Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications

N. Kiesel, A. Schürr, B. Westfechtel
Lehrstuhl für Informatik III
Technical University of Aachen
W-5100 Aachen, Germany

Abstract

Modern software systems for application areas like CAD, office automation, or software engineering are usually highly interactive and deal with rather complex object structures. For the realization of these systems a nonstandard database system is needed which is able to efficiently handle different types of coarse- and fine-grained objects (like documents and paragraphs), hierarchical and nonhierarchical relationships between objects (like composition-links and cross-references), and finally attributes of rather different size (like chapter numbers and bitmaps). Furthermore, this database system should support computation of derived data, undo/redo of data modifications, error recovery from system crashes, and version control mechanisms. In this paper, we describe the underlying data model and the functionality of GRAS, a database system which has been designed according to the above mentioned requirements. Furthermore, we motivate our central design decisions concerning its realization, and we evaluate our system by means of the so-called hypermodel benchmark for hypertext databases.

1 Introduction

Building integrated, interactive, and incrementally working tools/environments for application areas like CAD, software engineering, or office automation has been a great challenge. Among other factors such as presentation, control, and process integration, **data integration** is a key issue /ThN 92/. In order to ensure that all information in an environment is managed as a consistent whole, it should be modelled and realized in a uniform way. To this end, a **database system** is needed which takes the specific requirements of these environments into account /DGL 86, Be 87, SZd 87/.

In this paper, we present such a database system which is called **GRAS** (for **GRA**ph **S**torage, /BL 85, LS 88/). GRAS has been developed within the IPSEN project /Na 90/ which is concerned with integrated, structure-oriented software development environments. IPSEN environments consist of both tools cooperating within one working area (e.g. structure-oriented

editors, analyzers, and execution tools for programming-in-the-small), and tools integrating different working areas (e.g. for maintaining consistency between requirements and design). Since the IPSEN project was launched in the mid 80's, GRAS has been extensively used for the implementation of integrated, structure-oriented environments (not only in IPSEN, but also in other research projects). Thus, a lot of experience has been gained, resulting in continuous extensions and improvements of GRAS in response to the requirements of its applications.

In order to cope with the complexity of information managed in an integrated, structure-oriented environment, its underlying database system has to be based upon an **appropriate data model**. This data model should allow us to represent complex object structures in a natural way. While the relational model has proved to be adequate for traditional business applications, it is not well-suited for building applications such as software development environments, hypertext systems, CAD systems, etc. /Be 87, SZd 87/. On the other hand, it seems quite natural to model objects as nodes (with attributes representing their properties) and relations between them as edges. Therefore, we have selected **attributed graphs** as the data model underlying the GRAS system.

Our presentation of the GRAS system is structured as follows:

- In section 2, we give a brief overview of the most important features of the GRAS system and point out relations to other work.
- Section 3 describes the data model underlying the GRAS system.
- Sections 4 and 5 present the architecture of the GRAS system. Section 4 describes the kernel, while section 5 focuses on enhancing layers which extend the functionality provided by the kernel.
- In section 6, we evaluate our design decisions by means of a benchmark which has been run against the GRAS system.
- Finally, section 7 describes the current state of implementation and presents an outlook on future work.

2 Overview and Related Work

The purpose of this section is to provide the reader with an overview of the most important features of the GRAS system and to relate GRAS to other systems. To this end, we discuss the data model, the kernel, and the enhancing layers of GRAS in turn. All topics which are touched upon briefly here will be elaborated more thoroughly in subsequent sections.

As already mentioned in the introduction, we have selected **attributed graphs** as the **data model** underlying the GRAS system. There are a few other database systems which also rely on attributed graphs, e.g. Cactis /HK 89/ or Adage /GRD 90/. In all of these systems, objects are

represented by means of typed nodes which may carry attributes. Binary, directed relations between objects are modeled as edges which don't carry attributes. Apart from these common properties, the data models differ e.g. with respect to the type system, handling of derived attributes, and aggregation. Despite of these differences, the common philosophy behind attributed graphs is to provide a data model which is both **simple** and **general**. In this regard, attributed graphs may be considered as a compromise between attributed trees /RT 88/, which only support hierarchical relations, and the Entity-Relationship model /Ch 76/, which allows for n-ary relations with attributes. Note that both attributed trees and the ER model may be simulated by means of attributed graphs.

The GRAS system has a layered software architecture which may be divided into two parts. The lower part, which is called the **kernel**, provides basic operations for storing and manipulating graph structures. In order to execute such operations efficiently, the following techniques have been applied:

- **Different kinds of data** (e.g. nodes, long attributes, and edges) are mapped onto **separate substorages**. In this way, data necessary for processing different kinds of application requests are stored on different disk pages (see /KD 92/ and /KM 92/ for related approaches).
- Furthermore, an indexing scheme based on so-called **"tries" and static hashing** is used to map data of varying size and structure onto storage locations. By means of an **incrementally and heuristically working clustering algorithm** logically related data are mapped onto adjacent storage locations (i.e. on the same storage page wherever this is possible). This algorithm only incurs a negligible overhead in contrast to those profiling based and batch-oriented clustering algorithms which reorganize whole databases (cf. /TN 92/ for an in-depth comparison of different clustering algorithms).

The upper part of the GRAS architecture consists of **enhancing layers** which incrementally extend the functionality provided by the kernel:

- The **change management** layer comprises all functions which are concerned with logging, storing and executing sequences of change operations. Change management is performed to provide for user recovery /ACS 84/ (undo/redo of user commands), system recovery (recovery from system failures), and deltas (efficient storage of versions). These tasks are handled by GRAS in an integrated way by maintaining logs of change operations. This is done on a high level of abstraction (graph- rather than byte-oriented operations the latter of which form the basis of recording deltas e.g. in EXODUS /CDR 89/) in order to keep deltas short.
- The **event management** layer supports the definition of triggers and associated actions which are called when specified events occur within GRAS (see /DKM 86/ for a general overview about triggers within databases).
- The **scheme and attribute management** layer checks and ensures that all graph operations are consistent with a scheme defining types of graph components and their interrelations. In addition to optional runtime type checking, this layer supports derived attributes, i.e.

node attributes which are calculated from attributes of neighbor nodes. In contrast to Cactis /HK 89/ — which to the best of our knowledge is the only other database system which is based on attributed graphs and supports derived attributes —, the term "neighborhood" is not confined to the 1-context; rather, arbitrarily far reaching attribute dependencies are supported.

- The **concurrency control and distribution** layer manages concurrent access of different applications to shared graphs. It allows to describe the intended interaction mode with other applications by means of a flexible group-oriented access model. Controversial to most other approaches (e.g. /CDR 89/), an operation exchange protocol instead of a data exchange protocol was chosen for interprocess communication.

3 Data Model

In order to cope with the complexity of information managed in an integrated, structure-oriented environment, a formal specification language called **PROGRES** /Sc 89, Sc 91a/b, SZ 91/ has been developed within the IPSEN project. A formal definition of this language including syntax, static and dynamic semantics (the latter of which is defined by means of a calculus based on first-order predicate logic) is given in /Sc 91b/. PROGRES is based on attributed graphs and provides constructs for defining graph schemes (data definition) and complex graph transformations (data manipulation). Its name is an acronym for **PRO**grammed **G**raph **RE**writing **S**ystems: Graph transformations are specified graphically by means of graph rewrite rules which describe the replacement of a left-hand side subgraph with a right-hand side subgraph. The name prefix "programmed" indicates that the application of these graph rewrite rules is controlled by programs composed of deterministic and nondeterministic control structures.

GRAS and PROGRES are related in the following way: GRAS provides **basic operations on graphs** such as creation/deletion of nodes and edges, manipulation and incremental computation of attributes, etc. These operations are checked against a **graph scheme** which is defined by means of the data definition part of PROGRES. Complex graph transformations specified in PROGRES are mapped onto basic operations provided by GRAS (which is the task of a compiler/interpreter). Thus, GRAS serves as kernel of a more comprehensive database development environment for PROGRES (the implementation of which is currently under way).

In the following, we will focus on the definition of graph schemes which control the operations performed by the GRAS database system. Such schemes describe the components of **attributed graphs**: Objects are represented by means of typed nodes which may carry attributes. Binary, directed relations between objects are modeled by edges which don't carry attributes. Thus, simple relations (which occur very frequently) may be represented directly; complex relations (n-ary relations with attributes) have to be simulated. Attributes are either intrinsic (the value is assigned explicitly) or derived. In the latter case, the value is automatically calculated from values of other attributes which belong to the same node or to neighbor nodes.

Analogously, we distinguish between intrinsic relations (edges) and derived relations the latter of which are specified by means of path expressions.

In order to illustrate the capabilities of our data model, we present a sample scheme definition which is drawn from the **hypertext** area (cf. fig. 1). A documentation is divided into sections which may be nested arbitrarily. Each section has a name (title) and is numbered in a hierarchical fashion. Paragraphs, which contain plain text, form the leaves of the tree. The composition tree is augmented with cross references which may be created between arbitrary components.

Abstract node classes are used for modeling sets of nodes with a common interface (and common properties). A node class determines the attributes which all nodes of this class possess, and the relations in which they may participate. Attributes are either intrinsic or derived. In the former case (e.g. attribute *Name* of class *COMPOSITE*), an initial value may be specified; in the latter case, an attribute evaluation rule describes how the value is to be computed (see below).

Node classes are organized into an **inheritance hierarchy**. In our example, *OBJECT* acts as the root of the application specific hierarchy. A subclass inherits from its superclasses all attributes defined in the superclasses, and all relations in which nodes of the superclasses may participate. Multiple inheritance is supported. For example, *INNER* inherits from both *COMPOSITE* and *COMPONENT*. Therefore, a node of class *INNER* has a name and may serve both as source and as sink of *Contains* relations.

Concrete node types are instances of node classes (e.g. *Documentation* is an instance of *DOCUMENT*). Nodes, in turn, are instances of node types. All node types of a certain node class share a common interface, but may differ in their implementations, i.e. in the way derived attributes are calculated. Differing behavior is specified by redefining attribute evaluation rules (not shown in our example).

Edge types are used to represent intrinsic, binary, directed relations between nodes of certain classes. For example, edges of type *Contains* connect *COMPOSITE* to *COMPONENT* nodes. Optionally, the cardinality of an edge type may be specified (the default is many-to-many, i.e. *SOURCE_CLASS [0 : n] −> SINK_CLASS [0 : n]*).

Paths represent **derived relations** and are specified by means of path expressions which are composed of primitives (e.g. *<−Contains−* for following *Contains* edges in the opposite direction) and operators such as loop, sequence, transitive closure, etc. For example, the path *Predecessor* leads from a section to its predecessor section (more precisely : to a preceding node of class *INNER*), if any. *"&"* denotes a sequence, i.e. a concatenation of paths; curly brackets embrace a loop.

The values of **derived attributes** are calculated from the values of other attributes which either belong to the same node or to neighbor nodes. Derivation rules are specified in node class or node type declarations. In our example, nodes of class *INNER* carry derived attributes *Position*

```
node class OBJECT end;
    (*  Root of the class hierarchy. *)
node class COMPOSITE is a OBJECT;
    intrinsic Name : string := "";
end;
    (*  A composite is a named object which contains components. string is a built-in type. *)
edge type Contains : COMPOSITE [1 : 1] -> COMPONENT [1 : n];
    (*  A component is contained in exactly one composite, and a composite contains at least one component. *)
node class COMPONENT is a OBJECT end;
edge type Precedes : COMPONENT [0 : 1] -> COMPONENT [0 : 1];
    (*  Components of a composite are ordered linearly. *)
edge type RefersTo : COMPONENT [0 : n] -> COMPONENT [0 : n];
    (*  Cross references between arbitrary components. Each component may have arbitrarily many incoming
        and outgoing references. *)
node class DOCUMENT is a COMPOSITE;
    intrinsic Author : string := "";
end;
    (*  Root of a hypertext document. *)
node type Documentation : DOCUMENT end;
    (*  In our sample specification, only one type of document is defined. *)
node class TEXT_BLOCK is a COMPONENT;
    intrinsic Contents : Text := emptyText;
end;
    (*  Leaves of the component hierarchy. Text is a user defined type. *)
node type Paragraph : TEXT_BLOCK end;
node type Subparagraph : TEXT_BLOCK end;
    (*  Paragraphs and subparagraphs are differently typed instances of class TEXT_BLOCK. *)
node class INNER is a COMPOSITE, COMPONENT;
    derived Position : integer = [ self.Predecessor.Position + 1 | 1 ];
    derived Number : string = [   self.Father.Number & "." & IntToString(self.Position) |
                                  IntToString(self.Position) ];
end;
    (*  Inner nodes of the component hierarchy. The position of such a node is either obtained by incrementing the
        position of its predecessor or is set to 1 if no predecessor exists. Analogously, its number is calculated by
        concatenating the number of the father and the string representation of its own position, or by merely con-
        verting its own position if no father exists. *)
node type Section : INNER end;
path Predecessor : INNER [0 : 1] -> INNER [0 : 1] =
    <-Precedes- & { not instance of INNER : <-Precedes- }
end;
    (*  Path to the preceding INNER node. Notice that INNER and TEXT_BLOCK nodes may be mixed freely;
        therefore, an iterator is needed to obtain a predecessor which carries a Position attribute. *)
path Father : INNER [1 : n] -> INNER [0 : 1] =
    <-Contains- & instance of INNER
end;
```

Fig. 1 : Graph scheme of documentations

and *Number*. In both cases, conditional expressions are used for calculating their values. The alternative list is enclosed in square brackets; alternatives are separated by vertical bars. When a conditional expression is evaluated, the first alternative is chosen which yields a defined value. Notice that neighbor nodes don't have to belong to the 1-context; rather, they need only be connected to the node of interest via a path which may be arbitrarily long.

