesting component.

For this purpose, the data system implements all mechanisms of complex object processing which arise from the need to handle molecules dynamically defined at query time. It translates MQL queries into an internal representation called **query evaluation plan** (QEP), optimizes the QEP, and executes it by means of access system calls in order to compute the requested result. MQL statements are normally embedded in application programs which may be executed quite frequently. An interpreter approach would, therefore, repeatedly cause the full overhead of all query processing phases. To obtain the benefits of a 'compiled' approach [LW79], we separated compilation and optimization from execution leading to three distinct phases of complex object processing:

- Compilation of the MQL statement generates a valid, but not necessarily optimal query evaluation plan and stores it within an access module.
- Optimization transforms the QEP according to given heuristics and rules, in order to find the equivalent QEP which executes the query with minimal response time. This optimization phase includes the determination of evaluation strategies and the selection of access paths.
- Execution of the access module either retrieves or modifies atoms by means of access system calls according to the optimized QEP. Instead of generating code for the QEP, we designed an interpreter for QEPs, which contain the execution sequence in the form of access system calls. To handle retrieval requests, the data system maintains a main storage data structure where it inserts and combines the requested atoms in order to build up the query

result. Upon completion, the derived molecule set is passed on to the caller in a set-oriented manner. Such an execution can be repeated independently of the first two phases.

In this paper, we cannot describe all the complexities of complex object processing [HMS90]. We would rather sketch some important issues by means of examples.

Compilation of a retrieval statement

The MQL compiler checks the user query for syntactic and semantic correctness, performs the so-called query standardization, and creates the initial QEP in an access module. Standardization includes

- the replacement of ALL in SELECT clauses by the corresponding set of attribute names
- the resolution of molecule types predefined in the DB schema (FROM clauses)
- the representation of boolean expressions in conjunctive normal form as well as the expression completion of incompletely quantified expressions in the WHERE clauses.

Let us assume a rather simple query with qualified projection in the SELECT clause referring to the database in Fig. 3:

```
SELECT  Face, Edge, (SELECT Point
                     FROM RESULT
                     WHERE Point.x > 10)
FROM    Face-Edge-Point
WHERE   Face.fid < 3
```

The resulting QEP in Fig. 11 consists of an operator graph describing the execution plan. It serves as an easy--to-understand example which illustrates the principal ideas. Of course, such QEPs may be much more complex.

It is generally possible to divide all nodes of an operator graph into two classes: the **leaf nodes** are employed to construct the (simple) molecules, whereas the **inner nodes** subsequently operate on these molecules to derive more complex structures or properties, e.g. recursive molecules, qualified projection, aggregation etc.

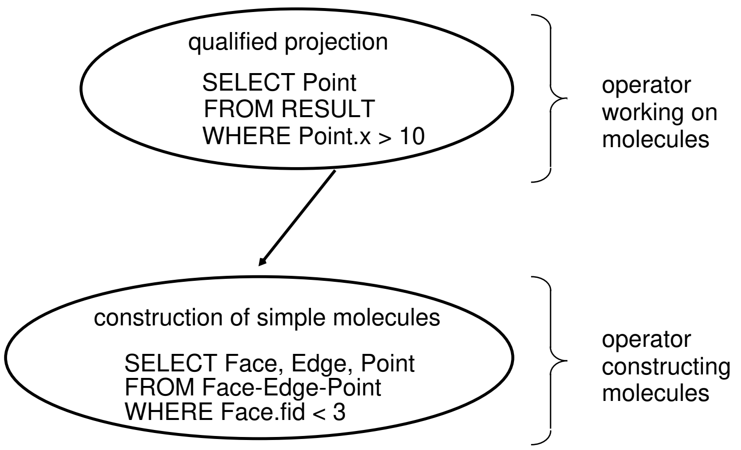


Fig. 11: Example of a simple QEP representation

All operators related to leaf nodes are of type CSM ('construction of simple molecules'); they can be used to derive molecules of the following form:

```
SELECT  <unqualified projections>
FROM    <one non-recursive, hierarchical molecule type>
WHERE   <molecule qualification Q>
```

Hence, the CSM type is in charge of selecting the qualified atoms via access system operators and of building up the (initial) molecule set by using a main memory data structure. Since leaf-to-root evaluation is always applied to the operator graphs, the operators represented by the inner nodes work on this data structure thereby accomplishing the requested result set step by step.

