Cooperative Object Buffer Management in the Advanced Information Management Prototype

K. Küspert, P. Dadam, J. Günauer

IBM Heidelberg Scientific Center
Tiergartenstr. 15
D-6900 Heidelberg, West Germany

Abstract

In the CAD, CAM, and Robotics environment the on-line construction and manipulation of data objects is very often done at dedicated workstations rather than at host systems. As the storage space of workstations is usually not that large and as large designs are also not performed by a single designer but by a group of designers, in general, one would like to use host database systems as central servers to store, to retrieve, and to "communicate" data objects. Current database management systems, however, have mainly been designed for business administration purposes where much simpler structured data objects occur. But even if the server database system offers adequate complex object support, the question remains how workstation and server database system should work together. That is, how the changes performed at the workstation should be communicated back to the server such that a new version of an object can be created at the host site in an efficient and storage saving way. In this paper the approach implemented in the Advanced Information Management Prototype (AIM-P) at the IBM Heidelberg Scientific Center is described. The AIM-P database management system is based on NF^2 (*N*on *F*irst *N*ormal *F*orm) relations and follows the strategy of a multi-level cooperation/communication between workstation and server database system in order to reduce redundant work at both sides as much as possible.

1. Introduction

Current database management systems have mainly been designed for business administration applications like accounting, banking, etc. Within these application areas, an "object" from the user's point of view, that is a part, a customer, a supplier, an employee, etc., is usually represented by just one record (tuple) in the database or - in more complex cases - by a small collection of records. Therefore, a simple tabular (relational) representation of the data is sufficient here, in general. In the engineering environment, however, that is in the areas of Computer Aided Design (CAD), Computer Aided Manufacturing (CAM), and Robotics, the data structures are usually much more complex. Though it is possible to map them into normal ("flat") relational structures, this is not very satisfying in many cases, both, from a conceptual as well as from a performance point of view. Therefore, database research has been working since several years on developing data models and underlying implementations to deal with these complex structures in a more adequate way (see e.g. /BB84, BC85, BK85, BKK85, Da86, Eb84, Fi83, HL82, HR83, Lu85, SRG83, VKC86, SW86/).

However, not only the structural complexity of the stored data differs between business administration data and engineering data, but also the way how users actually work with these data is different. In the business administration environment typical transactions access only a few records (exception: statistical queries), perform only a few operations, and last only a few seconds. In the engineering environment, due to the structural complexity of data, data accesses and data manipulation - often invisible for the user - cause a lot of database operations. Also the logical units of work, that is the manipulation or creation of data objects, are usually a rather complex piece of work, very often requiring some auxiliary software and hardware in order to display and verify the results. For both reasons (structural complexity and additional software/hardware demands), engineering users tend to perform their work at private workstations rather than at general purpose time-sharing systems (host systems), and to use the host system as a database server. (Note, that there are also trends in the business administration environment to off-load some of the work, especially for statistical evaluations, to departmental or personal computers (see e.g. /DBDZ85, GC85, Go84, RK86/).)

As data accesses of a CAD program to the host system at runtime would cause very long delays, the necessary data is usually extracted ("checked out", see /HL82/) at the beginning and brought back into the host database ("checked in") when all the work has been done. If large complex objects are manipulated at the workstation, the question arises what shall be transferred back at check-in time and what shall take place subsequently at the server site. To simply replace the old objects by the new objects is often not very adequate because in this environment users usually want to keep old *versions* of an object (see e.g. /BK85, DL85, Ka85, KL84, KSW86, Ne83/). On the other side, to generally store the new *and* the old versions always completely is also prohibitive because of storage space reasons.

An obvious solution would be to record all changes which occur to the object during manipulation at the workstation and to subsequently generate a sequence of normal "high level" data manipulation statements (e.g. in SEQUEL/SQL /Ch76, IBM2/) which are executed by the server database system at check-in time. By doing so, the server database system would be able to perform all the changes (hereby optionally creating a compact "delta version" (see /DLW84/)) of the object. On the other side, however, this would mean to duplicate at the server more or less all the work which has already been performed at the workstation. This is not only a waste of resources but also check-in times may become very long. Also the mapping of "low level" data manipulation operations produced by the workstation CAD software into reasonable "high level" data manipulation statements may become a non-trivial problem.

Under such circumstances the probably best solution is that server database system and workstation database system are de-

signed to cooperatively work together. This opens the possibility to lay out both systems in such a way that check-out requests, object transfer, and object check-in (complete object or information about changes) can be handled at the most appropriate system levels and in the most efficient way. This is the way we tried to follow with the Advanced Information Management Prototype (AIM-P /Lu85, Da86/).

The key idea can be outlined as follows: The workstation operates on a so-called *object buffer* which is assumed to reside completely in virtual (or real) memory, in general. This object buffer stores a [complex] object in a form which allows fast traversal as well as fast access to subobjects (parts). Workstation users will usually work with the AIM-P on-line interface /ALPS86/ to browse through object libraries but will use the Application Program Interface (API /EW86, EW87/) when really working with an object. Objects to be checked out are specified using the same "high level" query statements as provided by the on-line interface. After check-out the API offers hierarchically oriented cursors to "navigate" on the complex object in the object buffer as well as to communicate changes to the object buffer.

The object buffer is implemented in such a way that redundant changes (e.g. if certain pieces of data have been modified more than once since check-out) are detected and removed "automatically" during update processing. As a consequence, only the 'net' changes have to be communicated back to the server at check-in time. In addition, changes are reflected at the level of AIM-P's basic access and storage units - the so-called *subtuples* (see Section 2). Hence, check-in processing at the server can avoid the overhead of normal ("high level") query processing and - more than that - can selectively touch only those parts of a complex object which are actually affected by the change. Though the solution described in this paper uses AIM-P and its implementation as a reference basis, the general idea is certainly applicable to a large variety of systems.

The remainder of this paper is organized as follows: In *Section 2* an AIM-P system overview is given which covers the AIM-P data model, query language, storage structures, and system architecture. The techniques for AIM-P object buffer management (incl. object buffer layout, check-out processing, and check-in processing) are described in detail in *Section 3*. In *Section 4* the AIM-P Application Program Interface (API) is shortly discussed to give an impression how the programmer actually works with AIM-P at a workstation. *Section 5* provides some final remarks on the main issues of this paper and on some future directions of research in the AIM-P project.

2. System Overview

In the following we will briefly introduce the AIM-P data model, query language, storage structures, and system architecture, as far as needed to understand the subsequent discussion. More detailed descriptions of these issues can be found in /Lu85, DGW85, Da86, PT86, PA86, ALPS86/.

The AIM-P system is a DBMS prototype implementation to support NF^2 relations (see e.g. /JS82, PT83, RKS84, AB84, Sch85/), also known as 'relations with relation valued attributes' (see e.g. /Jae85a, Jae85b, SS86/), as 'unnormalized relations' (see e.g. /KTT83, VanG85/), or as 'nested relations' (see e.g. /OY85/). An NF^2 relation (also called NF^2 *table*) may have atomic and non-atomic attributes. Atomic attributes are e.g. of type boolean, integer, real, or string, whereas non-atomic attributes are relation-valued /Jae85a, Jae85b/ again. In the latter case we also use the term [NF^2] subrelation or [NF^2] *subtable*. A table in first normal form (1NF) is just a special case of an NF^2 table (without non-atomic attributes). Hence, in the NF^2 data model tables as well as subtables may be non-flat (i.e. hierarchically structured) or flat.

Fig. 1 shows an example of an NF^2 table which is non-flat. The PROGRAMS table has - at the top level - two atomic attributes PROGNAME (program name) and MAINPROG (main program) as well as two non-atomic attributes MODULES and MACLIBS (macro libraries).