Let us summarize the main features of our data model:

- **Attributed graphs** are both **simple** and **general**. Other data models may be simulated by means of attributed graphs (see below).
- Complex graph queries may be defined with the aid of **derived relations** and **attributes** which are specified using powerful, declarative language constructs. With respect to the usual terminology of database systems, these derived informations are **views** which will be constructed on demand and maintained incrementally.
- The **stratified type system** (classes are types of types) avoids the theoretical pitfalls of reflexive type systems with the 'type is the type of all types including itself' assumption (cf. /MR 86/) and allows for typed type parameters of operations.
- **(Multiple) inheritance** of node classes makes it possible to specify polymorphic graph operations which rely on dynamic binding of attribute evaluation rules and which exploit the knowledge contained in class hierarchies for pattern matching purposes.
- **Relations** are **bidirectional**, i.e. they may be traversed in both directions. Therefore, the application is not responsible for keeping pairs of unidirectional pointers consistent (as e.g. in IDL /IDL 87/).
- **Referential integrity** is guaranteed, i.e. when a node is deleted, all adjacent edges are deleted, as well.
- **Edge types** are declared **separately**, i.e. edges are not treated as pairs of pointer-valued node attributes which would significantly reduce the reusability of node classes (the declaration of pointer attributes within node classes prevents reuse of these node classes within another context with a different set of relationships; cf. /Ki 91/).

Comparing our data model to data models underlying other database systems, we observe that there are a few other database systems which rely on attributed graphs (e.g. Adage /GRD 90/ and Cactis /HK 88, HK 89/). The data models of these systems differ e.g. with respect to the type system, aggregation, and handling of derived attributes (the latter of these aspects will be discussed in section 5). Rather than going into details of these graph-based models, let us discuss the general relations to some other, more wide-spread data models:

- **Abstract syntax trees** have been used in various syntax-aided software development environments (or generators for such environments) such as e.g. Gandalf /HN 86/ and CPSG /RT 88/. Abstract syntax trees may be — and actually have been — simulated on top of GRAS. The advantage of using a more general framework — graphs instead of trees — lies in the uniform representation of tree and nontree relations, i.e. abstract syntax trees may be augmented with edges representing control flow, data flow, etc.. This leads to the more general notion of abstract syntax graphs /ELN 92/.

- Database systems such as DAMOKLES /DGL 86/ and PCTE /ECMA 90/ are based on **extended Entity-Relationship models** (which ultimately have their roots in the ER model proposed by Chen /Ch 76/). These models differ from attributed graphs at least in two ways: First, relationships may be attributed; second, there is a predefined type of relationship which provides for aggregation (i.e. complex objects). Furthermore, systems like PCTE introduce even more categories of relationships (*composition*, *reference*, *implicit*, *designation*, and *existence* in the PCTE ECMA standard). In order to avoid commitment to a complicated data model which incorporates many decisions which are subject to debate, we decided to keep our data model as simple as possible and to offer the user a powerful data definition and manipulation language for specifying different variants of EER models as desired.
- Recently, a couple of database systems (e.g. O_2 /De 91/, Objectstore /LLOW 91/, and GemStone /BOS 91/) have been developed which are based on an **object-oriented data model**. In an object-oriented framework, nodes would be represented as objects, and operations would be attached to single nodes only. On the contrary, operations such as creating a cross-reference between two nodes are attached to whole graphs rather than to single nodes in PROGRES; only operations for calculating attribute values refer to nodes (and their context) rather than to graphs. Another difference concerns the treatment of relations, which in many object-oriented data models have to be simulated through pairwise pointers.

4 Architecture: The Kernel

In the next two sections, we discuss the internal architecture of the GRAS system. In order to provide for reusability and flexibility, we have designed a **layered software architecture** which is divided into two parts:

- Its lower part (the kernel), which is depicted in fig. 2 and will be explained in this section, consists of layers of **data abstraction** each of which has an own, carefully designed data model and supports the implementation of a wide spectrum of different applications (with different data models) on top of it.
- By way of contrast, all layers of the upper part, which will be explained in the following section, rely on a common, graph-oriented data model. Each layer incrementally adds a set of **logically related services** to the functionality of the layer beneath it.

When proceeding through the layers of the architecture, we describe their **functionality** (i.e. the interface provided to upper layers) as well as their **realization**. We motivate our central design decisions not only with respect to individual layers, but also with respect to their arrangement (i.e. their position within the architecture).

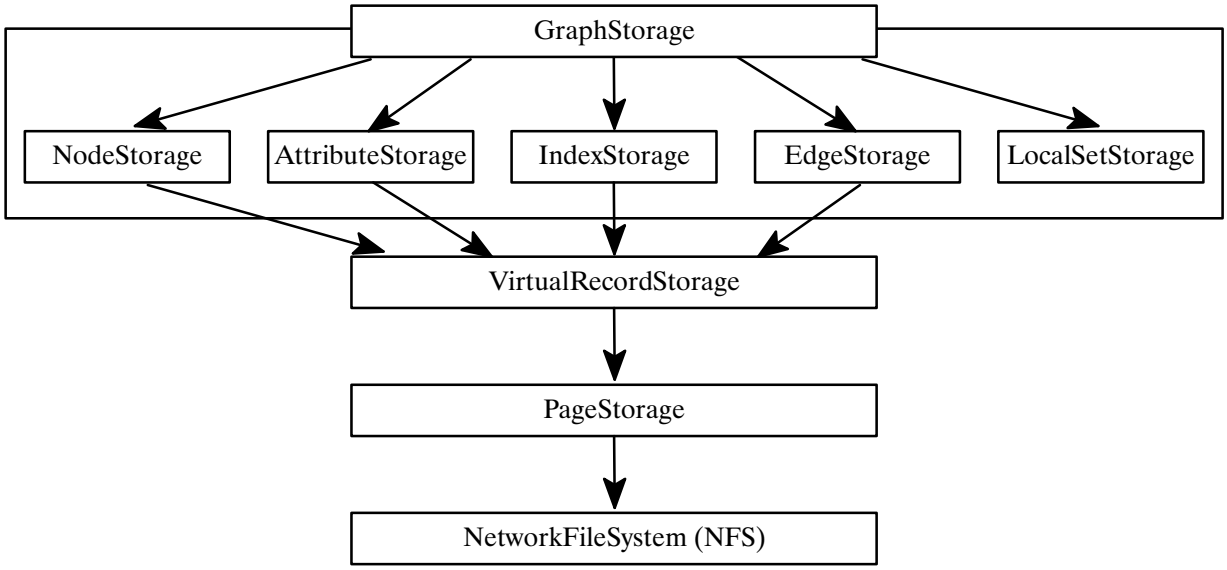

Fig. 2 : Lower layers of the GRAS architecture

4.1 The ADT Graph Storage

The *GraphStorage* is the topmost data abstraction layer. It provides a **graph-oriented interface** which exports all those GRAS resources which are necessary for creating, deleting, and accessing all kinds of graph components. In particular, the interface offers **write operations**

- for creating and deleting **typed nodes**,
- for creating and deleting **typed directed edges**,
- for initializing and modifying arbitrarily long **node attributes**,
- and finally for maintaining **attribute indexes**.

To characterize all **read operations**, graphs are considered to be sets of tuples of the following forms:

- node := (node key, node type).
- edge := (source node key, edge type, sink node key).
- attribute := (node key, attribute name, attribute value).

The GRAS system supports almost all possible **partial match queries** for the relations enumerated above[1]. There are e.g. resources for determining the in- and out-context of a node, for retrieving all nodes with a certain attribute value, and for returning the names of all attributes which have defined values for a certain node. The results of such partial match queries are either single elements or sets of elements or even binary relations. Therefore, the GRAS system offers facilities for efficiently handling (ordered) **temporary sets and relations**.

1. Partial match queries of the forms (? , edge type, ?) and (? , ? , attribute value) are not supported.

Internally, graphs are mapped onto **separate substorages** for each kind of data:

- The *LocalSetStorage*, the only storage which contains volatile data, supports storing and subsequent processing of temporary query results (sets and binary relations).
- The *NodeStorage* has been designed as a repository for nodes themselves and for their most frequently accessed, rather short attributes.
- Long node attributes (byte sequences of almost arbitrary length) are mapped onto the *AttributeStorage*.
- The *IndexStorage* is responsible for maintaining maps of attribute values to node keys and for executing corresponding partial match queries.
- Finally, the *EdgeStorage* provides operations for creating/deleting edges and for executing partial match queries with respect to these edges.

Storing different kinds of data within different substorages has been one of our most important design decisions. In this way, data necessary for processing different kinds of application requests are stored on different disk pages[2]. In particular, the **separation of large attributes** from all other kind of data reduces the volume of data which must be transferred into main memory for the execution of structure-oriented queries considerably. And keeping **relations** between objects (nodes) **separate** is an approved technique for accelerating navigational queries (in contrast to “path indexes” in /KD 92/ and “access support relations” in /KM 92/, GRAS even avoids duplication of intrinsic relations within an additional index structure).

4.2 The ADT Virtual Record Storage

All permanent storages described in the previous section share the following characteristics:

- They contain persistent data of **dynamically varying size** which ranges from a few hundred bytes up to some megabytes.
- Each data portion stored in one of these substorages has to be identified by a **unique database key**. These database keys are internally used for establishing cross-references between different portions of data. They are also used for representing nodes in application specific data structures. These database keys must not vary during the whole lifespan of the data portions they belong to.
- Furthermore, some of these substorages have to support efficient retrieval of data portions, selected by (only a part of) their database keys (**partial match queries**).

Having these common characteristics in mind, we have implemented one parametric *VirtualRecordStorage* with a **record-oriented interface**. This storage is used as a common basis for the realization of all specialized permanent substorages mentioned above. The parameters are mainly used for defining the structure of the records. Any record has a database key of fixed size and additionally may consist of

2. Clustering within one substorage will be discussed below.

- a data area of fixed size (e.g. for storing node types),
- a number of data areas of dynamically varying size (e.g. for storing large attributes),
- a number of areas for ordered sets of database keys (e.g. for storing references to other records).

In order to clarify the process of implementing substorages as special instances of the *Virtual-RecordStorage* let us consider the most complicated case, the realization of the *EdgeStorage*. As already explained before, the components of this storage are tuples of the form

(source node key, edge type, sink node key)

and we have to answer partial match queries with two specified components

(source node key, edge type, ?) and (?, edge type, sink node key)

as well as partial match queries with one specified component

(source node key, ?, ?) and (?, ?, sink node key) .

For efficiently processing queries with a specified source (sink) node key, this node key must be a prefix of the corresponding record key (see below). Therefore, we have to store two **permutations of edge triples** as records with the following layout:

- The record's key consists of a source (sink) node key *k* followed by a permutation flag (distinguishing between source and sink keys) and an edge type *t*.
- The fixed size data area has length 0.
- And the last record component is an ordered set of all those sink (source) node keys which belong to any edge with source (sink) *k* and type *t*.

Records of the virtual storage are mapped onto locations on external storage pages in the following way: The unique database keys which identify records are hashed onto physical addresses. A special variant of binary index trees, so-called "**tries**" (cf. e.g. /ED 88/), is used for selecting a record's page, and **static hashing** is used for determining an appropriate position on a selected page.

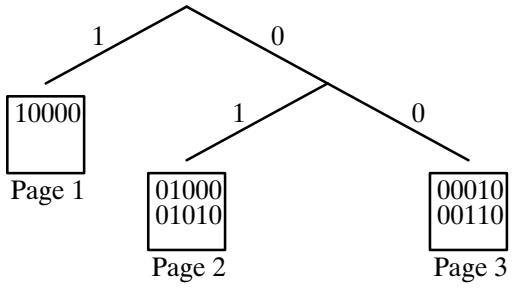

Fig. 3 : Indexing and clustering records

Fig. 3 contains a trivial example of a substorage with five records distributed over three pages. In this case, the first key bit of the record '10000' and the first two bits of all other records are

used to determine their pages (the first page contains all records with '1' as key prefix, the second page all records with '01' as key prefix, and the third page all records with '00' as key prefix). All remaining key bits are input to the static hashing function which computes record positions on selected pages.

As already indicated by fig. 3, any substorage has its own index tree so that structural data (e.g. edges) and nonstructural data (e.g. attribute values) are kept on separate pages. The data representing index trees are stored on separate pages, too. Beside this overall strategy for storing **different kinds of data on different pages**, the GRAS system additionally tries to cluster records within one substorage in the following way: The algorithms which generate database keys for new records and thus determine the page positions of new records are sensitive to requests of the application layer, i.e. the application may specify neighborhoods for records which are often accessed together (e.g. nodes connected by certain edges)[3]. This information is used for **clustering logically related records** onto adjacent storage locations. More precisely, the computation of a database key for a new record may be influenced by the database key of an already existing record (determined by the application), so that

(1) the dynamic hashing function maps the new record onto the same page as the already existing record whenever possible (both keys must have a sufficiently long identical prefix),
(2) subsequent splitting of pages does not destroy these record clusters (additional key bits, currently not used for page selection, must be identical, too),
(3) and the index trees do not degenerate in the case of splitting an old page into two new pages by mapping almost all records of the old page onto one of the new pages (the key creation algorithm has to guarantee an almost uniform distribution of additional key bits).

Note that it is difficult and sometimes even impossible to find an optimal solution for both the conditions (2) and (3). Consider again the situation of fig. 3 which contains a somewhat artificially constructed but illustrative example for this problem: When creating a new record "nearby" record '01010', we have the following alternatives for the new record key:

(1) Key '0100?'[4]: this choice has the consequence that splitting page 2 produces one empty page and one page containing all records of the old page (violation of condition (3) above).
(2) Key '011??': this choice has the consequence that splitting page 2 maps the old record '01010' and the new record '011??' onto different pages (violation of condition (2) above).

Therefore, we have tested different **key creation heuristics** which try to find the right balance between the risk of producing unbalanced index trees and the risk of scattering logically related portions of data over many pages. Furthermore, the creation of database keys and thus the selection of physical record addresses may even be controlled completely by a GRAS application which may exploit additional **application specific knowledge** to improve clustering (see benchmark results in section 6.3).