Optimization considerations

The optimization phase includes simplification, amelioration and finally refinement of a query [JK84]. Query refinement which is most important to our discussion considers alternative strategies for the execution of the operators in the QEP in order to find the cheapest execution plan. It should be clear so far that CSM is a performance-critical operator (besides the operator for the construction of recursive molecule) because it reads atoms from the database (as the only operator). Hence, the problem of access path selection and the algorithms for processing joins have to be considered.

We have pointed out for the relational model that reconstruction of complex objects has to be performed by general joins using primary/foreign key pairs. In our case, CSM executes specialized joins called hierarchical joins and evaluates conditions on the result. The molecule structure as specified in the FROM clause can be seen as a kind of join plan where atom types connected by a link type may be combined via the specified directed link. As opposed to the relational join, our hierarchical join is an (n:m)-join, that is, for example, each atom f of Face within a molecule Face-Edge may have several descendants e_i of type Edge which may be shared with other Face atoms. In this case, the join condition is "E.eid is contained in F.ref", where the REFERENCE_TO attribute F.ref points to atoms of type Edge. Hence, a nested loop algorithm may be efficiently applied:

```
Foreach atom f of type Face
  Foreach entry d in F.ref
     If atom e of type Edge with e.eid = d is not contained in the result's atom set
        call e from the access system via condition e.eid = d
```

Note that an atom may be shared among several molecules of a result set or may be a descendent of more than one atom within a molecule. Such cases give rise of the optimization in the inner loop.

It is clear that all atoms participating in a hierarchical join have to be fetched via access system calls. To minimize the number of these calls, the evaluation of the WHERE-clause conditions should be performed as early as possible to restrict the atom set to be investigated. Since restrictions may apply to every atom type appearing in the molecule structure, restriction evaluation is not straightforward. For a refined discussion, we refer to [HMS90]. Furthermore, the speed of a hierarchical join depends on the access paths and clusters to be used in locating the atoms in the database. Hence, CSM optimization has to perform the selection of the best available scans on existing storage structures for each of the hierarchical joins. Obviously, the use of atom-cluster scans which are tailored to the execution of hierarchical joins will greatly accelerate CSM operation.

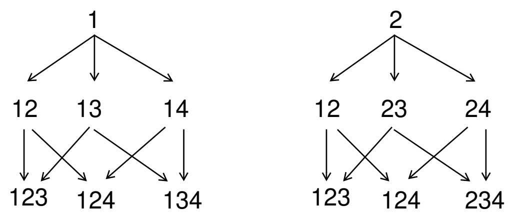
Execution of an access module

Query optimization yields the 'best' QEP in form of an operator graph which specifies the execution plan of the query still at quite an abstract level. Query execution is then performed by interpreting the operator graph node by node under control of the data system. Again, various strategies concerning sequential, pipelined, or even concurrent executing of nodes are applicable (see [HSS88]).

As mentioned earlier, the result set of a query is built up in a main memory data structure. Since distinct molecules are derived, one may get a substantial degree of storage redundancy for the representation of atoms due to the (n:m)-relationship involved. To avoid multiple copies of the same data in the resulting molecule set, we separated the representation of the molecule structure from the representation of its data (see Fig. 12). Hence, only REFERENCE_TO values of atoms are included several times to establish the structural view. To guarantee fast access to all elements, structure and atom data are organized in two tables based on extendible hashing.

Let us finish the discussion of the data system with some remarks on the materialization of recursive molecules. Each component molecule is derived just like a regular molecule using hierarchical joins, etc. These component molecules are then combined according to the recursion defining links. The hash-based data structures (Fig. 12) greatly facilitate the detection of loops in the recursion, since component molecules may be identified

structural view



each entry in the structure contains the
corresponding structure building pointers as well as a pointer
to the data tables shown on the right side

data tables

1	<data>
2	<data>

12	<data>
13	<data>
14	<data>
23	<data>
24	<data>

123	<data>
124	<data>
134	<data>
234	<data>

Fig. 12: The two components of molecule set representation: structure and data

by their root atoms. Hence, it is sufficient to check for the non-existence of the root atom in the resulting molecule, before the corresponding molecule component is materialized.

All further transformations and manipulations on recursive molecules are performed by operators corresponding to inner nodes of a QEP (see Fig. 11). Obviously, the availability of sufficient storage space in main memory determines the performance of these QEP operations.