MODULES is a non-flat subtable with an atomic attribute MODNAME (module name) and a non-atomic attribute PROCEDURES. PROCEDURES is a flat subtable with two atomic attributes PROCNAME (procedure name) and SIZE (size of the procedure in kilo bytes).

MACLIBS is also a non-flat subtable with an atomic attribute LIBNAME (library name) and a non-atomic attribute MACROS. MACROS is a flat subtable with two atomic attributes MACNAME (macro name) and MACTYPE (macro type, either TYPEDECL (type declaration) or PROCDECL (procedure declaration)).

In IMS-like notation /IBM1/ the PROGRAMS table would look like as shown in **Fig. 2**.

A few more remarks on Fig. 1 and on the terminology which will be used in the following: The PROGRAMS table in Fig. 1 contains two *complex objects* /HL82/: Programs AIMPV01 and AIMPV02. Program AIMPV01, for example, contains three *complex subobjects* in subtable MODULES (modules PARSL1, TIML1, and COROU2) and also three complex subobjects in subtable MACLIBS (libraries GENTY1, PROCD1, and SPECTY1). Module PARSL1 contains two *flat subobjects* (procedures PARSER and SCANNER), module TIML1 contains three flat subobjects (procedures GETTIME, CONVTIME, FROMTIME, and TOTIME), etc. For clarity it is always helpful to distinguish carefully between tables/subtables on the one side and objects/subobjects on the other side.

Fig. 3 gives an example of a query statement which performs a *projection* on the PROGRAMS table and also contains some *restrictions* (via predicates in the WHERE clauses). This query has been formulated in AIM-P's *H*eidelberg *D*ata *B*ase *L*anguage (*HDBL* /ALPS86/) which is an extension of SQL to cope also with hierarchical structures. The query retrieves those programs (attributes PROGNAME and MAINPROG) where the program name contains the substring 'V01'. The respective modules (attribute MODNAME) are also retrieved if their name contains the substring '1'. Finally, the query selects only those procedures (attributes PROCNAME and SIZE) of the selected programs and modules with a size of at least 25 kilo bytes. The reader should note that the result table of this query (**Fig. 4**) has - except the projection - the same structure as the PROGRAMS table shown in Fig. 1. A query result of this type will be called an *updatable query result* (see Section 3.2.1). Far more about the query and also about the data manipulation facilities of AIM-P can be found in some *HDBL* related papers /PT86, PA86, ALPS86/.

Some remarks on the storage structures of AIM-P: AIM-P uses a so-called *Mini Directory* concept which separates the structural information of a complex object from its data (a discussion of this approach and on other alternatives for NF^2 storage structures can be found in /DGW85/ and /Da86/). For illustration, **Fig. 5** shows the storage structure of the complex object 'program AIMPV01' and some of its subobjects from the PROGRAMS table of Fig. 1. In the AIM-P approach, a complex object consists of Mini Directory (MD) subtuples (rectangles in Fig. 5) which represent the complex object's structure and data subtuples (ovals in Fig. 5) which contain the (atomic) data fields. As one can see in Fig. 5, there is one data subtuple per object/subobject and one MD subtuple - as a kind of 'directory' - per subtable. One additional MD subtuple (called *root* MD subtuple) represents the root of the complex object and is also the database management system's entry point into that complex object. The subtuples inside a complex object are linked together in tree structure via pointers (tuple identifiers, TIDs): *D* (data) pointers are addresses of data subtuples, whereas *C* (child) pointers are addresses of MD subtuples, again. Data and MD subtuples are similar to and handled like 'normal' tuples or records in other database systems (in the RSS of System R /As81/, for instance) but are not limited to page size. All subtuples of a complex object are part of a so-called *local address space* which supports inner-object clustering /Da86/.

Because of lack of space, we cannot discuss the whole AIM-P architecture in full detail here (see /Lu85/). We will rather concentrate on those parts of the system which are relevant to show the relationship and cooperation between database server

(AIM-P main system) and the AIM-P agent running at the workstation.

Fig. 6 shows those components of AIM-P which are mainly involved in query execution and result table creation. We will first shortly explain how data extraction on the lower levels of the system is done: The database pages which contain the requested data are retrieved via the Buffer Manager (*step 1*). The Subtuple Manager (or Record Manager) is then responsible for the interpretation of the page contents; it retrieves - TID driven - those data and Mini Directory subtuples (records) of a complex object which are required for further processing on higher system levels. These subtuples are delivered to the Database (DB) Walk Manager (*step 2*). This component provides a so-called *walk interface* for the Query Processor (*step 3*). Since these database walk functions (together with the Result Walk Manager) are the essential tool for reading data from the database and writing these data into the result table (*step 4*), we will describe this interface in some detail in Section 3.2.2. The other steps shown in Fig. 6 (steps 5 to 8) will also be explained later on.

3. Result Table and Object Buffer Management

As already described in Sect. 2, the AIM-P main system at the database server uses MD subtuples as a special access path within a complex object (NF^2 tuple) to provide fast access with only few disk accesses to every subobject of a complex object. Moreover, MD subtuples and data subtuples are also kept separate such that scanning the MD structure of an NF^2 tuple can be done without touching the data subtuples and (in most cases) the pages they are residing on. The rationale behind this approach is to efficiently support projection and selection operations by reading only those parts of an NF^2 tuple which are involved in the predicate evaluation (if any) or show up in the result table.

When having specified a query, however, the result table contains nothing but relevant attributes and subobjects. Hence, a more compact representation scheme is adequate for storing the data and structural information of a complex result object - the AIM-P object buffer format. The AIM-P object buffer in its current implementation always contains *one [complex] result object* with an arbitrary number of [complex] subobjects at a time (but could be easily extended to contain several [complex] result objects at a time, if needed). The object buffer can therefore be seen as a kind of 'window' over the result table as shown in **Fig. 7**. Usually the result table of a query contains more than one [complex] result object. Each of these result objects is created at the server (in the object buffer) and then - as soon as it is complete - written into the result table on external storage (temporary segment). The result table as a whole is finally sent to the workstation where the object buffer is used - again - to accommodate a complex result object as long as it is internally processed (see Sect. 3.3).

The following four *demands* mainly guided our design and implementation of the AIM-P object buffer:

1. *Fast access:* Fast access to any part (subtable/subobject) of a complex object in the object buffer shall be adequately supported.
2. *Site autonomy:* Any kind of processing in the object buffer at a workstation shall be possible in an *autonomous* way. Especially, the workstation DBMS shall be able to create and insert new objects/subobjects locally, i.e. without having to ask the server DBMS for empty space, free addresses (TIDs), etc. All that work shall be postponed until check-in processing is done, thus avoiding unnecessary interactions between server and workstation.
3. *Object buffer = transfer unit:* It considerably simplifies and speeds up the check-out / check-in process at the workstation if a *common data structure* for complex object transfer between server and workstation on the one side *and* for complex object processing at the workstation on the other side is used. We therefore tried to find an object buffer layout which is suitable for both, object transfer and object processing.
4. *Efficient check-in:* The information kept in the object buffer should also directly *support efficient check-in* techniques at the server. It cannot be tolerated, for instance, that the database server has to scan the object buffer completely just to find out where changes have been performed at the workstation. That is, a mechanism must be provided by the object buffer management to locate changes in a complex object easily and to materialize these changes in the database efficiently. Especially, redundant work at the server should be avoided as far as possible to achieve also a substantial reduction of the server's workload by manipulating complex objects at a workstation.

Data which shall be processed at a workstation is extracted from the AIM-P database and thereby transformed into the object buffer format. In Section 3.1 we explain how this internal format of complex (result) objects looks like. Then, in Section 3.2, it will be shown how data extraction and format transformation are actually done. In Section 3.3 the mechanisms for data manipulation in the object buffer at the workstation are described. Finally, in Sections 3.4 and 3.5, propagation of changed data from the workstation back to the server and materialization of these changes in the server database are discussed.