3. The application specifies neighborhoods for node records only. The distribution of edges and node attributes over pages is the same as for the nodes they belong to.
4. Question marks in node keys must be replaced either by '0' or by '1'.

To summarize, the presented indexing scheme and its accompanying key creation algorithm have the following **characteristics**:

- The hashing function, which maps record keys onto page locations, is order preserving and thus allows for efficiently retrieving all records with a given key prefix.
- Order preserving hashing functions in general are not able to guarantee a uniform distribution of records over pages. This is the task of our key creation algorithm.
- The algorithms for inserting, deleting, and searching records are rather simple and straightforward to implement.
- Nevertheless, they can deal with data of rather dynamically varying size without any needs for maintaining overflow pages or for global reorganizations of data or index structures.
- And the index trees themselves (tries) are very small in comparison to the data they are used to address and, therefore, may and should normally be kept in main memory.

Thus, our indexing scheme is well-suited for **dynamically growing and shrinking medium sized databases** (with each database having the size of a typical engineering document). In this case, tries for addressing pages but not the addressed data itself fit into main memory. On the contrary,

- main memory indexing schemes — like those described in /AP 92/ and /LC 86/ — are tuned for the case that a whole database fits into main memory,
- whereas B-trees /BC 72/ and their variants assume that index structures themselves do no longer fit into main memory and traversing these structures requires disk accesses, too[5],
- and dynamic hashing techniques /ED 88/ which either use directories — like 'extendible hashing' /FNPS 79/, 'virtual hashing' /Li 78/ etc. — or which address their data without any index at all — like 'linear hashing' /La 80/, 'modified dynamic hashing' /Ka 85/ etc. — suffer from the following drawback: they require global reorganizations of index structures or data from time to time in order to avoid expensive maintenance of overflow chains[6].

4.3 The ADTs Page Storage and Network File System

The *PageStorage* provides a **page-oriented interface**; each page is a sequence of bytes which has a fixed length. Graphs are not required to fit into main memory; therefore, the page storage maintains a page cache in main memory and controls transfer of pages between disk and main memory. Paging is driven by an LRU strategy which takes additional factors such as frequency of access and priority into account (log pages are assigned the highest priority, followed by index pages and data pages). The size of the page cache may be fixed as needed. If the underlying operating system provides a sufficiently large virtual memory, the size may be

5. Furthermore, B-trees as well as B*-trees keep record keys as part of their index structures and, therefore, tend to require more storage space than tries.
6. Analysis and simulation results of /Fl 83/ and /La 88/ indicate that the above mentioned dynamic hashing techniques might be superior to our approach in the case of large databases, which consist of many thousands of pages and which would require deep tries.

chosen so that all graph operations are eventually performed in virtual memory. In this way, paging is delegated to the operating system.

Finally, the *NetworkFileSystem* layer is the interface to the underlying operating system's **distributed file management**. It maps page sequences onto files. Since its interface is independent of a particular operating system, it is an easy task to port the GRAS system to another operating system. To this end, only few pages of source code implementing the file system layer have to be adapted or written anew.

5 Architecture: Enhancing Layers

All layers on top of the GRAS system's kernel *GraphStorage* (see fig. 4) rely on a common, graph-oriented data model. Each layer incrementally adds a set of **logically related services** to the functionality of the layer beneath it. In contrast to the last section, which described layers top-down, we proceed bottom-up through the layers of the upper part. In this way, it is easier to understand what each layer adds.

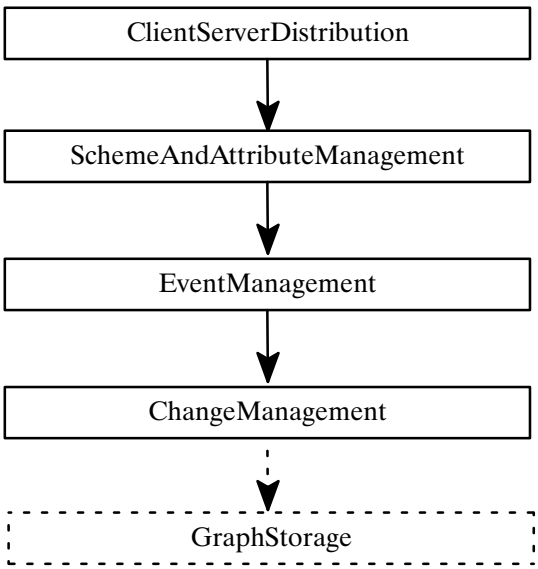

Fig. 4 : Upper layers of the GRAS architecture

5.1 The Change Management Layer

The *ChangeManagement* /Br 89/ layer is placed above the ADT *GraphStorage* which provides all resources necessary for creating, modifying, deleting, and accessing all kinds of graph components. It adds to this layer all functions which are concerned with logging, storing, and executing sequences of change operations. Change management is performed to provide for

- user recovery (undo/redo of user commands),
- system recovery (recovery from system failures), and
- deltas (efficient storage of versions).

These tasks are handled by GRAS in an integrated way by maintaining logs of change operations.

Following /ACS 84/, we define **user recovery** as recovery actions which are controlled by the user rather than by the system which he uses. For example, the user may undo a command which he has activated inadvertently, he may switch back and forth between breakpoints when debugging a program, etc. GRAS supports implementation of user recovery in the following way: While a graph is open, an application may define **checkpoints**. Typically, this is done when the execution of a user command terminates. By means of **undo** and **redo** operations, the application may switch back and forth between arbitrary checkpoints. Checkpoints are ordered sequentially. Note that undo and redo are constrained such that they always yield a semantically consistent state; this could not be guaranteed if e.g. selective undo were supported (undo command i, but not commands i+1, . . . , n).

Checkpoints are also used for **system recovery**: On system failure, GRAS tries to restore the most previous checkpoint. Furthermore, system recovery is supported by means of **nested transactions** which provide a useful means for implementing user commands in a layered architecture: Each application layer defines a corresponding level of consistency and uses transactions to guarantee atomicity of the operations which it provides. In particular, each layer is able to abort operation sequences of the next lower level which lead to an inconsistent state from its point of view. Note that checkpoints are treated as boundaries of **top-level transactions**.

In addition to supporting undo/redo 'in the small', a software development environment has to provide for **undo/redo 'in the large'**. This is the task of version control /Ti 85/. While GRAS does not incorporate a specific model of version control (which is nearly always subject to debate), it does provide flexible mechanisms for efficient storage of versions by means of **graph deltas** /We 89/. Here, the term 'delta' denotes a sequence of operations that, being applied to a version *v1*, yields another version *v2*. Instead of storing all versions of one document completely, it suffices to store a few of them completely and reconstruct the others by means of deltas.

GRAS provides for **flexible delta control**: Deltas save storage space at the cost of access time. Before a version may be operated on, it may have to be reconstructed. Thus, the (optional) use of deltas is controlled by the application in order to achieve an appropriate balance of storage and runtime efficiency. Thereby, the application may choose between **forward** and **backward deltas**. This is illustrated in fig. 5 which shows a version tree (part a) and different examples for the storage of its four graph versions (parts b-d):

- Only forward deltas are used (part b).
- The most recent version on the main trunk is stored completely; all other versions are reconstructed by means of backward and forward deltas (part c). This solution was pioneered by RCS /Ti 85/.
- Only backward deltas are used (part d). In this case, there may be multiple ways to reconstruct a version. Then, GRAS selects the most efficient one automatically.

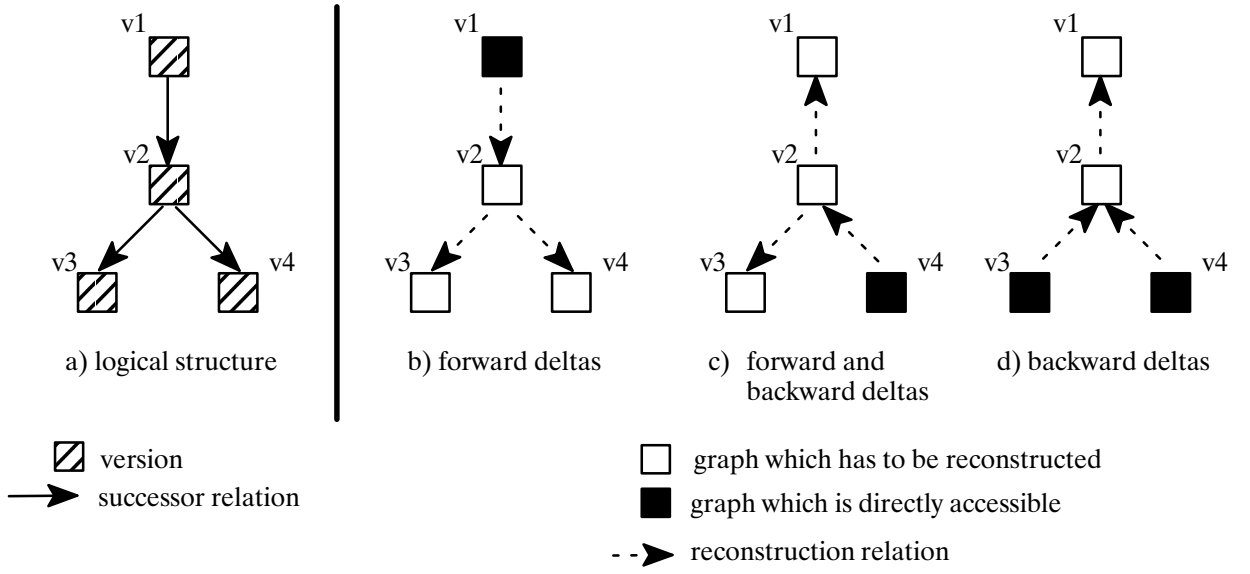

Fig. 5 : Flexibility of delta control

Virtually all functions of change management are implemented in a uniform way by means of **logs** of change operations on graphs. Two kinds of logs are needed: The **forward log** is used to implement redo, forward deltas, and recovery from system crashes[7], while the **backward log** is used analogously for undo, backward deltas, and transaction abort. Thus, logs are reused in an elegant manner for multiple purposes. Particularly, logging of change operations yields a delta on the side so that its costly a-posteriori reconstruction may be avoided. However, direct use of logs may be inefficient because the effect of one operation may be overridden by subsequent operations. Therefore, logs are compressed a-posteriori by removing redundant operations (which is still less costly than performing a *diff*-like a-posteriori comparison).

Logging is performed between *EventManagement* (see section 5.2) and *GraphStorage* layers for the following reasons:

- On the one hand, logging has to be done beneath event handling because otherwise event handlers would be invoked on executing undo and redo operations, i.e. event handlers would be invoked “one time too many”.

7. For system recovery, forward logs are supplemented by a shadow page mechanism in the *PageStorage* layer which prevents any corruptions of an open graph’s initial state. After “hard” system crashes, the most recent consistent graph state is reconstructed by applying this forward log to the still existing initial graph state until the last recorded check point.

- On the other hand, logged operations should be as complex as possible in order to obtain short deltas. Since each operation on one layer is mapped onto multiple operations on the next lower layer, logging is performed on the highest GRAS-internal layer of data abstraction, namely the graph-oriented layer.

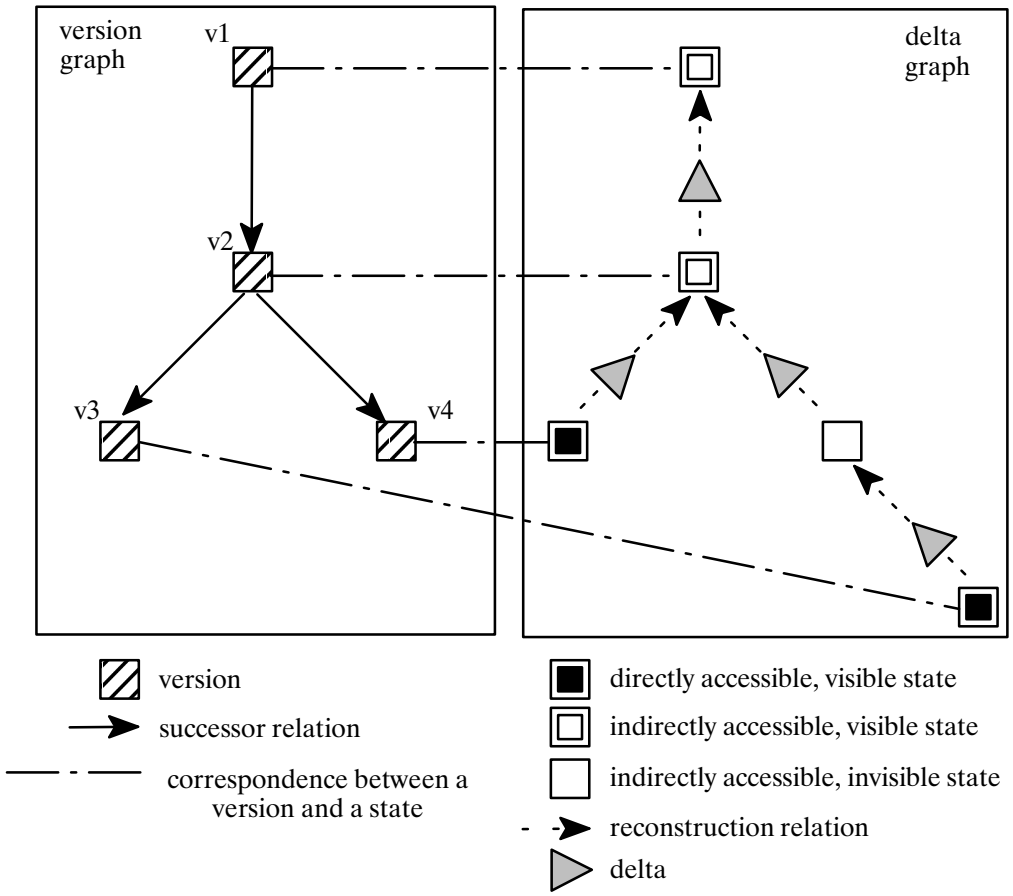

Fig. 6 : Example of a delta graph (right hand side) and its relations to the corresponding version graph (left-hand side)

In order to execute **coarse-grained operations** (e.g. *Open, Close, Delete, Copy* operations on graphs), GRAS maintains an internal data structure which is invisible for applications. This data structure is modeled and realized as a graph which is called **delta graph**. Fig. 6 shows an example of an (internal) delta graph and its relations to a corresponding (external) version graph. In general, a delta graph consists of the following components:

- **Graph state nodes** represent states which may be classified as follows: Firstly, a state is either directly accessible, or it is reconstructed by application of deltas. Secondly, a state is either visible to the application, or it is used internally as an intermediate state that is produced in the course of reconstruction of a visible state.
- **Delta nodes** represent sequences of graph operations which have been executed within one session (from *Open* to *Close*).