5. Transaction Concept for Processing Complex Objects

In the previous section, we have described which implementation concepts support the requirements of complex object management and how they fit in a layered DBMS architecture. Now, we will discuss the principles of the dynamics of query processing for complex objects, that is, the transaction concept [BKK85, Gr81, KLMP84] to be employed, and some hints related to its implementation. A major objective of the transaction concept design was the desire to exploit the inherent parallelism when processing MQL operations (see sect. 6). Thus, the transaction concept should enable concurrent operations in the various DBMS layers thereby reducing the response time for a given MQL request [WS84].

Flat transactions do not provide any intra-transaction control structure to enable cooperation and isolation on shared resources and to conceal the impact of failing activities. For this reason, nested transactions [Mo81] were proposed as a control structure to achieve a safe and robust run-time environment for parallel and/or distributed processing within a transaction. A transaction is recursively decomposed into subtransactions resulting in a transaction tree with the so-called TL-transaction (top level) as the root. These subtransactions control all activities in the system. Accordingly, they can be used as the units of concurrency control as well as recovery. A suitable transaction nesting may obtain sufficiently small granules of concurrency control (e.g. locking) to enable significant intra-transaction concurrency in a safe way. On the other hand, such a transaction nesting enables intra-transaction recovery where a subtransaction (and its subordinates) can be aborted and rolled back without any side-effects to others.

Flat transactions observe the ACID principle [HR83]. As the unit of Atomicity, Consistency, Isolation and Durability, a transaction guarantees concurrency transparency among transactions as well as failure transparency. These properties must also be realized by the TL-transaction as the outermost sphere of transaction control, whereas weaker properties may be provided by subtransactions. Atomicity and isolated execution remain key properties of each subtransaction; consistency control and the responsibility for the persistence of its modified data (durability), however, may be delegated by a subtransaction to its parent transaction (and ultimately to the TL-transaction). For further discussion of nested transaction properties, we refer to the literature [HR87,Mo81,We86].

A model for nested transactions

How can we apply the transaction nesting to the complex object processing? The dynamical flow of control during the execution of a DBMS request may be characterized as a tree of procedure activations; for example, each operator within a layer or at its interface embodies such a procedure. A closer look at the typical workloads of engineering applications [Hä89] reveals that the operator trees spanned by typical requests may obtain a very large fan-out at each of our system layers. For example, consider even the simple query of Fig. 11 and assume that the CSM operator performs some atom-type scans. Obviously, subgraphs of our operator graph have to be guarded by subtransactions. The critical question is what are the appropriate granules (subgraph sizes) to balance the overhead of transaction management with the benefits of fine-grained control structures (concurrency, failure isolation). The model of nested transactions itself does not restrict the use of subtransactions to a minimum granule. Hence, we may allocate multiple transaction levels within a single system layer or we may even bracket execution paths across layer boundaries. It is, however, a good design principle to observe layer boundaries within the transaction hierarchy. Therefore, we do not permit transactions to span multiple system layers. Hence, transaction properties (atomicity, isolation) may be used to control clean and safe cooperation across layer boundaries.

Our nested transaction model is illustrated in Fig. 13. The TL-transaction consists of a sequence of MQL queries where each of them is guarded by an MQL transaction. This, in turn, can be decomposed into multiple data system transactions (depending on optimizer decisions and the inherent potential of intra-operation concurrency). Each data system transaction may invoke many access system transactions each fetching or modifying a single atom (if maximum parallelism should be achieved). On the other hand, an entire scan operation may be enclosed by a single access transaction (with a conversational interface to the parent transaction).