3.1 Object Buffer Layout

The AIM-P object buffer consists of two major areas: A *description area* and a *data area*. Both are linear, consecutive storage spaces residing in virtual memory. **Fig. 8** shows the contents of the object buffer after having loaded the complex result object 'program AIMPV01' and its subobjects as shown in the result table of Fig. 4. We will use Fig. 8 in the following to explain the AIM-P object buffer concept in more detail.

The contents of the *data area* are the data of the complex object as selected via the predicates and projections of the query statement (Fig. 3 in our example). The data area does *not* contain any auxiliary information, such as length descriptions for fields, etc.

The *description area* contains all structure information which are needed for the interpretation of the data area as well as for the structural representation of the complex object. The description area consists of two parts: a main part and an instance part.

The *main part* contains some global information like the data area pointer, which points to the beginning (start address) of the data area, the free space offset, which tells where the unused (free) space area begins within the data area, and the no. of subobjects entry, which tells the number of *sub*objects within the complex object (this number is 5 in Fig. 8 - two modules plus three procedures).

The *instance part* consists of no. of subobjects + 1 entries (called 'rows' R_i), containing all the structure and hierarchical relationship ('child') information for every subobject of the complex object which is currently residing in the object buffer. To make the description area relocatable as well as to keep address calculation within the instance part simple, all instance rows have the same length. The first row (R_1) describes the object, the other rows (2 ... no. of subobjects + 1) describe the subobjects, i.e. there is also one row R_i per subobject ('instance').

Fig. 9 shows - from a logical as well as from a physical point of view - how these instance rows actually represent a complex object's structure.

Fig. 9a illustrates how the rows R_i are used to describe a tree structure. Node R_1 represents the root of the complex object (program AIMPV01 in the result table of Fig. 4), node R_2 and node R_5 represent the two modules PARSL1 and TIML1, which are children (subobjects) of program AIMPV01. R_3 and R_4 stand for the two procedures PARSER and SCANNER in module PARSL1. Node R_6 stands for the procedure CONVTIME in module TIML1. Conceptually, child and brother pointers are used as 'links' for a complex object's structural representation in the object buffer.

Fig. 9b shows how these 'links' are implemented via first child (FC), left brother (LB), and right brother (RB) 'pointers' which are actually row numbers. The child information (FC) is only set in rows R_1, R_2, and R_5 since only these rows - respectively the corresponding nodes in the tree representation - have other subobjects as children, again. The left and right brother information (LB, RB) are only set in rows R_2 and R_5 (module level) as well

as in rows R_3 and R_4 (procedure level). In all other cases LB, RB, and FC are 'null'.

Besides the LB, RB, and FC information the instance rows also contain data (DA) offsets relatively to the beginning of the data area. One data offset exists in the instance part for each (atomic) attribute value which is stored in the data area. Note, that the data area pointer is the only real *pointer* (as a virtual memory address) in the object buffer whereas all other kinds of addressing (FC, LB, RB, and DA) are done via row numbers and *offsets*. This saves much processing time when a complex object is moved to another place, especially from the database server to the workstation and vice versa, since the offsets are stable and need not be recomputed ('relocatable object').

Apart from what has been explained so far, the instance rows contain some more information (especially in the *header* fields) to keep track of changes which have been performed on the complex object at the workstation since check-out. We will come back to that issue in more detail in Sections 3.2.1 and 3.3.

3.2 Filling the Object Buffer - Check-out

As already mentioned above, the AIM-P object buffer is created and filled with data at the server side. First, an *HDBL query* is sent from the workstation to the server where query execution takes place. The server DBMS extracts the requested *data* from the database and transforms these data into the internal object buffer format. The reader should note that a complex object as it is stored in the database (Fig. 5) cannot - or should not, at least - simply be taken 'as it is' and sent to a workstation for several reasons:

1. In many cases only certain parts of a complex object are actually needed for processing at the workstation (see also the projections and restrictions in our example query (Fig. 3)). To avoid unnecessary data retrieval and transfer operations, only the requested data should be extracted and transformed to a 'dense' format before sending them out.
2. Database addresses (tuple identifiers (TIDs), page numbers, etc.) are only valid within a certain context, e.g. a database segment. Moving data to another place (such as a workstation) can usually not be done without recomputing these addresses. Even if a more indirect addressing concept via translation tables etc. is used, the contents of these tables must at least be adjusted.
3. Because of hardware and/or software restrictions the workstation DBMS should often be smaller and less complex than the general purpose server DBMS. Therefore, sophisticated storage structures and addressing concepts as they may be used for the server DBMS are not always appropriate for the workstation DBMS.

3.2.1 Updatable and Non-Updatable Query Results

In the following, we will have to distinguish between two kinds of queries and query results:

- A *non-updatable* query result means that the workstation user has formulated a query just to *read* the query result (result table) which he got from the database server. The user does *not* want to perform any updates (which shall be propagated back to the server) on this result table. A query result may also be non-updatable 'per se', for instance if join operations or aggregations (SUM, AVG, etc.) have been performed. In these cases there is no simple correspondence between the database objects on the one side and the result objects on the other side such that updates cannot be applied and materialized unambiguously.
- An *updatable* query result permits any kind of *update* operations on the result data at the workstation, and the updates can of course also be materialized at the server later on.

In AIM-P the workstation user must state for any query explicitly which kind of query result (updatable or non-updatable) he wants to have. If the server DBMS gets a request for an updatable query result, it checks whether the following conditions are all fulfilled:

1. The query extracts its data only from a single database table.
2. No joins between subtables have been specified (i.e. no restructuring has been done).
3. No attribute values in the result table have been generated using aggregation functions (SUM, AVG, ...) or arithmetic functions.

If one of these checks fails the workstation user gets a message that the given query produces a non-updatable query result. The query in Fig. 3 complies with these rules since it contains only projections and restrictions, i.e. the result table shown in Fig. 4 is updatable.

An updatable query result differs from a non-updatable one in so far, that the database addresses (TIDs) are part of the object buffer in the first case, while they are missing in the second case. These addresses are in fact 'address pairs' which consist of the address of the respective data subtuple (DST_i in Fig. 9b) and the address of the MD subtuple it belongs to (MD_i in Fig. 9b). These addresses are contained in the header fields of the instance rows R_i which, in turn, are part of the description area of the object buffer, as already described in Section 3.1 (see also Fig. 9b). DST_1, for instance, represents the address (TID) of the data subtuple 'AIMPV01 QPTEST' shown in Fig. 5, etc.

Most of the following discussion applies to both updatable and non-updatable query results. In our examples, however, we refer to the creation of an updatable query result (Figs. 3 and 4) since that scenario contains some more interesting aspects than the non-updatable case.

3.2.2 Result Table and Object Buffer Creation at the Server

As for the AIM-P system complex objects are no "special animals" but normal NF^2 tuples, it supports - according to the NF^2 data model - not only retrieval of complex objects as a whole but also selections and projections within a complex object (see Fig. 3). As a consequence, the hierarchical structure of such an object has to be explicitly exposed at some system-internal level to enable reasonable access to its subparts. As already outlined in Section 2, NF^2 tuples are stored in a hierarchical fashion using MD and data subtuples. To shield the Query Processor component (see Fig. 6) from implementation details of these "physical" structures, a somewhat "higher" logical interface - called *database walk* - is used in AIM-P to traverse this hierarchy.