- By means of **reconstruction edges**, a delta is related to the source state to which it is applied, and to the sink state which it produces.

For example, when the application invokes an *Open* operation on version *v1*, GRAS retrieves the corresponding state node which represents an indirectly accessible state. Then, it searches for directly accessible states from which this state may be reconstructed (in this case, the nodes for *v3* and *v4*). The state with the shortest distance (measured in accumulated length of deltas) is selected for reconstruction (this may well be the state for *v4* even though the length measured in numbers of deltas is larger than for *v3*).

Let us summarize the main features of our approach to change management (which is described more comprehensively in /Br 89, We 89, We 91/:

- GRAS provides for **flexible delta control**, i.e. the application controls whether and how deltas are used for the storage of versions.
- GRAS supports the realization of version control without introducing a version model (**separation of modeling and realization of version control**).
- GRAS supports **system** as well as **user recovery**.
- All functions of change management are implemented in a uniform manner by means of **logs**. Logs are maintained on a **high level of abstraction** in order to keep deltas short.

Comparing our approach to other work, we observe that traditional version control systems such as SCCS /Ro 75/ or RCS /Ti 85/ are different in the following respects: Applications can't control the use of deltas, modeling of version control is mixed up with its realization, and deltas are constructed a-posteriori[8] such that undo/redo has to be supported by a different mechanism. In contrast, more recent systems such as Gypsy /Co 89/ and EXODUS /CDR 89/ are closer to GRAS inasmuch as they provide for flexible delta control and separate modeling from realization of version control.

While Gypsy (unlike GRAS) relies on a-posteriori construction of deltas, EXODUS constructs deltas on the side when performing change operations. In contrast to GRAS, EXODUS relies on **data sharing**: Data which are common to multiple versions are physically shared among them. This is achieved by applicative operations on tree-like structures (EXODUS uses B^+ trees for storing file objects). Similar techniques have been applied e.g. in /FM 86/ and /Al 88/. Although this approach seems attractive because versions are always immediately accessible[9], it has been ruled out for the following reason: Data sharing techniques as they have been applied in other systems operate on a low level of abstraction. Within the GRAS system, such a technique would be applied on the page level: Each graph is realized by an index tree the leaves of which point to storage pages. Small changes on the graph level (e.g. creation of a single node) may imply comprehensive physical reorganizations on the page level. Therefore, deltas

8. An algorithm for a-posteriori construction of deltas which is applicable to arbitrary non−text files (in contrast to the built-in algorithms of SCCS and RCS) is presented in /Re 91/.
9. When following the operation-based approach, reconstruction costs may be reduced by introducing a cache of recently reconstructed versions (as it is done e.g. in Gypsy).

on this level would become too large to be useful for an application. Note that this is not an argument against data sharing in general; rather, we argue against the usage of data sharing on a low level of abstraction. To the best of our knowledge, adequate techniques for high-level data sharing in attributed graphs have not yet been developed[10].

5.2 The Event Management Layer

Event/Trigger mechanisms are a fundamental concept of most so-called **active database systems** /DKM 86, Da 88, Ch 89, BM 91/. They are also used in modern applications for separating the user interface component from the functional part of the application. Their possible usage includes

- the incremental supervision of certain consistency constraints,
- the incremental computation of derived data like derived attribute values or relationships,
- and the a-posteriori integration of different applications accessing the same data structure.

In GRAS, triggers consist of the following four components:

(1) An **event** determining the type of the graph modification to be observed, as e.g. the insertion or deletion of edges with a given edge type.
(2) An **action** to be executed when a graph modification matches (directly or as a side effect by another operation) the given event.
(3) A **priority** to order the concurrent raising of multiple triggers for the same event at the same node.
(4) An additional application context **parameter** which is one of the arguments for the called action.

GRAS offers resources for defining and manipulating such triggers. Due to the special characteristics of the GRAS system, the class of possible **events** is fixed and determined by the possible **basic state modifications of graphs**: insertion, manipulation, and deletion of nodes and edges. This is compensated by the possibility to use arbitrary application-defined procedures for actions. Therefore, further restrictions for the execution of actions can be given within the action code.

Whenever an event happens for which one or more triggers are defined, their actions are called sorted by their priority with a node key determining the current event context and the application context parameter. In the normal case, actions are called implicitly as a side-effect of a graph modification performed by the application, but they can also be raised explicitly using a special GRAS operation. By raising and handling one of the predefined "synthetic" events, the event management can also be used for **inter-client communication** (see also section 5.4).

10. The Smalltalk-based PIE system /GB 82/ which arranges nodes and attributes in different layers is not a satisfying solution since the runtime of operations increases with the number of layers which have to be searched (roughly speaking, for each version a new layer is needed which contains the changes relative to the previous version).

Up to now, the GRAS system only supports implicit **post-events** for a restricted subset of all possible basic graph modifying operations. Especially when using triggers for realizing constraints, one often wants to execute an action before the event happens and perform the triggering operation only when the action returns a positive acknowledge.

Another aspect which currently undergoes a major redesign copes with the execution of actions. As GRAS is a distributed system without shared memory, application-defined actions must be executed either in the context of the client or in the context of the server. In the current GRAS version, executing an action is performed in the context of the trigger−defining client by means of an **upcall service** within the communication layer. However, this approach is problematic for certain classes of actions which logically belong to the graph (e.g. dynamic constraints). These triggers should of course only be defined once for each open graph copy. Therefore, GRAS will be extended to support **persistent triggers** which belong to their graphs and which are executed in the context of the server.

5.3 The Scheme and Attribute Management Layer

Presenting the GRAS system's basic graph operations, we have neglected one important requirement. The system has to **preserve a graph's consistency** with respect to its graph scheme by rejecting forbidden graph modifications and by recomputing derived attribute values. Concerning the documentation example of section 3 (cf. fig. 1), all operations creating e.g. two emanating *Precedes* edges at one *COMPONENT* node must be rejected, and insertions or deletions of *INNER* nodes eventually trigger the reevaluation of dependent *Position* and *Number* attributes.

In order to be able to fulfill these tasks, information about a graph's class hierarchy, its attribute dependencies and evaluation functions, etc. must be available. Therefore, the GRAS interface comprises a group of operations for constructing internal graph schemes which provide all these informations in an efficiently accessible format. Such a **graph scheme** may be extended arbitrarily during its whole lifetime so that the upward-compatibility of graph-based tools and their data structures is guaranteed.

Discussing the realization of this part of the GRAS architecture, we focus onto its most important task: the incremental evaluation of derived attributes. For this purpose, we have implemented a variant of the well-known **two-phase, lazy attribute evaluation algorithm** (presented e.g. in /Hu 87/) which has already been used successfully within another graph-oriented database management system (Cactis /HK 88, HK 89/). This demand-driven algorithm uses a potentially cyclic static attribute dependency graph containing all information about possible attribute dependencies and evaluation rules, and it works as follows:

(1) **Phase 1**: The assignment of a new value to an intrinsic attribute or the insertion/deletion of certain edges trigger the **invalidation** of all potentially affected derived attributes (propagation stops at already invalid attributes).
(2) **Phase 2**: The reevaluation of an invalid attribute will be delayed until the first attempt to read its value. During its then necessary **reevaluation**, read accesses to other attributes' values may raise evaluation processes for these attributes, too (a bookkeeping mechanism guarantees the abortion of the attribute evaluation process in the presence of forbidden cyclic attribute dependencies).

This rather primitive two-phase algorithm is at least equivalent to all other incremental attribute evaluation algorithms if

- almost all graph (attribute) changes result in changes of all potentially affected attributes,
- or a graph contains many rarely accessed attributes with often changing values.

Note that both conditions are fulfilled in the case of our *Documentation* example (and in many other cases, too). Each insertion or deletion of a section changes *Number* and *Position* attributes of all following sections and their subsections, but only *Number* and *Position* attributes of currently displayed sections (and their predecessors) must be up-to-date. For a more detailed discussion of the advantages and disadvantages of different graph-based incremental attribute evaluation algorithms the reader is referred to /ACR 87, Hu 87, Sc 91b/.

During the adaptation of the algorithm to the special needs of GRAS we encountered one major problem which has not been addressed yet by the graph attribute evaluation algorithms of /ACR 87, ACR 89, HK 88, HK 89, Hu 87/: The language PROGRES allows for the definition of **complex n-context attribute dependencies**. Therefore, graph modifications at unrestricted far away locations may influence an attribute's value. The *Documentation* specification of fig. 1 contains an example for the usefulness of n-context attribute dependencies. There, the *Position* of a *Section* will be computed by incrementing the *Position* of its preceding *Section* (or by assigning the value 1 in the case of the first *Section*). In order to find this preceding *Section* we have to follow a potentially unrestricted number of *Precedes* edges from sinks to sources skipping all *COMPONENTS* without *Position* attributes (i.e. *Paragraphs* and *Subparagraphs*).

In order to be able to handle these far reaching attribute dependencies, we have introduced the concept of **virtual attributes** (cf. /Sc 91b, Hö 92/) in GRAS. These attributes are invisible to application programs, and they even don't possess values of their own (with exception of an invalid flag). Their only purpose is to reduce the difficult problem of propagating invalid flags in the case of n-context dependencies to the already solved 1-context problem. In the case of our *Documentation* example we have to introduce one virtual attribute *VAtt* which propagates changes of *Position* attributes across an arbitrary number of *COMPONENTS* without *Position* attributes. Consider e.g. the deletion of the first *Section* in fig. 7. This operation initiates the following **propagation process**:

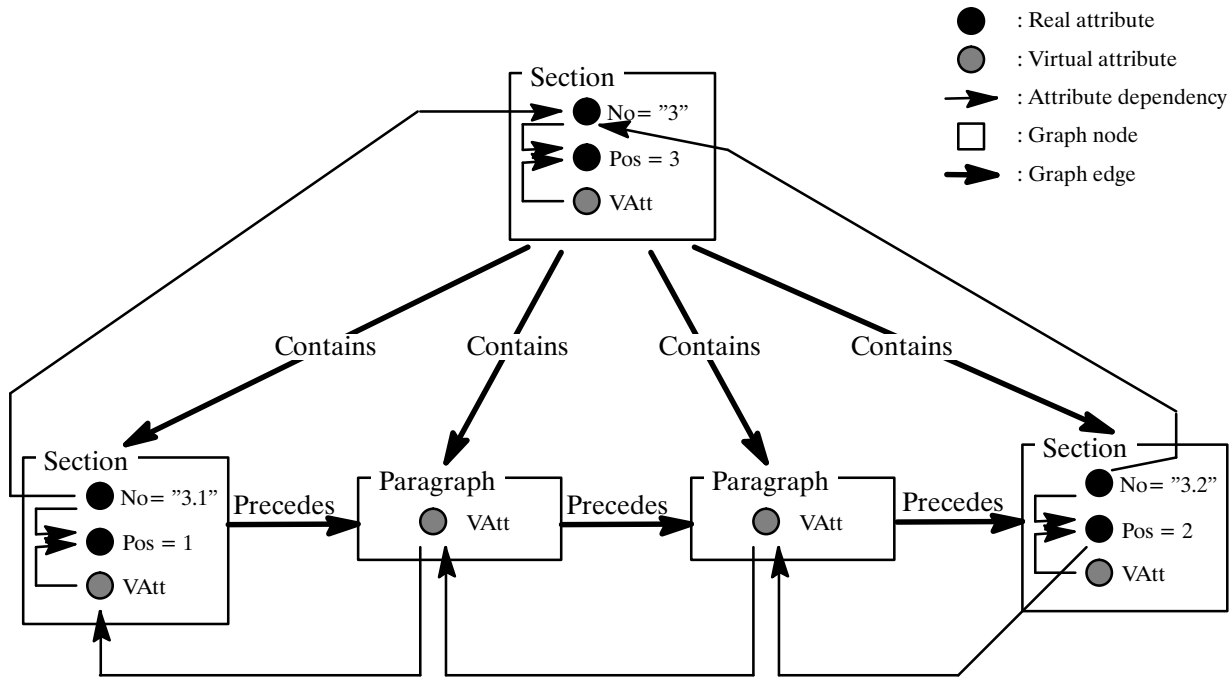

Fig. 7 : Cutout of a documentation's actual attribute dependency graph

(1) the invalidation of the *VAtt* attribute of the first *Paragraph* in response to the deletion of the incoming *Precedes* edge,
(2) the invalidation of the *VAtt* attribute of the next *Paragraph* in response to the invalidation of the *VAtt* attribute of the previous *COMPONENT*,
(3) the invalidation of the *Position* attribute of the next *Section* in response to the invalidation of the *VAtt* attribute of the previous *COMPONENT,*
(4) and finally the invalidation of the *Number* and *VAtt* attributes at the same *Section* in response to the invalidation of its own *Position* attribute.

All these attributes remain invalid as long as their values are not required by the application, i.e. they will be **recomputed on demand** (and propagation processes of subsequent insertions or deletions immediately stop at already invalid attributes). Reading e.g. the *Number* attribute of a *Section* has the side effect of recomputing *Number* and *Position* attributes of all preceding *Sections* and of "validating" *Vatt* attributes of all preceding paragraphs.