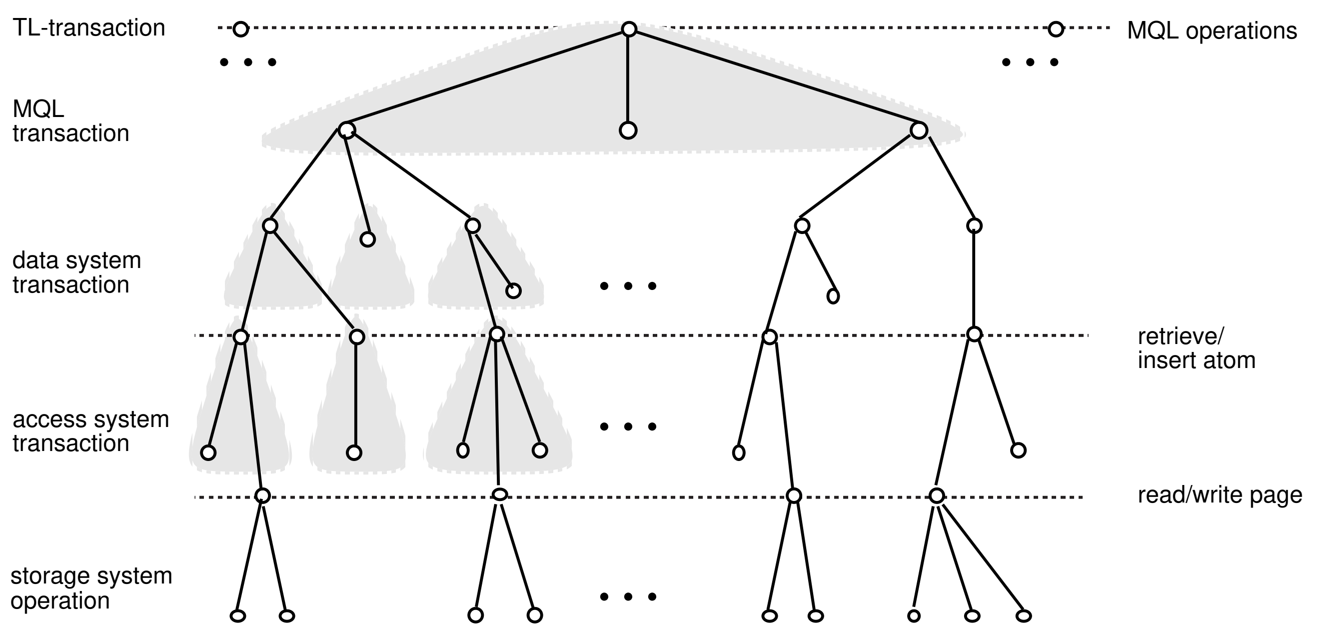


Fig. 13: Example of a transaction hierarchy to be established by an MQL operation

As indicated in Fig. 13, storage system operations are not organized as separate subtransactions, that is, I/O operations within an access system transaction are not performed in parallel. Hence, as far as concurrency and recovery issues of them are concerned, access system transactions will provide the necessary functions [HPS90].

In our PRIMA implementation, we decided to group all transaction-related services in a so-called transaction management subsystem (see Fig. 14). It consists of four major components which are jointly responsible for preserving the ACID properties during transaction execution. The transaction manager is responsible for the management of the transaction hierarchies and for accomplishing the atomicity property. Furthermore, it is acting as a coordinator to distribute information or to obtain general agreement, e.g. during commit or abort. Consistency is controlled by the consistency manager whereas concurrency control performed by the lock manager takes care of isolated execution. Finally, durability of database updates is guaranteed by the recovery manager which collects log information to cope with various failure types. For performance reasons, the services of these managers are directly invoked by a transaction as long as it proceeds normally. In the case of special events, however, all managers have to be synchronized by the transaction manager to do their job.

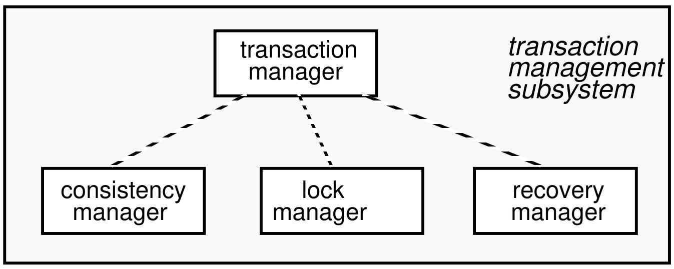


Fig. 14: Structure of the transaction management subsystem

Implementation issues of nested transaction management

In our context, we will limit our discussion to the most relevant aspects which affect complex object processing. The transaction manager offers general functions to create, commit, or abort a transaction in the framework of nested transactions. Hence, their implementation has to reflect the anticipated type of processing, that is, highly dynamic transaction trees embodying a lot of concurrent activities have to be efficiently maintained by m-ary trees (e.g. mapped to binary trees adjusted to specific traversal requirements).