In reality, this database walk provides not just one operator for traversing a complex object, but a set of interdependent elementary walks, each of which is bound to one NF^2 table or subtable. At the instance level, the walks are "walking" over the respective NF^2 table or subtable occurrences. To process e.g. an NF^2 table having the structure as shown in Figs. 1 and 2, one walk would have to be opened on PROGRAMS, two others - "below" this walk - on MODULES respectively MACLIBS, and - finally - also on PROCEDURES ("below" the MODULES walk) and MACROS ("below" the MACLIBS walk).

Being positioned on a specific [sub]object, a walk gives access to the atomic fields defined at this level, e.g. to PROGNAME and MAINPROG at the PROGRAMS level, to MODNAME at the MODULES level, to PROCNAME and SIZE at the PROCEDURES level, etc. The current walk position also defines the scope for the dependent walks "below". If, for example, the top level walk is positioned on the first complex object (first program), hereby giving access to the atomic values (AIMPV01, QPTEST), the walk at the MODULES level can only process the modules belonging to the first program (PARSL1, TIML1, COROU2). The same holds, analogously, for the walk on PROCEDURES (PROCNAME, SIZE), whose scope is defined by the walk on MODULES (within the walk on PROGRAMS).

In total, one can also see a database walk as a *multi-level scan* operation or some kind of currency indicator as used in CODASYL-like systems /CODA78/. There is also a pretty close relationship to the concepts used for "navigation" in IMS /IBM1/.

Processing a query leads to the creation of a *result table* containing the updatable or non-updatable query result (see Section 3.2.1), which is then sent to the workstation for further processing (step 5 in Fig. 6). Opposed to ordinary NF^2 tables, as permanently or temporarily stored in the database, the NF^2 tuples (complex objects) within this result table are stored in the object buffer representation form as described in Section 3.1. As also

described there, the object buffer representation is equivalent - with respect to describing the structural relationships - to the MD and data subtuple representation in the stored database. Therefore, an analogous walk interface - called *result walk* - is used for creating and traversing the object(s) in the result table.

Transforming an NF^2 database table (in MD / data subtuple format) into a result table - hereby applying selections and projections as specified in the query expression - requires a "synchronized walking" on both, the NF^2 table and the result table, hereby, step by step, filling the latter one.

Because of lack of space, we cannot describe these walks and their implementation in more detail here. However, to understand the subsequent discussion on object buffer processing and update propagation, this outline of these functions should be sufficient. For readers who are interested in some more details, a list of the AIM-P walk functions is given in the Appendix; a comprehensive description can be found in /Kü87/.

3.3 Object Buffer Processing at the Workstation

As already mentioned above, the reason for providing an object buffer at the workstation is to achieve some level of site autonomy, efficient processing of complex objects at the workstation site, and workload reduction at the server site by off-loading some of the DBMS work to the workstation.

A necessary precondition to fulfill the first goal (site autonomy) is to provide the workstation with enough information about the object(s) in the object buffer(s). How this information currently looks like in AIM-P has been described in Section 3.1. Autonomous processing has two sides, however: check-out and check-in. To enable an efficient and "selective" (transmission and processing of changes, only) check-in mechanism at the server later on, the workstation has to *protocol* all changes which happened to its complex objects in one way or another.

The most straightforward solution would be to use traditional DBMS *after image logging*; this is not feasible here, because the object buffer representation does not correspond to the complex object representation in the NF^2 database tables at the server side. That is, these after image log records could not directly be processed at the server side. But even if this would have been possible, the amount of data to be transmitted and the resulting processing overhead would be prohibitive, in general. To use *operation logging* instead of after image logging would be possible, in principle, contradicts, however, to the goal of reducing some of the work at the server side. Without applying highly sophisticated optimization techniques to the initial sequence of (log) operations in order to eliminate redundant operations or to group operations where possible by object and subobject, the server would just repeat all the work already done at the workstation side, hereby causing long check-in times.

In AIM-P, protocolling is therefore done by *flagging* the affected parts of an object in the object buffer according to the operation(s) performed. By having different flags for insertions, updates, and deletions, and by having some kind of flag priority scheme (delete "overrules" a previously performed insert or update), the object buffer is *self-optimizing* with respect to the elimination of redundant operations.

The flagging itself can be done in two ways, namely

- one-level flagging and
- multi-level flagging.

When performing *one-level flagging*, only those instance rows (respectively subobjects) in the object buffer are flagged, which are directly affected by the change. This is sufficient from an information point of view, requires, however, a complete sequential scan of the description area at check-in time at the server (see Section 3.5). If the objects to be processed are not very big, i.e. if they do not consist of too many subobjects, this overhead is tolerable, in general. If the objects are very big, however (and in many applications they are), a more selective scheme, which avoids - or at least reduces - this overhead, would be preferable.

For that purpose, the *multi-level flagging* scheme has been developed. In this scheme, not only the directly affected instance rows (respectively subobjects) are flagged, but also all other rows lying on the same hierarchical path from the directly affected instance row up to the root of the hierarchy (see Section 3.1). The purpose of the additional "higher level" flags is to signal the *Update Manager* (see Fig. 6 and Section 3.5) whether - within a given subtree of the object hierarchy - a modification has been done at some lower level or not. To distinguish "signal flags" from real change flags, different *flag types* are used. This *upward propagation* of changes via signal flags is already done at the workstation during normal object buffer processing. The upward propagation of flags causes very little extra overhead since the workstation DBMS keeps always track of the 'parent' rows of an instance row.

3.4 Update Propagation - Check-in Preparation and Execution at the Workstation

If a query result has been changed at the workstation (via the result walk services shown in the Appendix), the Result Walk Manager must be able to propagate these changes back to the server where they can be materialized in the database. It would be far too expensive - especially if just a few small changes have been performed in some large complex objects - to send all objects from the result table back to the server. For a given result table the Result Walk Manager at the workstation must therefore be able to

- find out which complex objects have been changed,
- extract the *changed* data from these complex objects.

The Result Walk Manager can then perform some kind of 'delta propagation' in order to reduce the communication overhead between workstation and server.

To find out which complex objects have been changed, the Result Walk Manager maintains a bit list for each result table with one bit position per complex result object. A bit is 'on' if and only if the respective object has been changed. At propagation time this bit list is scanned and those complex objects which have been changed are read into the object buffer for further examination.

To support delta propagation on complex object level, the Result Walk Manager at the workstation does not mix the 'old' data (which have been sent from the server to the workstation) and the 'new' data (which are created at the workstation during updates and inserts). As a general strategy, no update in place is performed in the *data area* even if the length of an attribute value does not change. New data or changed data are always appended to the current end of the data area. The respective data offset in the instance row (DA field in Fig. 9b) is set or changed to maintain the correct address.

Fig. 10 shows the object buffer of Fig. 9 after a new module (with one procedure) has been inserted and the name of an old module (PARSL1) has been changed to PARSL2. To reflect the insertions, two new instance rows have been created at the end of the description area, one (R_7) for the new module and one (R_8) for the new procedure in that module. To connect R_7 and R_8 to the existing instance tree, RB (right brother) in R_5 - which was previously 'null' - and LB (left brother) in R_7 have been set. The name of the new module (CHECK) and the data for the new procedure (CHECKDTA, 40) have been appended to the current end of the data area (offsets O_{11}, O_{12}, O_{13}). The update of the MODNAME attribute for module PARSL1 has then been done via an insertion at offset O_{14} in the data area, and the respective data offset in instance row R_2 has been changed from O_3 (→ PARSL1) to O_{14} (→ PARSL2).

This mechanism requires, of course, some more storage space in the data area than an update-in-place strategy. The major advantage is, however, that the new data is always separated from the old data and hence needs not be extracted at propagation time. Since both kinds of data (old and new) are stored in the same data area, addressing can still be done in a simple and uniform way.