Note that the actual attribute dependencies of our documentation graph in fig. 7 do not really exist in the form of additional edges but will be deduced during the propagation process by means of the **static attribute dependency graph**. Fig. 8 displays a cutout of this graph which is a part of the *Documentation* graph scheme and contains the following information (cf. fig. 1):

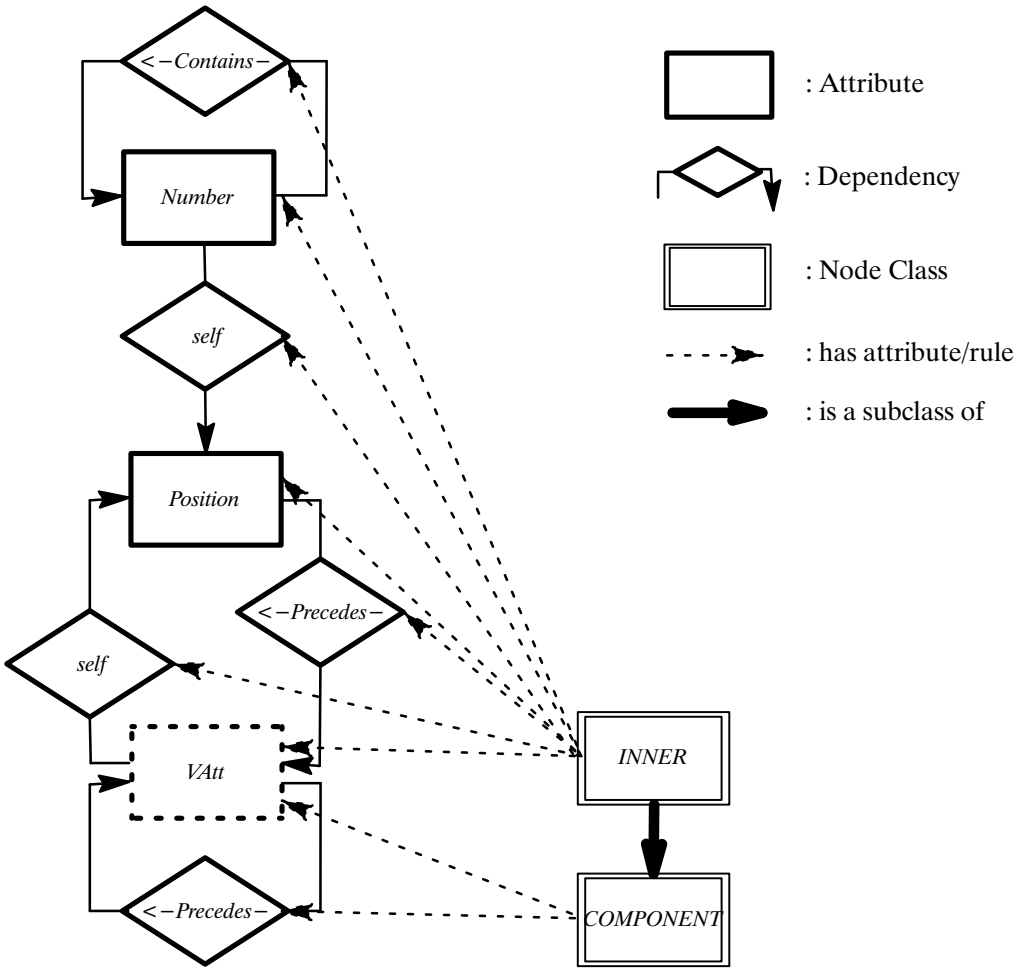

Fig. 8 : Cutout of the documentation example's graph scheme

- The attributes *Number* and *Position* are defined for all nodes being instances of (a type of) the node class *INNER.*
- The value of an *INNER* node's *Number* attribute depends on the value of its own *Position* and the value of the *Number* of that node which is the source of the always existing and always unique incoming *Contains* edge[11].
- The value of an *INNER* node's *Position* attribute depends on the value of a preceding node's (source of an incoming *Precedes* edge) virtual attribute *VAtt*.
- The virtual attribute *VAtt* of a *COMPONENT* node (which is not an *INNER* node) in turn depends on a preceding node's *VAtt.*
- The node class *INNER* is a subclass of the class *COMPONENT* and thus inherits the attribute *VAtt* but not the accompanying attribute dependencies (in this case).
- Instead of this, a redefinition replaces the old attribute dependencies of the node class *COMPONENT* by a new one from an *INNER* node's *VAtt*-attribute to its own *Position.*

11. In general, attribute evaluation functions are either partial or nondeterministic due to the absence of edges or the existence of multiple edges of the same type at one node.

Considering this small example of a graph scheme, the reader might already guess that the manual introduction of additional (virtual) attributes and their attribute dependencies — in order to realize n-context dependencies on top of systems like Cactis /HK 89, HK 89/ which support only 1-context dependencies — would be a very tedious and error-prone task. Therefore, the language PROGRES supports the formulation of complex dependency rules directly, and the PROGRES compiler translates these complex rules into a set of simple rules by means of virtual attributes (cf. /Hö 92/). Furthermore, we believe that the same techniques may be used to implement the incremental computation of derived relations and complex consistency constraints in GRAS.

After this short discussion of the attribute and scheme management layer, we conclude with a final remark about its **location in the GRAS architecture**. The reasons for implementing this part of the GRAS system's functionality beneath its distribution layer and on top of its event management layer are twofold:

- Services provided by the GRAS server process should be as complex as possible in order to **minimize the communication overhead** between servers and clients.
- The necessary supervision of all graph modifications with respect to their consequences for the values of derived attributes may be implemented efficiently by exploiting the GRAS system's **event handling facilities**.

5.4 The Concurrency Control and Distribution Layer

The top-most architectural layer of GRAS handles concurrency control and distribution. As described in section 3, GRAS has a two-layered object model: on the fine-grained level the application operates on nodes and edges, whereas one the coarse-grained level, graphs as structured collections of nodes are manipulated. both concurrency control and distribution refer to the coarse-grained level, i.e. **graphs** rather than nodes **are locked and distributed**.

Concurrency control within a database system is concerned with preserving data integrity while keeping data highly available for access. Its importance is determined by the degree of multi-user access: within a single-user database system, concurrency control is reduced to user identification, validation, and global access right checks. On the other hand, a multi-user database system which allows shared access to data needs sophisticated concurrency control mechanisms down to atomic access operations.

Distribution is realized by a variant of the **client-server approach** used by most distributed database systems /DM 90, GJ 91, De 91/. However, instead of using one centralized database server, GRAS uses a **pool of graph servers**. Each of these graph servers controls and manipulates one or more graphs. Whenever an application requests access to a graph, it is connected to a graph server which performs all of its requests for this graph. This graph server is selected according to following rules:

(1) If the graph is already open due to a request from another application, the corresponding graph server is used.
(2) If a graph server exists which is willing to accept another graph, this server is selected. Acceptance is calculated within the servers based on current host load and number of served graphs.
(3) Else a new graph server is started. The location of this server is determined by the current load of a set of trusted hosts. It is even possible to start more than one server on a host.

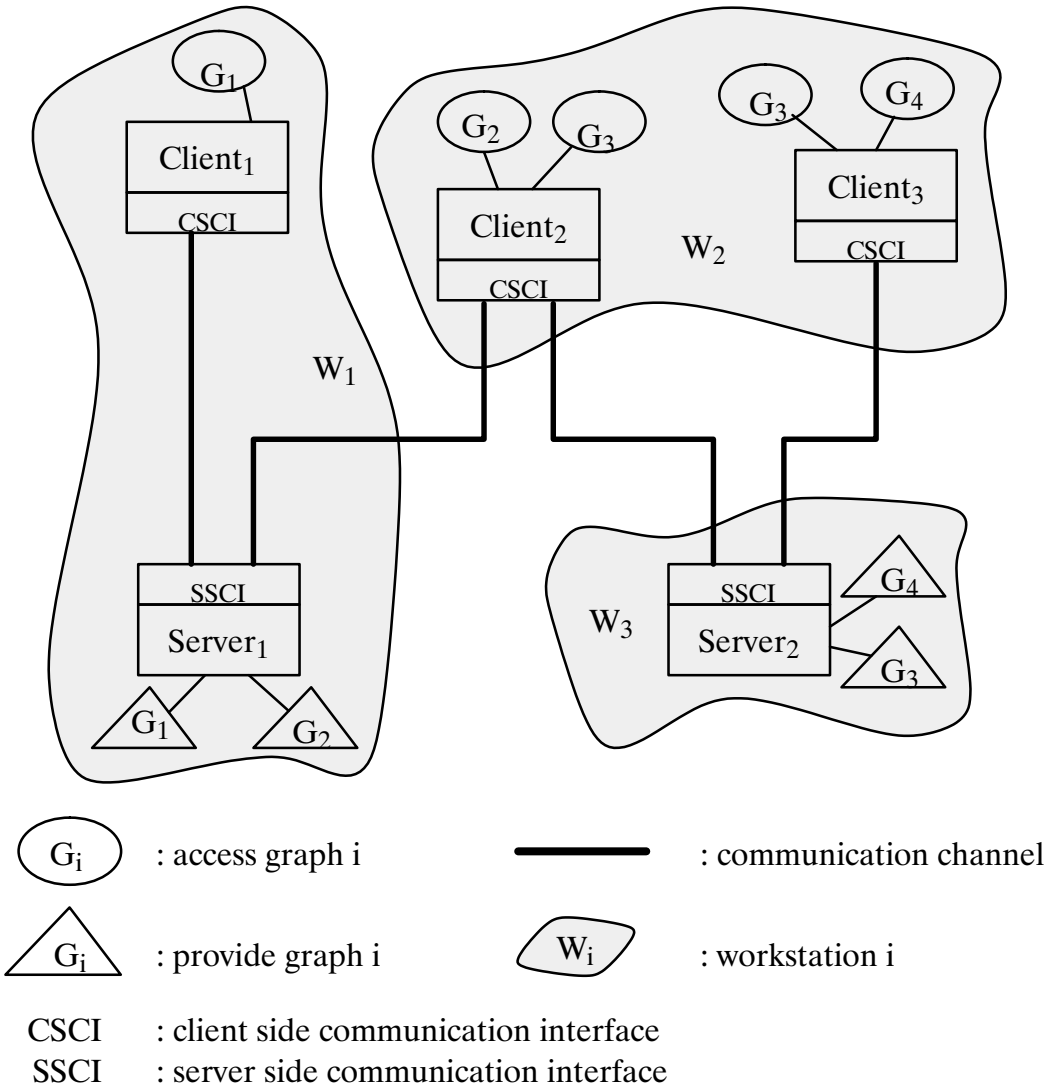

Fig. 9 : A typical distributed GRAS scenario

A special **control server** keeps track of the mapping of open graphs to graph servers /Zo 92/. This control server also serializes open and close requests for graphs to avoid race conflicts and interconnection between different graph servers. Fig. 9 shows a typical scenario with three client and two graph server processes (without the control server). Notice that both *Client2* and *Client3* access the same graph *G3*, and that *Client2* communicates with more than one server.

The interoperability of clients and servers is realized by inserting a **communication layer** consisting of two parts — the client side communication interface (CSCI) and the server side communication interface (SSCI) — which are linked to them. The procedural database interface as described in the previous sections is offered to clients through the CSCI, which uses a **remote procedure call** library to forward calls down to the SSCI of the right server. These interfaces also are responsible for concurrency control and error handling. Furthermore, they handle communication failures and client or server shutdown using follow-up-RPCs and heartbeat protocols.[12]

To organize the concurrent accesses of applications to the same graph, GRAS supports an **access group model**: each application accessing a graph belongs to a group which has an own copy of the accessed graph. Groups are a temporary collection of applications which access the same graph. Groups are created explicitly by one application (which becomes the initial member of this group), potentially joined by other applications (who want to share their view of the graph with other members of that group), and terminate implicitly when the last group member leaves it (by closing the graph).[13]

Graph access groups have a set of attributes which are established at group creation time and can't be changed during their lifetime:
(1) an **interaction granularity attribute** controlling the intervening of operations from different group members,
(2) a **name attribute** which uniquely identifies the group,
(3) a **persistent-modification attribute** controlling whether graph changes survive group termination, and
(4) a **unique group attribute** indicating that there may be no other groups accessing the same graph.

The concurrency of members within a group is controlled by the interaction granularity attribute determining exclusive access units within the time space. GRAS supports three different levels of **interaction granularity**: operational, transactional and sessional access. Access conflicts due to concurrency conflicts are solved by either delaying or aborting the execution of the conflicting operation (the actual reaction is eligible at operation invocation).

- **Operational access** provides for a graph locking during the execution of basic GRAS operations. This is the lowest level of locking over time and, therefore, gives the highest possible rate of concurrency. Allowing even the concurrent execution of these operations is possible (although not yet implemented), but should be transparent for the application.
- The next level, **transactional access**, allows application access only within mutually exclusive top-level transactions. These transactions (like normal transactions within GRAS) can either be aborted or committed, thereby providing for both the grouping of basic opera-

12. For a more detailed discussion of this techniques, please refer to /An 91/.
13. This is analogous to the UNIX© file access model where a file is physically closed when the last accessor (using the same handle) closes the graph.

tions to a complex atomic operation and the basic level of undoing. As the whole graph is locked for the time frame of the top-level transaction, there can't occur lock conflicts during its execution, allowing the application to perform operations without concerning about other applications. This decoupling of other applications can be seen as a great advantage, as applications can operate on shared graphs without knowing or even noticing (besides delay of transaction processing) the presence of other applications. These top-level transactions are also the units of undo and redo (cf. section 5.1), because the SSCI automatically checkpoints the graph after each successful top-level transaction.

- Compare this with the **sessional access** mode which reserves the graph for one group member. This mode virtually provides for a single-user database system, as accessing the graph is allowed for exactly one member of the group and all other members are delayed (or rejected) upon the "graph open" operation. When the active member leaves the group (by closing the graph), the next pending member of this group becomes the active one.