Concurrency control is more strongly influenced by the properties of complex objects. If only static objects (defined in the DB schema) are involved, the checking of synchronization conflicts is comparably simple. For example, [He89] proposes an algorithm based on a hierarchical data model [SS86] which allows easy detection of common subhierarchies. With dynamically defined objects incorporating network structures, however, there is hardly a chance to find an effective conflict detection algorithm at the type level. Synchronization which is based on conflict checking of molecule structures (type graphs) would be very pessimistic and would mostly detect phantom conflicts. Similar arguments apply if the predicates (together with the type definitions) of the MQL queries are utilized. (Note, molecules are 'elusive' as far as conflict checking is concerned. They have to be materialized before component overlap can be determined.) Therefore, the only realisitic approach with tolerable performance characteristics seems to employ synchronization at the atom level. Thus, we have implemented an R-X locking protocol (with lock inheritance) which requires an explicit lock request for each atom involved in a complex object operation. When a page is accessed to fetch or modify an atom, an appropriate short-term lock (fix-phase) is acquired for the entire page or page sequence in order to prevent side-effects due to page modifications. Thus, concurrent operations of multiple subtransactions (or independent transactions) in a page are supported by our locking protocol (excluding only fix-phases).

Consistency control [Es76] becomes more and more important even in conventional DBMS, e.g. the preservation of referential integrity. This is particularly true for complex object processing where the system must guarantee consistency in situations with much more elaborate objects and operations. As indicated in sect. 4, our system provides increasing levels of abstraction where the objects, operations and integrity constraints grow more complex with each layer. Hence, our framework of nested transactions seems to fit nicely the requirements of this level-to-level control of integrity. We mentioned earlier that a subtransaction may delegate its responsibility for integrity preservation to its parents. This fact allows more complex integrity conditions to be controlled and optimized. Consider the deletion of a molecule where multiple atoms with their references and counter-references have to be deleted. Since referential integrity cannot be guaranteed for each single delete of an atom, the corresponding check operation must be deferred.

Another example is the control of the cardinality restrictions for links which sometimes require several modfication operations before they satisfy their specification. In such situations, a consistency control component [Sch90] can be used to collect the resp. information relevant for integrity control and to optimize the checking and modification overhead, e.g. an atom to be addressed by multiple references has to be located only once. The transaction hierarchy then allows flexible control for such deferred checks.

Finally, let us highlight our design decisions in the case of the recovery component. Buffer management applies a NOSTEAL policy [HR83]; thus, the abort operation of a subtransaction is a in-memory rollback by using a log of inverse operations at the atom level. To speed-up commit processing, we employ a NOFORCE scheme, which requires partial REDO of committed transactions when a system crash occurs. For this purpose, we keep a REDO log on disk; since we use atom locking (and the log granularity cannot be larger than the lock granularity), physical entry logging based on atoms was chosen. Hence, our NOSTEAL/NOFORCE scheme optimizes the normal case (and not the crash recovery) by avoiding synchronous I/O as far as possible.

6. Using Parallelism in Complex Object Management

Our design decision for the nested transaction concept was explicitly motivated by the desire to achieve intra-transaction parallelism in various tasks of complex object management. As illustrated by Fig. 13, the transaction framework allows the exploitation of inherent parallelism and the control of medium or even fine grained parallel activities in several system layers.

As discussed elsewhere [Du87, HSS89], suitable hardware configuration and run-time environments are a prerequisite for implementing these concepts. In our case, we assume a tightly or closely coupled multiprocessor architecture where database buffer, lock and log information, as well as other common data may be efficiently shared among the various processes distributed across the processors. In order to simulate such an environment in a network of SUN workstations, we have implemented the RC-System (remote cooperation) which allows efficient location-transparent communication among processes configurated as a system [HKS90].

Intra-operation parallelism

In this section, we will briefly identify the DBMS functions where it pays to generate parallel activities. Obviously, retrieval of atoms and the subsequent molecule materialization is a promising area. Since an MQL query specifies a set of molecules, one strategy is to derive all molecules in parallel (inter-molecule parallelism), that is, each molecule is handled by an own process or task. Another strategy could derive the molecule's components concurrently (intra-molecule parallelism). This strategy is enabled by our link concept where the REFERENCE_TO attribute of an atom contains the set of identifiers of atoms it is linked to. Hence, the traversal of subgraphs in the atom network may be performed in parallel. Depending on the specific molecule qualification (WHERE clause), various optimizations of search strategy are conceivable [HSS88].

Similar efforts can be employed to manipulation operations and consistency management. Again, the link concept supports concurrent activities in an obvious way. For example, the deletion of a molecule or checking of referential integrity could proceed in parallel.

Finally, the maintenance of redundancy is an area we have designed parallel algorithms for. In sect. 4, we have advocated the use of replicated atom representations (e.g. sort orders) or atom-clusters provoking redundant storage structures in order to speed-up retrieval operations and, of course, dynamic materialization of molecules.