To propagate all these changes back to the server, the description area (incl. rows R_1 to R_8) and the *new* (!) contents of the data area (from offset O_{11} to offset O_{15}) are concatenated in virtual storage. The start offset of the new data (O_{11}) is stored within the main part of the description area since it is needed later on for address

calculation at the server side. The concatenated storage spaces are then sent back to the database server (*step 6* in Fig. 6).

3.5 Update Materialization at the Server

After having received the 'delta' object buffer(s) from the workstation at the server the *Update Manager* of the server DBMS is responsible for the materialization of all changes (updates, insertions, and deletions) which are reflected in these object buffer(s). The changes must be transformed into modifications of the database storage structures (Mini Directory (MD) and data subtuples, steps 7 and 8 in Fig. 6).

A complex object in the object buffer may contain an arbitrary number and 'mixture' of changes at different places and on different levels: Some of its data may have been updated at the workstation, new subobjects may have been inserted, and old subobjects may have been deleted. The Update Manager must look at the *change flags* (which have been introduced in Section 3.3) in the instance rows to find out where (and which kind of) changes have been performed.

When a change has been detected in the object buffer (via the flags), the database addresses (TIDs) in the header of the respective instance row (MD_i and DST_i in Fig. 10b) are used to determine those database subtuples which have to be modified. Depending on the change flags, the following actions have to be performed to materialize the changes in the database:

- *Update*: In case of a simple update the TID of the respective data subtuple (DST_i) is known from the instance row. The Update Manager modifies that data subtuple via an appropriate Subtuple Manager call, where the update is (physically) performed at the end.
- *Deletion*: If the change flag says that a subobject shall be deleted from a certain subtable, the MD subtuple of this subtable is accessed via its TID (MD_i in the instance row). The address entry for the subobject is determined via a search and can then be eliminated from the MD subtuple. Finally, the whole subobject can be deleted. This is all done via an appropriate series of Subtuple Manager calls ('update subtuple', 'delete subtuple(s)').
  The deletion of a complex object as a whole is just a special case of that strategy.
- *Insertion*: For an insertion of a new subobject, the MD subtuple of the subtable where the insertion shall be done must be retrieved. Its address can again be found as MD_i in the instance row. The subobject is stored via Subtuple Manager calls ('store subtuple(s)'), and a new address entry is appended to the end of the respective MD subtuple. This MD subtuple is finally updated.
  The creation of a new complex object as a whole can again be treated as a special case of that scenario. The new object is built up in the object buffer at the workstation autonomously and finally - at check-in time - sent to the server for insertion into the database table where empty space is acquired, addresses (TIDs) are assigned, etc.

4. Introduction to the Application Program Interface

As already stated earlier, an application program at a workstation does not directly interact with the Result Walk Manager. The Result Walk interface (see Appendix) is not yet the right tool for that purpose since it offers only some basic services (as procedure calls) for object buffer and result table processing. A more user friendly - and also more powerful - interface for data access and manipulation via an application program had to be provided.

In the following we just want to give an impression how the AIM-P application program interface (API) works and how it looks like. A more comprehensive discussion can be found in /EW86/ and /EW87/.

For using AIM-P from an application program at a workstation a *precompilation* approach has been taken which is an extension to what has been done for System R /LW79/ and SQL/DS /IBM2/. To define which query result shall be processed, the programmer *embeds* the appropriate *HDBL* query statement (such as the one in Fig. 3) into the source code of the application program. Such a result declaration can be done via the statement:

DECLARE RESULT result_name [FOR UPDATE]
FROM QUERY_STATEMENT 'SELECT ... '

'Result_name' becomes the program-internal name (identifier) of the result table to be obtained from the database server, and via the 'FOR UPDATE' option an updatable query result (see Section 3.2.1) can be requested. The DECLARE RESULT statement, however, does not imply the query execution and data extraction. This has to be done via another statement which is also embedded into the application program: EVALUATE result_name. These statements for result table declaration and data extraction have been separated in order to permit a repetitive execution of a query which has to be defined (and parsed, etc.) only once (see also /LW79/ and /Ch81/ for a similar concept in System R).

Before we discuss how the contents of a result table can be processed via the API, the general strategy for program precompilation and execution shall shortly be explained. The embedded statements (like DECLARE, EVALUATE, and others which will be described later on) are understood by the API *precompiler*. The precompiler transforms the source program with the embedded statements so that it can then be processed by the 'normal' compiler. At precompilation time, the embedded statements are replaced by procedure calls to the API *runtime system* (RTS) which runs on top of the Result Walk Manager (see Fig. 6). At program execution time (runtime), the application program calls the API runtime system which in turn calls the Result Walk Manager for certain operations on the result table.

Query execution and data extraction at the server are also done at runtime via a call to the API RTS which is forwarded to the Query Processor. The result table is then sent to the workstation where further processing is done via the API.

As an extension to the cursor concept for (flat) tables in System R and SQL/DS, *hierarchical cursors* can be defined in the application program to handle also non-flat result tables. Like a database or result walk (see Section 3.2.2), a hierarchical cursor consists of a set of interdependent elementary cursors. Each elementary cursor is bound to one "data subtuple level" in the hierarchy. Because of this one-to-one correspondence between cursors and walks, a cursor can directly be implemented as a walk at the Result Walk interface /Kü87/.

The cursor definition in the application program is again done via an embedded statement (DECLARE CURSOR ...) which is understood by the API precompiler. Other statements, which the API precompiler transforms into API RTS calls, are available to OPEN, to MOVE, or to CLOSE a cursor. The application program interface offers some more options for these statements than are provided by the Result Walk Manager one level "below". The programmer may move, for instance, a cursor forward or backward by a given step-width. He may also move it directly to a certain position in a table or subtable what can be used for partial and range processing of lists ('from element ... to element ...'). These more powerful cursor operations are mapped down by the API RTS to a sequence of calls at the Result Walk interface.

Embedded program statements like FETCH, UPDATE, INSERT, and DELETE can be used

- to *read* data from a result table into application program variables,
- to *change data* in objects or subobjects, i.e. to move the contents of program variables into a result table,
- to *insert* new objects or subobjects into a result table,
- to *delete* existing objects or subobjects from a result table.

These statements are always bound to a cursor. The cursor's *state* (ON an object/subobject, BEFORE an object/subobject, etc.) and *position* within the table or subtable determine which data are actually read, changed, inserted, or deleted. (UPDATE, INSERT, and DELETE statements are only allowed, of course, for updatable query results.) All these statements are in fact *set-oriented*. This simplifies programming substantially and may also speed up runtime processing since the number of RTS calls can be reduced. The programmer may, for instance, delete all objects or subobjects of a table or subtable with *one* DELETE statement, and he may also use *one* INSERT statement to insert more than one object or subobject at a time. Since an INSERT state-

ment also delivers the data, this implies that a *set* of data must be provided rather than the data for a single object or subobject. For that purpose, the programmer may fill an *array* in the application program with the new data before the insertion is done.

Besides that set-orientation, there is also - as another option - a *tuple-orientation* which can be used for further simplification of data handling and data transfer between the application program and the API runtime system /EW86/. If the programmer wants to read the data tuple-oriented, he must provide suitable *record* variables in his application program where the data can be delivered by the API runtime system. Tuple and set orientation can, of course, also be used in combination if an array of records is provided.

5. Summary and Outlook

In this paper we have described how a workstation and a server DBMS can closely work together such that in an engineering design environment (and not only there) a substantial reduction of the server's workload can be achieved. As opposed to concepts used for most distributed database management systems, the systems in our approach do not only communicate via the "high level" [relational] database interface but also via "lower" system interfaces. That is, we have proposed a logically tight cooperation between workstation and server database system using a multi-level cooperation and communication strategy. The solution described in this paper has been fully implemented and is in use as an integral part of our DBMS. Though this approach helps already quite a lot to speed up check-in processing, we would like to gain even more from the local processing which is done (or could be done) at the workstation.