As groups operate on own copies of the graph, interaction between different groups only take place when groups terminate. To avoid update conflicts between access groups of the same graph, at most one group is allowed to replace the original version of the graph by its copy during termination (and thus making its modifications permanent). The creation of such a **persistent-modification group** is denied when there already exists a group with this attribute. As nonpersistent-modification groups have no permanent effect on the graph, more than one of these groups is allowed to exist at the same time. Applications joining such groups address them by their names (supplied at group creation time).

There are at least two reasons for the concept of multiple concurrent nonpersistent−modification groups of a graph:

- The initial graph belonging to a newly created group is an exact copy of the final graph of the last terminated writing group. As lifetimes of reading groups are unrelated, reading groups may still see an old version of the graph while another writing group already established a new permanent graph. If only one reading group were allowed, either the old group would be forced to close or new applications would still see the old version.
- As members of reading groups are allowed to actually modify the corresponding graph copy, the group copies of graphs can diverge even when starting with identical copies.

The **unique group attribute** finally requires that only one group for a graph exists at the same time. It may be used to lock a graph completely, even forbidding the creation of concurrent nonpersistent−modification groups.

In contrast to other distributed (object-oriented) database systems which exchange data (normally in page or object units) between clients and servers (e.g. /BOS 91/ or /LLOW 91/), GRAS doesn't exchange data but **exchanges operation calls** and results, a concept e.g. successfully used in the X Window system /SG 86/. Although data exchange on the physical or logical level as realized by page servers or object servers shows a better performance in general (because

much of the actual work can be shifted from the server to the clients), this approach has been chosen because in most cases, a graph is used either by only one application exclusively or by many applications for frequent but short times:

- For achieving high performance of **exclusively working applications** as e.g. an analyzer checking a large graph, all kind of interprocess communication should be avoided at all, that is the database engine should better be linked directly to the application.
- And when **many applications access a graph** for short updates or queries, e.g. when operating on a project configuration graph, the lock protocols between clients and servers for ensuring consistency in the case of data exchanges are even more expensive than simple remote procedure calls.

The layered architecture of GRAS allows a straightforward realization of the direct linking of one application to a graph server. An implementation is currently underway that even performs an automatic and **transparent shift from direct linking to network coupling** when a graph is accessed by more than one application.

6 Performance Evaluation

The presentation of a database system's functionality and implementation is at least incomplete if it isn't accompanied by some tables and charts which provide the reader with an overall impression of the **system's performance**, and which demonstrate the effects of important design and implementation decisions.

Therefore, we were looking for a **benchmark definition** which is tailored to the special needs of software engineering or hypertext system applications. This benchmark should comprise a set of operations for creating, querying, and modifying complex object structures containing many (cross-) references and attributes of rather different size. Furthermore, it should be well-suited for testing a database system's capability of efficiently executing different kinds of partial match queries (i.e. graph traversals) and range queries, and especially for testing the effectiveness of its incorporated caching and clustering algorithms.

Unfortunately, many **benchmarks** proposed in the literature **do not fulfill these requirements**. For example, all benchmarks — except OO1 — described in /Gr 91/ have been designed for business-oriented and/or relational database system applications, and even the OO1 benchmark for object-oriented database systems /CS 92/ uses a database which contains only one type of relationship (connecting randomly selected objects), and is therefore not well-suited for testing the effects of different clustering algorithms with respect to different types of queries.

6.1 The Hypermodel Benchmark

Finally, we have selected the so-called **hypermodel benchmark** /ABM 90/ which does not propose any delete or structure-oriented update operations but is otherwise well-suited for our purposes. Its underlying data model is an abstract variant of the documentation example of section 2 (cf. fig. 1). A hypermodel database's dominant structure is a totally balanced tree with fan-out 5. Every *NODE* within this tree possesses a couple of integer attributes with one of them playing the role of a unique node key. *INNER* nodes of this tree are sources of an ordered one-to-many-relationship (**1n-rel.**) connecting any node of level n with its five children nodes at level n+1. An additional hierarchical many-to-many-relationship (**mn-rel.**) connects each *INNER* node with five randomly chosen nodes of the next lower level. Finally, we have a third type of attributed many-to-one relationships (**m1a-rel.**). Every *NODE* of the database is the source of exactly one attributed reference to another randomly chosen node (cf. fig. 10).

```
node class NODE;
    intrinsic key UniqueId: integer;
    intrinsic index Hundred: integer;
    intrinsic Ten, Thousand, Million: integer;
    derived index MillionIndex: integer = self.Million div 10000;
end;

node class INNER is a NODE end;
node class LEAF is a NODE end;

edge type 1stChild: INNER [0:1] -> NODE [1:1] ; (* ordered one-to-many-relationship with fan-out 5 *)
 ...
edge type 5thChild: INNER [0:1] -> NODE [1:1] ; (* represented by 5 different one-to-one-relationships *)

edge type Part: INNER [0:n] -> NODE [0:n] ; (* many-to-many-relationship *)

node class LINK; (* representation of attributed many-to-one-relationship *)
    intrinsic OffsetFrom, OffsetTo: integer := nil;
end;
edge type From: NODE [1:1] -> LINK [1:1] ;
edge type To: LINK [0:n] -> NODE [1:1] ;

node class TEXT is a LEAF;
    intrinsic TextAtt: string;
end;
node class FORM is a LEAF;
    intrinsic FormAtt: Bitmap;
end;
```

Fig. 10 : Cutout of the hypermodel benchmark's graph scheme

The *LEAF* nodes of our tree are instances of two node classes. The majority of them are *TEXT* nodes attributed with a sequence of words (between 10 and 100 words). But every 125th node is a so-called *FORM* node which possesses a bitmap of a randomly selected size (between 100 x 100 and 400 x 400). Thus, a test database of depth 7 (with levels 0−6) contains 19531 objects belonging to three different classes, 58591 relationships of three different types, and many attributes with a total size of more than 6.5 MB.

The benchmark itself comprises **operations** for
(1) **creating** the initial test database with clustering along the one-to-many-relationship,
(2) **incremental modifications** of large text- or bitmap-attributes at a number of randomly selected *LEAF* nodes
(3) **range queries** of the form "find all nodes whose integer attribute *Hundred*/*Million* has a value in a randomly chosen interval",
(4) so-called **group lookups** (**reference lookups**), following a specified type of relationship from 50 randomly chosen sources to their sinks (sinks to their sources),
(5) and finally **closure operations**, recursively following a specified type of relationship from 50 randomly chosen nodes on level 3 up to 25 steps in the case of the potentially cyclic m1a-rel. with some of them performing attribute read and write operations.

In order to be able to measure the effects of caching and clustering, every test out of groups (2) through (5) is performed twice. The first run (**cold**) has to start with empty operating system's file buffers and database caches, and the second run (**warm**) with file buffers and caches the state of which is determined by the cold run. In total, the hypermodel test suite consists of about 35 different tests with many of them measuring similar performance characteristics of the underlying database system (at least in the case of our system GRAS). Therefore, the following charts only display the performance results for significant subsets of all benchmark operations. These operations and their abbreviations are explained in more detail in fig. 11. For a more precise description of the whole benchmark the reader is referred to /ABM 90, HPR 90/.

When we started to implement the hypermodel benchmark on top of the GRAS system, we discovered three **major problems** with two of them concerning our data model and one of them concerning the GRAS system's functionality:
(1) Our simple data model of attributed graphs doesn't allow for the definition of **ordered relationships**. Thus, we were forced to introduce 5 different edge types in order to be able to distinguish between the 1st, 2nd, 3rd, . . . child node of an *INNER* node[14].
(2) Our data model doesn't support **attributed edges**. Therefore, attributed relationships are represented by attributed *LINK* nodes which are related to their sources and sinks by *From* and *To* edges, respectively.
(3) The GRAS system supports partial match queries for indexed attributes but not **range queries** for integer attributes. In the case of the *Hundred* attribute it was acceptable to re-

14. For another reasonable solution of this problem see section 1. There we use only one type of *Parent*/*Child* edge, and we use additional *Precedes* edges to represent the order of a *COMPOSITE* node's children.

place a range query of the form “find all nodes whose *Hundred* attribute has value in the range [x . . x+n]” by at most one hundred partial match queries of the form “find all nodes whose *Hundred* attribute has the value x, x+1, . . .”. But in the case of the *Million* attribute we had to introduce an auxiliary indexed attribute whose value is 1/10000 of the value of the *Million* attribute. Thus, we were able to replace every *Million* range query by at most one hundred *MillionIndex* partial match queries.

Abbreviation	Short description of benchmark operation
C. INNER	Creation of all *INNER* nodes of the database.
C. LEAF	Creation of all *LEAF* nodes of the database, i.e. *TEXT* and *FORM* nodes.
C. 1n-rel.	Creation of all 1n-relationships (*Child* edges) of the database.
C. mn-rel.	Creation of all mn-relationships (*Part* edges) of the database.
C. m1a-rel.	Creation of all m1a-relationships *(LINK* nodes, *To* and *From* edges).
Chg. Text c.	Cold search and replace of words in 50 randomly chosen *Text* attributes.
Chg. Text w.	Repetition of “Chg. Text c.” with warm cache and same *Text* attributes.
Gr. 1n c./w.	Group lookup for 1n-rel. (traversal of *Child* edges) starting at 50 nodes.
Gr. m1a c./w.	Group lookup for m1a-rel. (traversal of *From* and *To* edges) at 50 nodes.
1n* c./w.	Closure of Group lookup for 1n-rel. starting at 50 nodes.
mn* c./w.	Closure of Group lookup for mn-rel. starting at 50 nodes.
1n*-S. c./w.	Same as “1n* c./w.” + summing up all visited *Hundred* attributes.
m1a*-S.c./w.	Closure of “Gr. m1a c./w.” + summing up all visited *Offset* attributes.

Fig. 11 : Descriptions and abbreviations of benchmark operations

6.2 Benchmark Results for Different Database Sizes

After this brief description of the hypermodel benchmark, we will now present and analyze the performance results of its implementation on top of the Modula-2 GRAS interface on the following **hardware platform**: a Sun 4/390 in multi-user mode with 32 MB main memory and a 1000 MB CDC IPI 9720 disk under Sun OS 4.1.1.

This implementation does not exploit the GRAS system’s facilities for
- undo/redo of graph modifications,
- event-handling,
- incremental attribute evaluation,
- and communication of multiple clients with multiple server processes.

Therefore, the benchmark application code and the GRAS system code are both belonging to the same address space, and their communication is not based on time-consuming rpc-calls[15]. Furthermore, the *EventManagement* and *SchemeManagement* layers of GRAS are almost inactive during the whole benchmark. Thus, their contribution to the overall processing time is negligible. The *ChangeManagement* layer, on the contrary, spends a considerable amount of time for creating logs, although all tasks based on these logs — like abortion of transactions, recovery from system crashes etc. — are not part of the hypermodel benchmark suite, too.

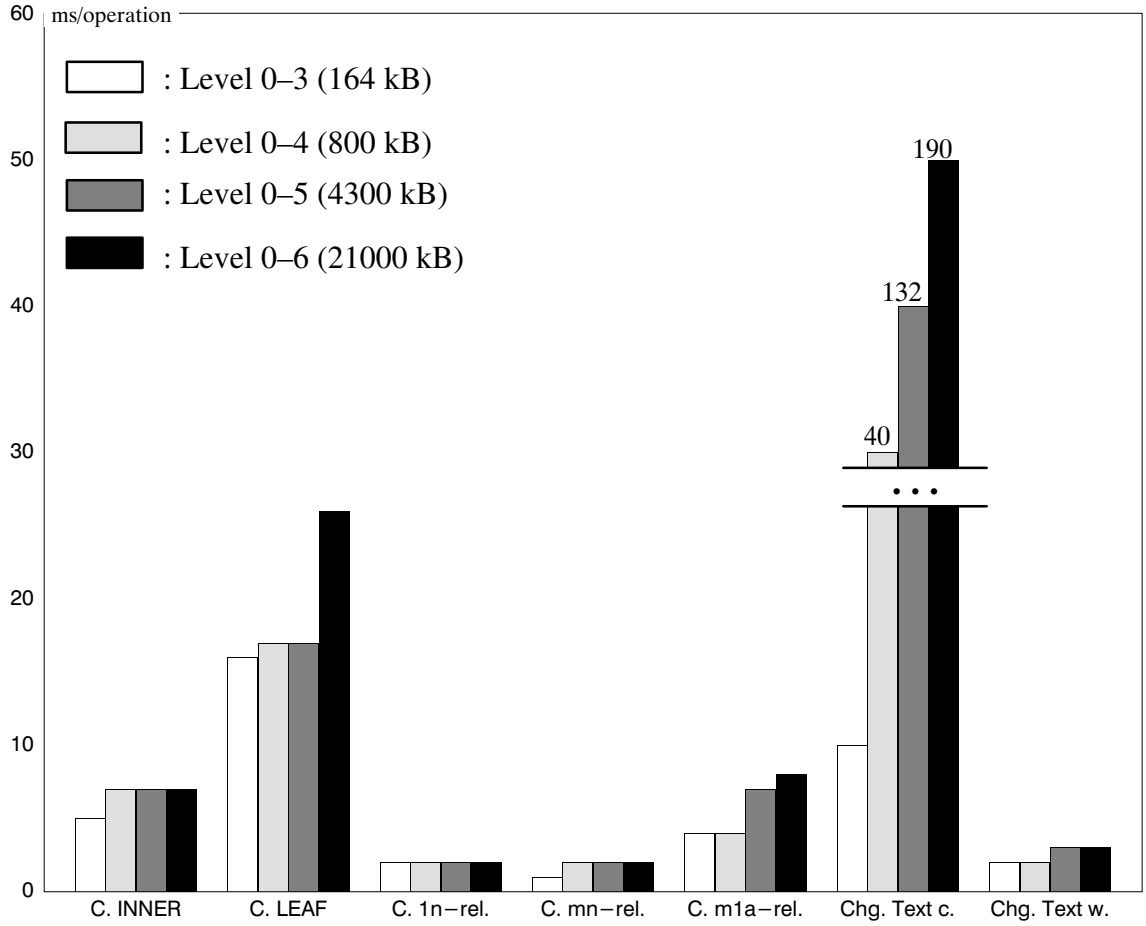

Fig. 12 : Create ops. with varying graph size and fixed cache size (4 MB)

The charts of fig. 12 and fig. 13 present the performance results for **four databases of different sizes** with each database being five times larger than the next smaller one. All reported times in these charts and the chart of the following subsection are given in milliseconds and have been divided by the overall number of affected objects or binary relationships (e.g. the elapsed time for creating all 3906 *INNER* nodes of the level 0–6 database is 3906 * 7 milliseconds; cf. fig 12). Furthermore, the GRAS system's cache has been set to the required upper limit, i.e. to 4 MB.