Deferred update of replicated storage structures

The maintenance of replicated storage structures could easily cause a performance bottleneck if update operations occur too frequently. Therefore, the concept of deferred update was proposed in the literature [DLPS85] to avoid response time penalties. However, this concept requires replicated copies to be invalidated as long as they do not represent the latest value; depending on the number and type of replication, the implementation of a practical invalidation scheme may turn out to be cumbersome.

Concurrent update [HSS88] avoids the problem of maintaining invalid storage structures. Parallelism in the access system is used to keep all replicas of a modified atom up-to-date. For this purpose, an evaluation component identifies update calls to the access system and invokes concurrent operations on the various redundant structures. Hence, response time of such a call should not be strongly increased as compared to the non-redundant case.

7. Conclusions and Outlook

The data model plays a key role in any DBMS application. This fact is particularly true, if complex objects, e.g. in engineering applications, have to be processed. Although once faithful believers in the relational model, we now feel that this simple model is too simple, at least as far as advanced applications are concerned. For this reason, we were converted more and more to another 'kind of religion' and advocate the MAD model. This data model allows for dynamically defined and recursive complex objects with non-disjoint components which are derived from atom networks incorporating symmetrical relationship representation. Recursion definition enables the solution of transitive closure as well as path problems which may be directly specified by means of MQL queries.

For the implementation, we have outlined a number of important concepts. We aimed to increase the freedom in defining segments with page sizes tailored to the atom type to be stored as well as in page sequences in order to reduce the I/O problem. The mapping of atoms to records was achieved by a flexible storage scheme which guarantees fast access to any field position and limits the overhead/displacement of updates directed to variable length fields. Atom clusters were introduced to speed-up materialization of molecules which are frequently referenced. Since atoms are clustered in physical contiguity according to the molecules to be derived, 'non-disjoint atoms' have to be stored redundantly. Generally, replication was proposed as a concept to accelerate dynamical object derivation.

The materialization of complex objects is achieved by compiling MQL queries into operator graphs which are optimized according to the available access paths and storage structures. Execution of such operator graphs is then performed by an interpreter which uses a hash-based main memory data structure to represent the result molecule set.

A nested transaction concept was designed to employ parallel algorithms when processing MQL queries. By the decomposition of the dynamical execution trees of such queries into hierarchically nested subtransactions, medium or even fine grained parallelism may be achieved. This parallelism within complex object processing can be applied to retrieval as well as update operations of molecules in order to speed-up molecule materialization or the maintenance of replicated storage structures.

The stepwise abstraction process realized by the various layers of our system leads to structure-oriented complex objects; that is, the result of an MQL query is a set of molecules which, in turn, consist of a set of interconnected heterogeneous atoms without any specific application semantics. We expect that advanced DBMS applications will primarily take place in workstation/server environments where the application-oriented processing is carried out in the workstations. For this reason, our molecules are transferred across a set-oriented interface to a so-called application layer located at the workstation where they are stored in an object buffer to obtain 'nearby the application' locality. The application, e.g. a solid modeler, can then manipulate these data structures via a predefined interface which offers operations tailored to the resp. application (e.g. in the form of ADTs). Hence, the application layer may be equipped with concepts and mechanisms to overlay application-oriented semantics [BM86] to the more structure-oriented molecules of the MAD model. This approach avoids the overloading of the complex object data model by application specific aspects.

Based on this architectural framework, we have implemented a technical modeler (TechMo), a VLSI design application, and a knowledge base management system KRISYS [DHMM89, Ma89]. The former two application refer to tailored ADT interfaces which are realized by specific application layers. KRISYS, in turn, can be perceived as a (generic) modeling system which allows the specification of rich facilities and abstraction concepts for application support [Ma88]. Quite a number of applications in the areas of expert systems and intelligent CAD are currently being investigated.

Acknowledgements

We would like to thank H. Schöning for his careful reading and useful hints concerning the paper.

8. References

As76 Astrahan, M.M., et al.: SYSTEM R: A Relational Approach to Database Management, in: ACM TODS, Vol. 1, No. 2, 1976, pp. 97-137.

BB84 Batory, D.S., Buchmann, A.P.: Molecular Objects, Abstract Data Types and Data Models; A Framework, in: Proc. 10th VLDB Conf., Singapore, 1984, pp. 172-184.

BKK85 Bancilhon, F., Kim, W., Korth, H.F.: A Model of CAD Transactions, in: Proc. 11th Int. Conf. on VLDB, Stockholm, Aug. 1985, pp. 25-33.