In the current implementation, e.g. index updates are performed completely at the server side during check-in processing. Though this is not that bad for simple indexes like B-trees, the overhead for maintaining more complex indexes - e.g. like our text fragment index /KW81/ - is considerable. Instead of performing all this work at check-in time at the server, a lot of preparatory work (e.g. text decomposition and computation of all index terms for text index maintenance) could already be done at the workstation. Also more has to be done for consistency control, like checking whether a unique key condition has been violated during update processing at the workstation. This should also already be done ahead of normal check-in processing. We are cooperating with the University of Darmstadt (see /De86, DO86/) to develop more comprehensive solutions. -- In this cooperation we are also currently looking into the problems of *concurrency control* for workstation server DBMS's what has not been mentioned in this paper so far. The granularity of locking (complex object level, subobject level, subtuple level, ...), for instance, will of course be a decisive factor for the overall system performance in a multi-user environment.

The Advanced Information Management Prototype has now entered the phase of being experimentally used in various application areas. Especially connecting the system to CAD, CAM, and Robotics applications will be of major interest (see /DDKL86, KLW86, Kl85, KSW86/). We plan to report about our experiences in a forthcoming paper.

Acknowledgements

We wish to thank our colleagues of the Advanced Information Management Prototype project, especially R. Erbe, U. Herrmann, P. Pistor, and N. Südkamp, for their helpful comments on an earlier version of this paper.

References

AB84 S. Abiteboul, N. Bidoit: Non First Normal Form for Relations to Represent Hierarchically Organized Data. Proc. ACM PODS, 1984, pp. 191-200

ALPS86 F. Andersen, V. Linnemann, P. Pistor, N. Südkamp: Advanced Information Management Prototype - User Manual of the On-line Interface of the Heidelberg Data Base Language (*HDBL*) Prototype Implementation (Release 1.1). Technical Note TN 86.01, IBM Heidelberg Scientific Center, Nov. 1986

As81 M.M. Astrahan et al.: A History and Evaluation of System R. Communic. of the ACM, Vol. 24, No. 10, Oct. 1981, pp. 632-646

BB84 D.S. Batory, A.P. Buchmann: Molecular Objects, Abstract Data Types, and Data Models: A Framework. Proc. VLDB 84, Singapore, Aug. 1984, pp. 172-184

BC85 A.P. Buchmann, C.P. de Celis: An Architecture and Data Model for CAD Databases. Proc. VLDB 85, Stockholm, Sept. 1985, pp. 105-114

BK85 D.S. Batory, W. Kim: Modelling Concepts for VLSI CAD Objects. ACM TODS, Vol. 10, No. 3, Sept. 1985, pp. 322-346

BKK85 F. Bancilhon, W. Kim, H.F. Korth: A Model for CAD Transactions. Proc. VLDB 85, Stockholm, Sept. 1985, pp. 25-33

Ch76 D.D. Chamberlin et al.: SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control. IBM Journal of Research and Development, Vol. 20, No. 6, Nov. 1976, pp. 560-575

Ch81 D.D. Chamberlin et al.: Support for Repetitive Transactions and Ad-Hoc Queries in System R. ACM TODS, Vol. 6, No. 1, March 1981, pp. 70-94

CODA78 Report of the CODASYL Data Description Language Committee. Information Systems, Vol. 3, No. 4, 1978, pp. 247-320

Da86 P. Dadam, K. Küspert, F. Andersen, H. Blanken, R. Erbe, J. Günauer, V. Lum, P. Pistor, G. Walch: A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies. Proc. ACM SIGMOD 86, Washington, D.C., May 1986, pp. 356-367

DBDZ85 A. Diener, R.P. Brägger, A. Dudler, C.A. Zehnder: Replicating and Allocating Data in a Distributed Database System for Workstations. Proc. ACM SIGSMALL Symposium on Small Systems, Danvers, Mass., May 1985, pp. 5-9

DDKL86 P. Dadam, R. Dillmann, A. Kemper, P.C. Lockemann: Objektorientierte Datenhaltung für die Roboterprogrammierung (Object-Oriented Data Management for Robot Programming). University of Karlsruhe, Dept. of Computer Science, Technical Report 18/86, Nov. 1986 (in German)

De86 U. Deppisch, J. Günauer, K. Küspert, V. Obermeit, G. Walch: Überlegungen zur Datenbank-Kooperation zwischen Server und Workstations (Thoughts on Database Cooperation between Server and Workstations). Proc. GI-Jahrestagung 86, Berlin, Oct. 1986, Informatik-Fachberichte 126, Springer-Verlag, pp. 565-580 (in German)

DGW85 U. Deppisch, J. Günauer, G. Walch: Speicherungsstrukturen und Adressierungstechniken für komplexe Objekte des NF^2-Relationenmodells (Storage Structures and Addressing Techniques for Complex Objects of the NF^2 Relational Model). Proc. GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft", Karlsruhe, March 1985, Informatik-Fachberichte 94, Springer-Verlag, pp. 441-459 (in German)

DL85 K.R. Dittrich, R.A. Lorie: Version Support for Engineering Data Base Systems. IBM Research Report RJ4769, San Jose, Cal., July 1985

DLW84 P. Dadam, V. Lum, H.-D. Werner: Integration of Time Versions into a Relational Database System. Proc. VLDB 84, Singapore, Aug. 1984, pp. 509-522

DO86 U. Deppisch, V. Obermeit: Tight Database Cooperation in a Server-Workstation Environment. Univ. of Darmstadt, Dept. of Computer Science, 1986 (accepted for publication)

Eb84 W. Eberlein: Architektur technischer Datenbanken für integrierte Ingenieursysteme (Architecture of Technical Databases for Integrated Engineering Systems). Ph.D. Dissertation, University of Erlangen-Nürnberg, 1984 (in German)

EW86 R. Erbe, G. Walch: Usage of the Application Program Interface of the Advanced Information Management Prototype. Technical Note TN 86.03, IBM Heidelberg Scientific Center, Dec. 1986

EW87 R. Erbe, G. Walch: An Application Program Interface for an NF^2 Data Base Language or How to Transfer Complex Object Data into an Application Program. Technical Report TR 87.04.003, IBM Heidelberg Scientific Center, April 1987

Fi83 W.E. Fischer: Datenbanksysteme für CAD-Arbeitsplätze (Database Systems for CAD Workstations). Informatik-Fachberichte 70, Springer-Verlag, 1983

FT83 P.C. Fischer, S.J. Thomas: Operations on Non-First-Normal-Form Relations. Proc. IEEE Computer Software and Applications Conf., Oct. 1983, pp. 464-475

GC85 D. Gantenbein, A. Cockburn: Architecture and Usage of a Host-Coupled Workstation. Research Report RZ1382, IBM Zürich Research Lab., Rüschlikon, Schweiz, June 1985

Go84 B.C. Goldstein, A.R. Heller, F.H. Moss, I. Wladawsky-Berger: Directions in Cooperative Processing Between Workstations and Hosts. IBM Systems Journal, Vol. 23, 1984, pp. 236-244

HL82 R.L. Haskin, R.A. Lorie: On Extending the Functions of a Relational Database System. Proc. ACM SIGMOD 82, Orlando, Florida, June 1982, pp. 207-212

HR83 Th. Härder, A. Reuter: Database Systems for Non-Standard Applications. Proc. ICS 83, Nürnberg, pp. 452-466