15. The implementation of a multiple-client version of the hypermodel benchmark is under way.

Fig. 12 presents the **disk storage consumption** of each database and the **database creation times** for all types of nodes and relationships as well as the times for updating randomly selected text attributes. Note that the disk storage consumption of each database is directly proportional to its net size. Furthermore:

- Times reported for creating *INNER* nodes are identical for all databases but the first one. This is due to the fact that the benchmark starts with the creation of all *INNER* nodes and that all *INNER* nodes fit into a 4 MB cache.
- Times reported for creating *LEAF* nodes are always higher than those for *INNER* nodes (especially in the case of the largest test database). This has the following two reasons: (1) *LEAF* nodes possess additional large attributes the creation of which is rather time- and space-consuming, and (2) each *LEAF* node will be placed onto the same page as its already existing *INNER* parent node (cf. clustering strategy *position* in subsection 6.3). Therefore, this operation permanently creates new attribute containing pages and additionally accesses old node pages. This causes many page faults in the case of the largest database.
- Creation times for 1n- and mn-rel. are very low and remain nearly constant for all databases. This is a result of the design decision to store large attributes as well as all relationships (edges) on separate pages. Thus, this operation affects only a small number of edge containing pages. And additional "node existence" consistency checks, which prevent the creation of edges between non-existent nodes, demand only read accesses to a limited number of node containing pages.
- Creation times of m1a-rel. are always higher than those for 1n- and mn-rel., and they are noticeably influenced by a database's size. To understand this fact, you have to remember that the creation of one m1a-rel. requires the creation of one attributed node and two edges (in contrast to the creation of one edge in the case of simple relationships). Therefore, this operation demands additional write accesses to node containing pages. This leads to a significantly greater number of "dirty" page transfers from cache to disk (at least in the case of larger databases).
- The operation "Change *Text* attribute cold" has the most significant increase in time from small databases to large databases. This is a consequence of the fact that all touched attributes are selected randomly. Therefore, the probability for two *Text* attributes belonging to the same page is inverse proportional to a database's size, i.e. growing databases require a growing number of page transfers.
- Finally, the excellent performance results for the operation "Change *Text* attribute warm" are easy to explain. In this case all necessary pages are already stored in the database system's cache and transferring these pages back to the disk is part of a separate "Close database" operation (transaction commit only forces log pages but not data containing pages back to the disk).

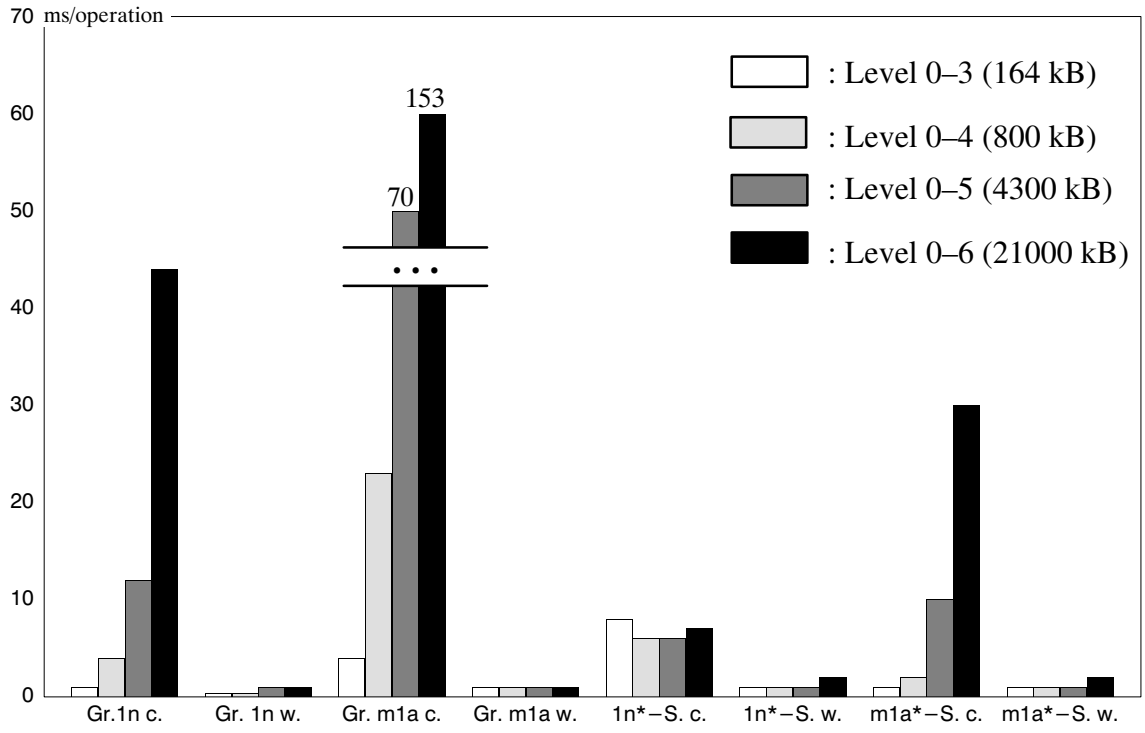

Fig. 13 : Query ops. with varying graph size and fixed cache size (4 MB)

Similar explanations may be given for the **query performance results** displayed in fig. 13:

- The explanations for cold performance results of all group lookup operations, which follow outgoing edges at a small number of randomly selected nodes, are the same as those for the operation "Change *Text* attribute cold".
- And the explanation of all displayed warm execution results is trivial, too: almost all accessed data is already present in the GRAS system's cache. Note that in the case of the closure operation "1n*-S. w." and "m1a*-S. w." and the level 0−6 database only the separation of different kinds of data ensures that almost all accessed data fits into a 4 MB cache (both operations reach about 40% of the nodes of a 21 MB large database).
- Finally, we have to discuss the quite different cold performance results for the closure operations "1n*-S. c." and "m1a*-S. c.". Traversals along 1n-rel. with read accesses to *Hundred* attributes initially are more expensive than traversals along m1a-rel. with read accesses to *Offset* attributes (read accesses to the larger number of *NODE* attributes require more page transfers than read accesses to *LINK* attributes). This initial effect is soon compensated by the fact that traversals along 1n-rel. are compatible with our clustering strategy (strategy *position*, see subsection 6.3). Therefore, the probability for crossing page boundaries is almost independent of the database's size. Traversals along m1a-rel., on the contrary, lead from randomly selected sources to randomly selected sinks and touch a considerably greater number of pages. In this case, the probability for crossing page boundaries is proportional to the overall database size.

To summarize, our hypermodel performance results for growing databases and constant cache size are mainly determined by the following design decisions:

- The order in which objects and 1n-relationships are created is compatible with the system's **clustering strategy** along 1n-relationships.
- Especially 1n- and mn-relationships are **concentrated on a small number of pages**; this is caused by the system's strategy to store different types of informations on different pages.
- The hashing overhead for mapping logical database keys onto physical page addresses does not noticeably increase with growing database size; this is mainly due to the fact that our key creation algorithm guarantees an almost equal **distribution of data** over pages and thus minimizes the number of used pages and the depth of index trees.
- The GRAS paging system, which is responsible for the management of all available cache space, uses a **priority-based LRU-strategy** guaranteeing fast access to all currently used log pages, index pages, and to all frequently and/or recently used data pages.

For similar reasons **varying the cache size** within a wide range while keeping the database size constant causes only minor changes of almost all warm performance results and influences the cold performance results in the same way as varying database sizes do in the presence of a constant cache (cf. fig. 14).

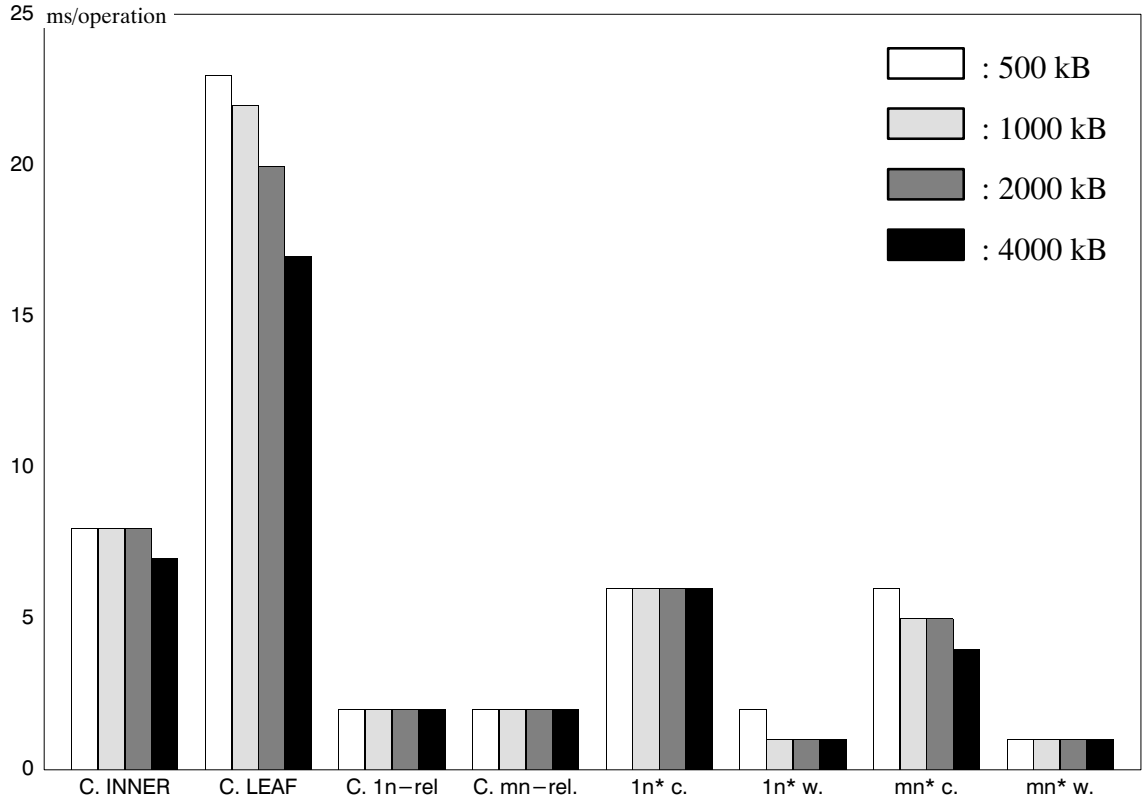

Fig. 14 : Varying cache size and fixed graph size (4.3 MB)

6.3 Benchmark Results for Different Clustering Strategies

The last important parameter, mentioned in the introduction of this section, is the GRAS system's **clustering strategy**. In order to be able to study its influence onto the system's storage consumption and its overall performance, we have executed the hypermodel benchmark with four different node clustering algorithms (cf. subsection 4.2):

(1) **Random distribution** (*random*): Nodes are randomly distributed over pages, so that every page contains about the same number of nodes. The performance results of this method for graph traversals along the 1n-relationship should be considered as upper boundaries for the following clustering algorithms.

(2) **Heuristic page clustering** (*page*): New nodes receive an internal node key such that they are stored on a randomly selected position on the same pages as their already existing parent nodes (with respect to the 1n-rel.) ; therefore, subsequent page splits distribute a parent node's children randomly over different successor pages.

(3) **Heuristic position clustering** (*position*): New nodes receive an internal node key such that they are stored on a carefully selected position on the same page as their already existing parent node. In this case, subsequent page splits normally preserve the physical neighborhood of a parent node's children (as long as this is possible without producing unbalanced index trees or almost empty successor pages).

(4) **Optimal distribution** (*optimal*): Guided by the application's knowledge about the final size and form of the test database it is possible to compute optimal page positions for all nodes (with respect to breadth first graph traversals along the 1n-rel.). Although the GRAS system's interface offers resources for directly determining a node's position, this possibility won't be used by "normal" applications. The main reason for presenting this method here is to find lower boundaries for the above mentioned heuristic algorithms.

The chart of fig. 15 presents the anticipated **results for all node positioning algorithms**. Note that we were forced to use a very small cache (100 kB) in order produce remarkable **execution time differences** between the different clustering algorithms. Considering the increasing execution times for traversing mn-rel. and the decreasing execution times for creating/traversing 1n-rel. from algorithm *random* to algorithm *optimal*, the *position* algorithm is considerably better than the *page* algorithm. For the most important operation, the warm traversal along the 1n-rel., the *position* algorithm even performs as well as the *optimal* solution.

Concerning the **overall database size**, the algorithm *random* outperforms all other algorithms. This might be due to the fact that a random distribution of data over pages is the best way to guarantee balanced index trees and to avoid almost empty pages. Therefore, in a number of cases the strategy *random* needs the smallest number of page transfers and its performance results for creating the initial database and for following randomly chosen paths through the database along mn-rel. are better than those for other algorithms. For similar reasons, the database size and the cold performance results for closure mn-rel. (MN* c.) of the strategy *page* are better than those for the strategies *position* and *optimal*.