BM86 Brodie, M.L., Mylopoulos, J. (eds.): On Knowledge Base Management Systems (Integrating Artificial Intelligence and Database Technologies), Topics in Information Systems, Springer-Verlag, New York, 1986.

CD86 Carey, M.J., DeWitt, D.J., et al.: The Architecture of the EXODUS Extensible DBMS, in: Proc. int. Workshop on Object-Oriented Database Systems, Pacific Grove, 1986, pp. 52-65.

Ch76 Chen, P.P.: The Entity-Relationship-Model - Toward a Unified View of Data, in: Proc. ACM TODS, Vol. 1, No. 1, 1976, pp. 9-36.

Da86 Dadam, P., et al.: A DBMS Prototype to Support Extended NF^2-Relations: An Integrated View on Flat Tables and Hierarchies, in: Proc. ACM SIGMOD Conf., Washington, D.C., 1986, pp. 356-367.

DD86 Dittrich, K.R., Dayal, U. (eds.): Proc. Int. Workshop on Object-Oriented Database Systems, Pacific Grove, 1986.

DE84 Special Issue on Engineering Design Databases, IEEE Database Engineering, Vol. 7, No. 2, June 1984.

DHMM89 Deßloch, S., Härder, T., Mattos, N., Mitschang, B.: KRISYS: KBMS Support for Better CAD Systems, in: Proc. 2nd International Conference on Data and Knowledge Systems for Manufacturing and Engineering, Gaithersburg - Maryland, Oct. 1989, pp.172-182.

DLPS85 Dadam, P., Lum, V.Y., Prädel, U.: Schlageter, G.: Selective Deferred Index Maintenance and Concurrency Control in Integrated Information Systems, in: Proc. 11th VLDB Conf., Stockholm, 1985, pp. 142-150.

Du87 Duppel, N., Peinl, P., Reuter, A., Schiele, G., Zeller, H.: Progress Report #2 of PROSPECT, Research Report, University of Stuttgart, 1987

Es76 Eswaran, K.P.: Aspects of a Trigger Subsystem in an Integrated Database System, in: 2nd Int. Conf. on Software Engineering, 1976, pp. 243-250.

Gr81 Gray, J.N.: The Transaction Concept: Virtues and Limitations, Proc. 7th Int. Conf. on VLDB, Cannes, Nov. 1981, pp. 144-154.

Hä88 Härder, T. (ed.): The PRIMA Project Design and Implementation of a Non-Standard Database System, SFB 124 Research Report No. 26/88, University of Kaiserslautern, 1988.

Hä89 Härder, T.: Engineering Applications - a Challenge for the Next Generation of DBMS, Internal Report ZRI 3/89, University of Kaiserslautern, 1989.

He89 Herrmann, U., et al.: A Lock Technique for Disjoint and Non-Disjoint Complex Objects, Fern-Universität Hagen, Informatik Berichte, Nr. 85, 03.1989, 15 p.

HKS90 Hübel, Ch., Käfer, W., Sutter, B.: A Client/Server System as a Base Component for a Cooperating DBMS (in German), SFB 124, Research Report 26/90, University of Kaiserslautern, May 1990.

HM90 Härder, T., Mattos, N.M.: An Enhanced DBMS Architecture Supporting Intelligent CAD, in: Proc. Int. Conf. TECHNO-DATA'90, Berlin, Dec. 1990 (invited lecture).

HMMS87 Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, SFB 124 Research Report No. 22/87, University of Kaiserslautern, 1987; in: Proc. 13th VLDB Conf., Brighton, UK, 1987, pp. 433-442.

HMS90 Härder, T., Mitschang, B., Schöning, H.: Query Processing for Complex Objects, submitted for publication, 1990.

HPS90 Härder, T., Profit, M., Schöning, H.: Supporting Parallelism in Engineering Databases by Nested Transactions, submitted for publication, 1990.

HR83 Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery, in: ACM Computing Surveys, Vol. 15, No. 4, 1983, pp. 287-317.

HR87 Härder, T., Rothermel, K.: Concepts for Transaction Recovery in Nested Transactions, in: Proc. ACM SIGMOD'87 Conf., San Francisco, May 1987, S. 239-248.

HSS88 Härder, T., Schöning, H., Sikeler, A.: Parallelism in Processing Queries on Complex Objects, in: Jajodia, S., Kim, W., Silberschatz, A. (eds.), Proc. Int. Symp. on Databases in Parallel and Distributed Computing, Austin, Texas (1988) 131-143.