IBM1 IBM Systems Journal (Special Issue on IMS), Vol. 16, No. 2, 1977

IBM2 SQL/Data System, Concepts and Facilities. IBM Corporation, GH24-5013

Jae85a G. Jaeschke: Nonrecursive Algebra for Relations with Relation Valued Attributes. Technical Report TR 85.03.001, IBM Heidelberg Scientific Center, March 1985

Jae85b G. Jaeschke: Recursive Algebra for Relations with Relation Valued Attributes. Technical Report TR 85.03.002, IBM Heidelberg Scientific Center, March 1985

JS82 G. Jaeschke, H.-J. Schek: Remarks on the Algebra of Non First Normal Form Relations. Proc. ACM SIGACT-SIGMOD Symp. on Principles of Data Base Systems, Los Angeles, Cal., March 1982, pp. 124-138

Ka85 R. Katz: Information Management for Engineering Design. Springer-Verlag, 1985

KL84 R.H. Katz, T.J. Lehman: Database Support for Versions and Alternatives of Large Design Files. IEEE Transactions on Software Engineering, Vol. SE-10, No.2, March 1984

Kl85 P. Klahold et al.: A Transaction Model Supporting Complex Applications in Integrated Information Systems. Proc. ACM SIGMOD 85, Austin, Texas, May 1985, pp. 388-401

KLW86 A. Kemper, P.C. Lockemann, M. Wallrath: An Object-Oriented Database System for Engineering Applications. To appear in Proc. ACM SIGMOD 87, San Francisco, Cal., May 1987

KSW86 P. Klahold, G. Schlageter, W. Wilkes: A General Model for Version Management in Databases. Proc. VLDB 86, Kyoto, Japan, Aug. 1986, pp. 319-327

KTT83 Y. Kambayashi, K. Tanaka, K. Takeda: Synthesis of Unnormalized Relations Incorporating More Meaning. Information Sciences, 1983

Kü87 K. Küspert: Advanced Information Management Prototype - Result Walk: External Interface Description. Technical Note TN 87.01, IBM Heidelberg Scientific Center, Feb. 1987

KW81 D. Kropp, G. Walch: A Graph Structured Text Field Index Based on Word Fragments. Information Processing and Management, Vol. 17, No. 6, 1981, pp. 363-376

Lu85 V. Lum et al.: Design of an Integrated DBMS to Support Advanced Applications. Proc. Int. Conf. on Foundations of Data Organization (Invited Talk), Kyoto, Japan, May 1985, pp. 21-31 (also published in Informatik-Fachberichte 94, Springer-Verlag, 1985, pp. 362-381)

LW79 R.A. Lorie, B.W. Wade: The Compilation of a High Level Data Language. Research Report RJ2598, IBM San Jose Research Lab., 1979

Ne83 T. Neumann: On Representing the Design Information in a Common Database. Proc. ACM SIGMOD 83, San Jose, Cal., May 1983

OY85 Z.M. Ozsoyoglu, L.Y. Yuan: A Normal Form for Nested Relations. Proc. ACM PODS, March 1985, pp. 251-260

PA86 P. Pistor, F. Andersen: Designing a Generalized NF^2 Model with an SQL-type Language Interface. Proc. VLDB 86, Kyoto, Japan, Aug. 1986, pp. 278-285

PT86 P. Pistor, R. Traunmüller: A Data Base Language for Sets, Lists, and Tables. Information Systems, Vol. 11, No. 4, 1986, pp. 323-336 (also available as Technical Report TR 85.10.004, IBM Heidelberg Scientific Center, 1985)

RK86 N. Roussopoulos, H. Kang: Preliminary Design of ADMS±: A Workstation-Mainframe Integrated Architecture for Database Management Systems. Proc. VLDB 86, Kyoto, Japan, Aug. 1986, pp. 355-364

RKS84 M.A. Roth, H.F. Korth, A. Silberschatz: A Theory of Non-First-Normal-Form Relational Databases. Technical Report TR-84-36, Univ. of Texas at Austin, Dept. of Computer Science, Dec. 1984

Sch85 H.-J. Schek: Towards a Basic Relational NF^2 Algebra Processor. Proc. Int. Conf. on Foundations of Data Organization, Kyoto, Japan, May 1985, pp. 173-182

SRG83 M. Stonebraker, B. Rubenstein, A. Guttman: Application of Abstract Data Types and Abstract Indices to CAD Data Proc. Database Week - Engineering Applications Stream, Database Week 83, San Jose, Cal., May 1983

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

SW86 H.-J. Schek, G. Weikum: DASDBS: Concepts and Architecture of a Database System for Advanced Applications. Technical Report DVSI-1986-T1, University of Darmstadt, Dept. of Computer Science, 1986

VanG85 D. van Gucht: Theory of Unnormalized Relational Structures. Ph.D. Dissertation, Vanderbilt University, 1985

VKC86 P. Valduriez, S. Khoshafian, G. Copeland: Implementation Techniques for Complex Objects. Proc. VLDB 86, Kyoto, Japan, Aug. 1986, pp. 101-110

Tables and Figures

{ PROGRAMS }							
PROGNAME	MAINPROG	{ MODULES }			{ MACLIBS }		
		MODNAME	{ PROCEDURES }		LIBNAME	{ MACROS }	
			PROCNAME	SIZE		MACNAME	MACTYPE
AIMPV01	QPTEST	PARSLI	PARSER	120	GENTY1	COMMON	TYPEDECL
			SCANNER	80		MIXTYPE	TYPEDECL
		TIMLI	GETTIME	20		DATE	TYPEDECL
			CONVTIME	30		STRTY	TYPEDECL
			FROMTIME	15	PROCD1	DATEPROC	PROCDECL
			TOTIME	15		PTPROC	PROCDECL
		COROU2	MOVEOBJ	20		TTPROC	PROCDECL
						BUFFTY	TYPEDECL
						UBUFFTY	TYPEDECL
					SPECTY1	INTTY	TYPEDECL
AIMPV02	AIMPDB	OBJIIDL1	GETOBJ	12	GENTY2	COMMON	TYPEDECL

Fig. 1: Example of an NF^2 Table

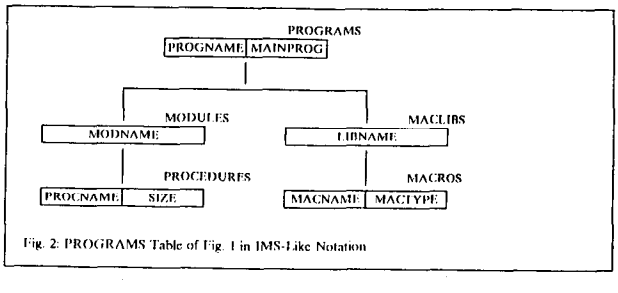

Fig. 2: PROGRAMS Table of Fig. 1 in IMS-Like Notation

```
result      "first level"          "second level"          "third level"
name        attributes             attributes              attributes
Q_RESULT :=
SELECT   [ x.PROGNAME,
           x.MAINPROG,
           MODULES :   ( SELECT  [ y.MODNAME,
                                   PROCEDURES : ( SELECT  [ z.PROCNAME,
                                                            z.SIZE
                                                  ]
                                                  FROM   z IN y.PROCEDURES
                                                  WHERE  z.SIZE ≥ 25
                                                )
                                 ]
                         FROM   y IN x.MODULES
                         WHERE  y.MODNAME CONTAINS '*1*'
                       )
         ]
FROM     x IN PROGRAMS
WHERE    x.PROGNAME CONTAINS '*V01*'
```

Fig. 3: Example of an HDBL Query on the NF^2 Table PROGRAMS

{ Q_RESULT }				
PROGNAME	MAINPROG	{ MODULES }		
		MODNAME	{ PROCEDURES }	
			PROCNAME	SIZE
AIMPV01	QPTEST	PARSLI	PARSER	120
			SCANNER	80
		TIMLI	CONVTIME	30