HSS89 Härder, T., Schöning, H., Sikeler, A.: Parallel Query Evaluation: A New Approach to Complex Object Processing, in: IEEE Data Engineering, Vol. 12, No. 1, March 1989, pp. 23-29.

JK84 Jarke, M., Koch J.: Query Optimization in Database Systems, in: Computing Surveys 16 (1984) 111-152.

KLMP84 Kim, W., Lorie, R., McNabb, D., Plouffe, W.: Nested Transactions for Engineering Design Databases, in: Proc. 10th VLDB Conf., Singapore, 1984, pp. 355-362.

LK84 Lorie, R., Kim, W., et al.: Supporting Complex Objects in a Relational System for Engineering Databases, IBM Research Laboratory, San Jose, CA, 1984.

LL89 Lehman, T.J., Lindsay, B.G.: The Starburst Long Field Manager, in: Proc. 15th VLDB Conf., Amsterdam, Aug. 1989, pp. 375-384.

LW79 Lorie, R.A., Wade, B.W.: The compilation of a high level data language, IBM Research Report RJ 2589, San Jose, Calif. 1979.

Ma88 Mattos, N.M.: Abstraction Concepts: the Basis for Data and Knowledge Modeling, in: 7th Int. Conf. on Entity-Relationship Approach, Rom, Italy, Nov. 1988, pp. 331-350.

Ma89 Mattos, N.M.: An Approach to Knowledge Base Management - requirements, knowledge representation and design issues, Doctoral Thesis, University of Kaiserslautern, Computer Science Department, Kaiserslautern, 1989.

Mi88 Mitschang, B.: A Molecule-Atom Data Model for Non-Standard Applications - Requirements, Data model Design, and Implemlentation Concepts (in German), Doctoral Thesis, University of Kaiserslautern, Computer Science Department, Kaiserslautern, 1988.

Mi89 Mitschang, B.: Extending the Relational Algebra to Capture Complex Objects, in: Proc. of the 15th VLDB Conf., Amsterdam, 1989, pp. 297-306.

Mo81 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.

Sch89 Schöning, H.: Integrating Complex Objects and Recursion, in: Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, Japan, Dec. 1989.

Sch90 Schöning, H.: Preserving Consistency in Nested Transactions, in: Proc. HICSS-23, Volume II, Hawaii, Jan. 1990, pp. 472-480.

Si88 Sikeler, A.: VAR-PAGE-LRU: A Buffer Replacement Algorithm Supporting Different Page Sizes, in: Proc. Int. Conf on Extending Database Technology (EDBT), Venice, Italy, 1988, Lecture Notes on Computer Science 303, pp. 336-351.

SPSW90 Schek, H.-J., Paul, H.-B., Scholl, M.H., Weikum,, G.: The DASDBS Project: Objectives, Experiences, and Future Prospects, in: IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March 1990, pp. 25-43.

SR86 Stonebraker, M., Rowe, L.A.: The Design of POSTGRES, in: Proc. ACM SIGMOD Conf., Washington, D.C., 1986, pp. 340-355.

SS86 Schek, H.-J., Scholl, M.H.: The Relational Model with Relation-Valued Attributes, in: Information Systems, Vol. 11, No. 2, 1986, pp.137-147.

SS89 Schöning, H., Sikeler, A.: Cluster Mechanisms Supporting the Dynamic Construction of Complex Objects, in: Proc. 3rd Int. Conf. on Foundations of Data Organization and Algorithms FODO'89, LNCS 367, Paris, France (1989) 31-46.

SS90 Schöning, H., Sikeler, A.: Design of Storage Schemes for Enhanced Database Management Systems, SFB 124 Research Report No 25/90, University of Kaiserslautern, 1990.

We86 Weikum, G.: A Theoretical Foundation of Multi-Level Concurrency Control, in: Proc. ACM SIGACT-SIGMOD: Symposium on Principles of Database Systems, Cambridge, March 1986, pp. 31-42.

WS84 Weikum, G., Schek, H.J.: Architectural Issues of Transaction Management in Multi-Layered Systems, in: Proc. 10th VLDB Conf., Singapore, 1984, pp. 454-465.

WSSH88 Wilms, P.F., Schwarz, P.M., Schek, H.-J., Haas, L.M.: Incorporating Data Types in an Extensible Database Architecture, in: Proc. 3rd Int. Conf on Data and Knowledge Bases, Jerusalem, 1988.