Fig. 4: Result Table for the Query of Fig. 3 (Applied to the Table of Fig. 1)

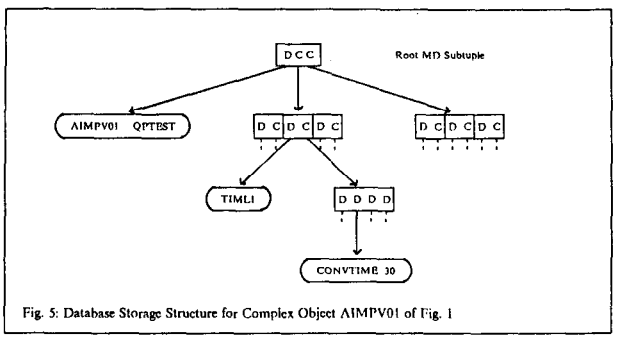

Fig. 5: Database Storage Structure for Complex Object AIMPV01 of Fig. 1

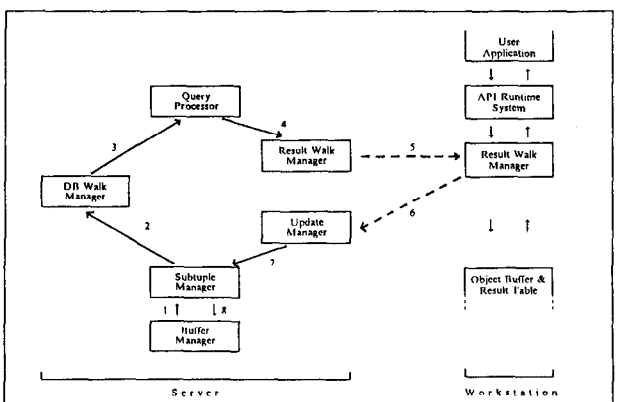

Fig. 6: AIM-P Architecture and Data Flow:
- Data Extraction and Check-out (steps 1 to 5)
- Check-in and Update Materialization (steps 6 to 8)

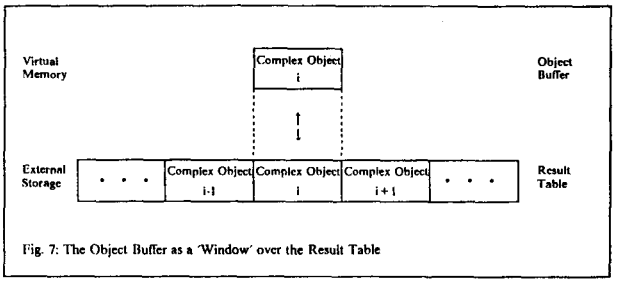

Fig. 7: The Object Buffer as a 'Window' over the Result Table

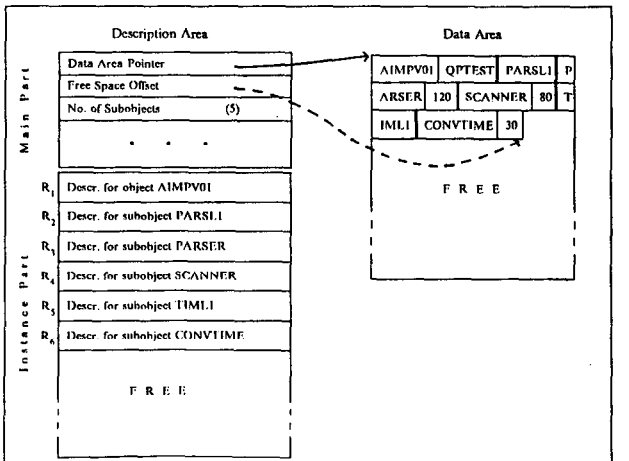

Fig. 8: Object Buffer Contents for Complex Object AIMPV01 of Fig. 4

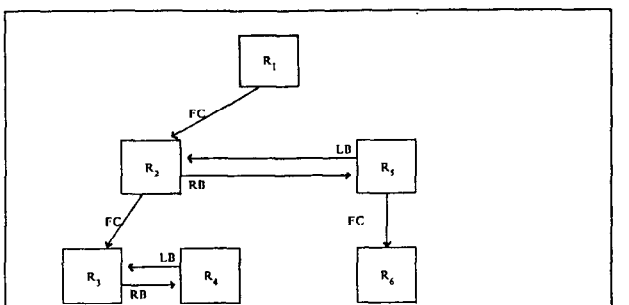

Fig. 9a: Instance Part -- Logical View

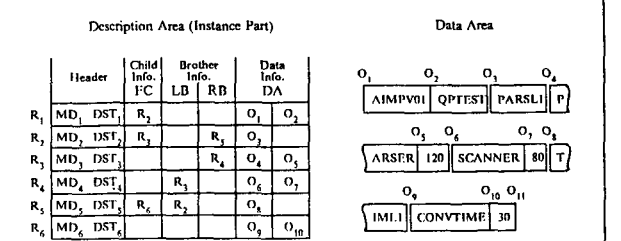

Fig. 9b: Instance Part and Data Area -- Physical View

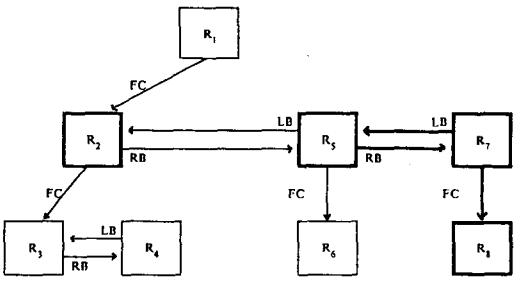

Fig. 10a: Instance Part -- Logical View (After Update Execution)

Description Area (Instance Part)

	Header	Child Info. FC	Brother Info. LB	Brother Info. RB	Data Info. DA	
R_1	MD_1 DST_1	R_2			O_1	O_2
R_2	MD_2 DST_2	R_3		R_5	O_{14}	
R_3	MD_3 DST_3			R_4	O_4	O_5
R_4	MD_4 DST_4		R_3		O_6	O_7
R_5	MD_5 DST_5	R_6	R_2	R_7	O_8	
R_6	MD_6 DST_6				O_9	O_{10}
R_7	MD_7 DST_7	R_8	R_5		O_{11}	
R_8	MD_8 DST_8				O_{12}	O_{13}

Data Area

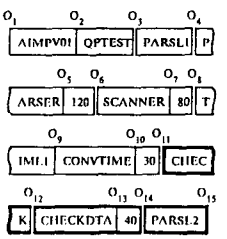

Fig. 10b: Instance Part and Data Area -- Physical View (After Update Execution)

Appendix: Database Walk and Result Walk Operations -- Summary

Database Walk (retrieval)	Result Walk (retr. & upd.)	Short Description
OPEN_DB_TABLE	OPEN_R_TABLE	Opens a (database or result) table
OPEN_DB_WALK	OPEN_R_WALK	Opens a walk on a table or subtable
START_DB_WALK i	START_R_WALK i	Sets a walk on a table or a subtable occurrence *before* object/subobject i
DB_WALK	R_WALK	Moves a walk *on* the next object/subobject within a table or a subtable occurrence
DB_GET	R_GET	Reads attribute values at a walk position
CLOSE_DB_WALK	CLOSE_R_WALK	Closes a walk (incl. all dependent walks)
CLOSE_DB_TABLE	CLOSE_R_TABLE	Closes a table (incl. all walks)
MOVE_FROM_DB_TO_RES		Moves attribute values from a database table (walk position) to a result table (walk position)
	R_PUT	Updates or sets attribute values at a walk position
	R_INSERT	Inserts an object/subobject at a walk position
	R_DELETE	Deletes an object/subobject at a walk position