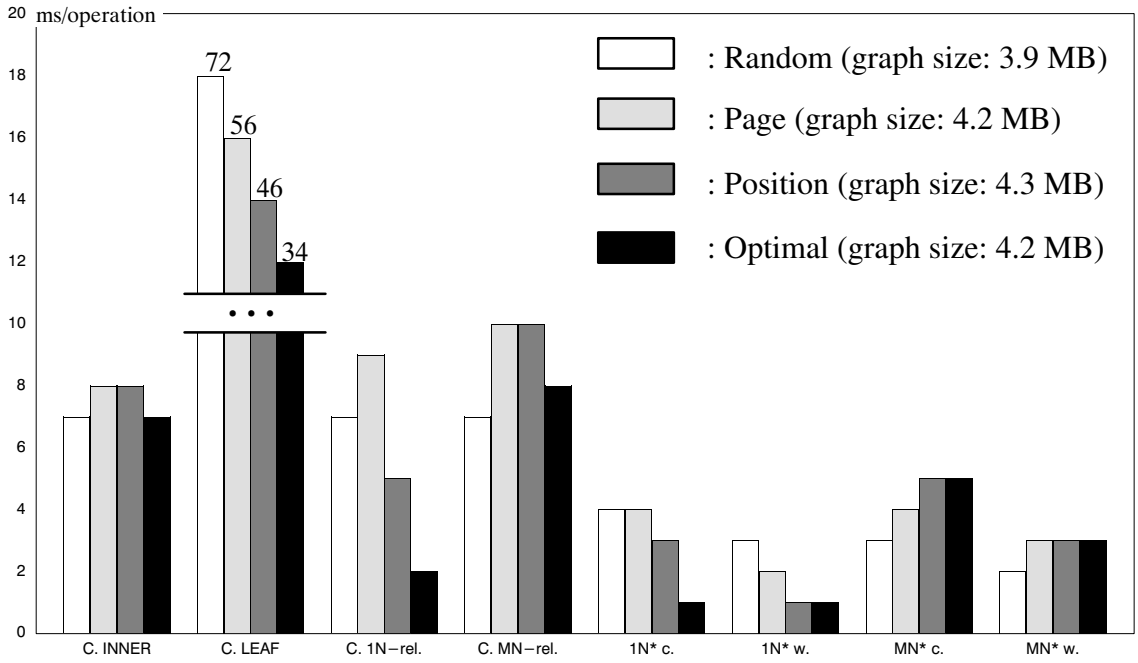

Fig. 15 : Different clustering strategies with fixed cache size (100 kB)

Beside these runtime performance comparisons of clustering strategies, we were also curious to compare **page access patterns** directly. Therefore, we were looking for some **metrics**

- which are easy to compute (based on a number of page access counters),
- which do not depend on the database system's cache or the benchmark's database size,
- and which are proportional to the expansion of a query and its locality.

The **expansion factor** (*EF)* of a query simply measures the overall number of accessed pages. It is low if all accessed data is concentrated on a very small number of pages. The **locality factor** *(LF)* of a query takes its concrete page access pattern into account. It is high, if all accesses to a distinct page are not interleaved with accesses to other pages (counted by the metric "interleaving accesses" *IA)*, and thus the overall number of page faults becomes minimal even in the case of a cache space which is considerably smaller than the expansion of a query.

With *Q* being one of the benchmark's queries (e.g. closure 1n-rel.), *T* being a certain **traversal strategy** for this query (e.g. depth first traversal), *S* being an arbitrary **clustering strategy** (e.g. *random*), and *(OT, OS)* being the best traversal strategy and the best clustering strategy for a given query, our metrics are defined as follows:

$EF(Q, S) =$ no. of touched pages (Q, S) / no. of touched pages (Q, OS)
$LF(Q, T, S) = IA(Q, OT, OS) / IA(Q, T, S)$
$IA(Q, T, S) =$ avg. no. of data accesses between two accesses to same page (Q, T, S)
$(= \infty$, if every portion of data resides on a different page!)

The paper /TN 92/ already suggests to compare different clustering strategies with respect to the expansion as well as the locality of a query. But this paper does not contain a proposal for the definition of the "locality factor", and proposes an "expansion factor" which is difficult to compute. This is due to the fact that the suggested formula depends on a database system's minimal number of bytes/pages necessary for storing a given number of objects, relationships etc. This number is not computable without very intimate knowledge about a database system's internal storage layout (in GRAS, the layout of pages even changes from time to time in order to adapt it to the special needs of a certain application).

Strategy:	*random*	*page*	*position*	*optimal*
EF(1n)*	4.6	1.8	2.9	1.0
LF(1n, DF)*	0	0.17	0.21	0.32
LF(1n, BF)*	0	0.20	0.25	1.0

Fig. 16 : Expansion and locality factors of different clustering strategies

Fig. 16 contains the expansion and the locality factors of the above introduced clustering strategies for a **depth first traversal** (*DF*) and a **breadth first traversal** (*BF*) along the 1n-rel. of a hypermodel database with levels 0−5. Note

- that the expansion, i.e. the total amount of accessed data, is always independent of the selected traversal order (therefore, *EF* is independent of the traversal strategy),
- that the expansion of the strategy *random* is much greater than the expansion of all other strategies, although the strategy *random* creates the smallest database,
- that the strategy *page* places the traversed part of the database on a smaller number of pages than the strategy *position* but is considerably less successfull in minimizing the locality of both a depth first and a breadth first traversal (the dominant factor for the number of page faults if the database cache is much smaller than the whole database),
- that for the *page*, *position*, and *optimal* strategy *"LF(DF) < LF(BF)"* always holds true, because all these strategies attempt to position children nodes nearby their parents and not nearby their neighboring siblings,
- that in the case of the strategy *random* every portion of accessed data resides on a new page ($\rightarrow$ *LF(1n*, DF, random) = LF(1n*, BF, random)* = 0), and
- that the clustering strategy *optimal* is most sensible to changes in access patterns ($\rightarrow$ *LF(1n*, DF, optimal)* $\approx$ *LF(1n*, BF, optimal) / 3*).

Our **conclusions** from these analysis results are the following: Time consuming and sophisticated clustering strategies which depend on very specific assumptions about database access patterns are superior to more heuristically working algorithms with respect to our metrics, but they might degrade considerably in the case of rapidly changing databases with rather different applications on top of it. Furthermore, runtime performance results — especially in the case of warm and not artificially small caches — even show less differences between "dumb" and

"intelligent" clustering strategies (at least in our system GRAS, where object references and node attributes are treated differently and stored on different pages). Therefore, the GRAS system only provides a rather primitive **incrementally and heuristically working clustering algorithm** (the aforementioned algorithm *position*), and it does not spend any efforts on analyzing access patterns of queries or on batch-oriented reorganizations of data clusters (but its interface allows for the implementation of such algorithms on top of it, as e.g. those proposed and/or analyzed in /HK 89, BDK 92, GKK 92, TN 92/).

7 Current Status and Ongoing Work

A first prototype of the GRAS system — described in /BL 85/ — was already realized in 1985. Since this time gradually improving versions of GRAS have been **used at different sites within the software engineering projects** IPSEN /Na 90/, Rigi /MK 88/, CADDY /EHH 89/, MERLIN /PS 92/ and MELMAC /DG 90/. Based on these experiences, almost all parts of the original prototype have been redesigned and reimplemented.

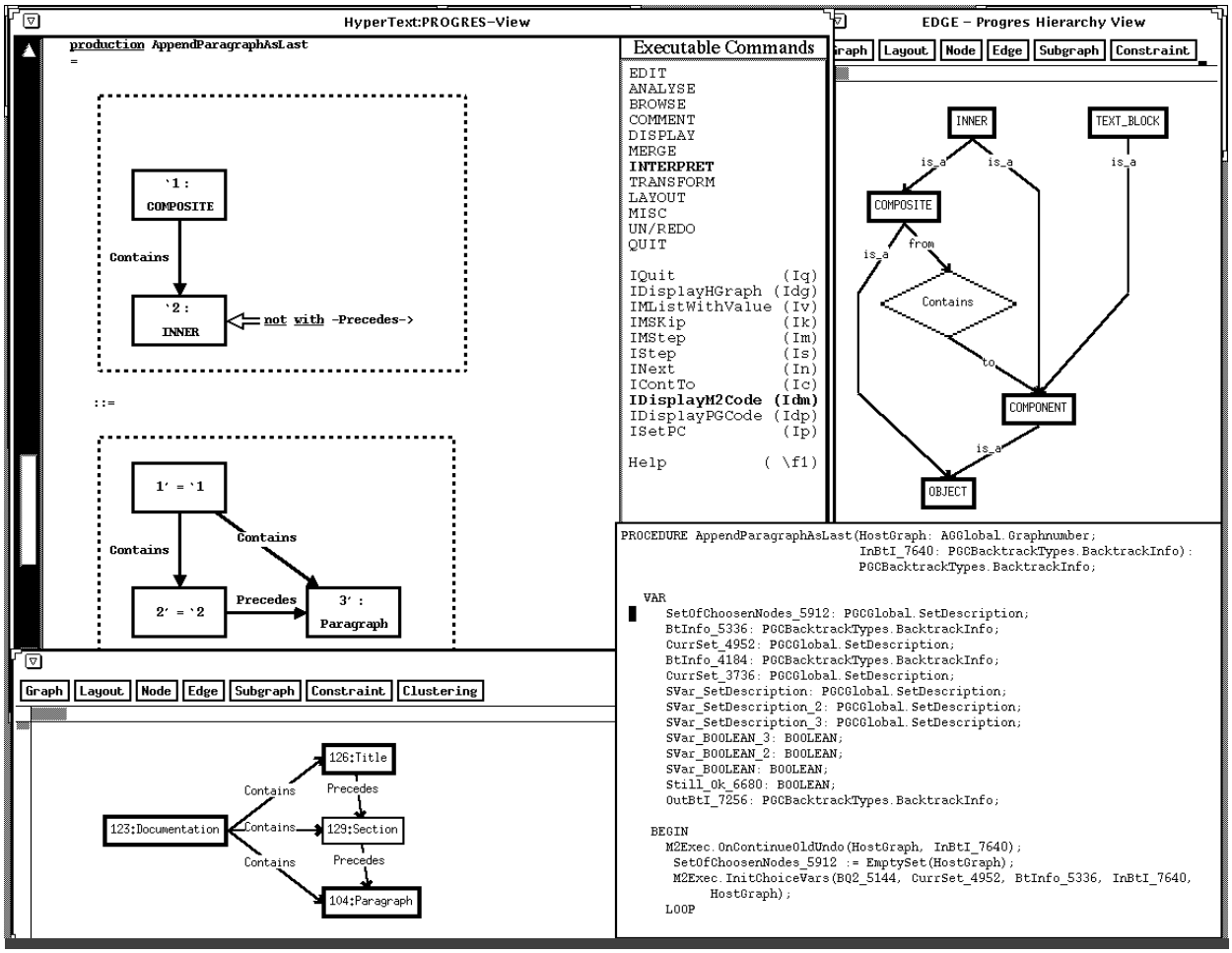

Fig. 17 : Screen dump of the PROGRES/GRAS database development environment

Nowadays a 60,000 lines of code large **stable and efficiently working** GRAS version with multiple-reader/single-writer concurrency control and with interfaces for the programming languages Modula-2 and C is available as **free software**[16]. The implementation of the *Client-ServerDistribution* layer has not yet been released. Up to now, a first prototype of multi-client GRAS shows the feasibility of the selected approach but is subject to further improvements with respect to its functionality and efficiency.

The realization of a **database development environment** based on the language PROGRES comprising a graph browser (a slightly extended version of the system EDGE of the University of Karlsruhe /Ne 91/), a syntax-aided editor for defining graph schemes and graph rewrite rules, an incrementally working type-checker, and an interpreter as well as a cross-compiler (from PROGRES to Modula-2) is nearby completion /NS 91/ and will be available as free software, too. The screen dump of fig. 17 provides the reader with an overall impression of the current status of this 300,000 lines of code large environment. There, we have

- one window displaying a cut-out of the documentation graph scheme of fig. 1 (top/right),
- one window displaying an instance of a very small documentation graph (bottom/left),
- one window displaying the definition of a graph rewrite rule for appending a new *Paragraph* as the last child of a *COMPOSITE* node (top/left),
- and a cut-out of the generated Modula-2 code for this graph rewrite rule (bottom/right).

Last but not least, an **object-oriented Modula−3 version** of the system GRAS with enhanced event handling capabilities and a port to the new **ECMA-standard platform PCTE** /ECMA 90/ for software engineering environments are under development.

Acknowledgements

The development of the GRAS system has been supported by grants from the “German Research Council” (DFG Na 134/2−1, 2−2 and DFG Na 134/4−1, 4−2), the “Stiftung Volkswagenwerk” (VW I.61512), and the “European Community” (EspritProject 5409).

References

/ABM 90/ Anderson T.L., Berre A.J., Mallison M. et al.: *The Hypermodel Benchmark*, in: Bancilhon, Thanos, Tsichritzis (Eds.): Advances in Database Technology—EDBT ’90, LNCS 416, Springer Press (1990), 317−331

/ACR 87/ Alpern B., Carle A., Rosen B. et al.: *Incremental Evaluation of Attributed Graphs*, Technical Report CS-87-29, Brown University (1987)

/ACR 89/ Alpern B., Carle A., Rosen B. et al.: *Graph Attribution as a Specification Paradigm*, in: /He 89/, 121−129

16. Under the GNU licence conditions of the “Free Software Foundation” via anonymous ftp from: ftp.informatik.rwth−aachen.de [137.226.112.172] in /pub/unix/GRAS<version_no>.

/ACS 84/ Archer J.E., Conway R., Schneider F.B.: *User Recovery and Reversal in Interactive Systems*, ACM Transactions on Programming Languages and Systems, vol. 6−1, ACM Press (1984), 1−19

/Al 88/ Alderson, A.: *A Space-Efficient Technique for Recording Versions of Data*, Software Engineering Journal, IEE & BCS Joint Publ. (1988), 240−246

/An 91/ Andrews, G.R.: *Concurrent Programming: Principles and Practice*, The Benjamin/Cummings Publishing Company, Inc. (1991)

/AP 92/ Analyti A., Pramanik S.: *Fast Search in Main Memory Databases*, in: /St 92/, 215−224

/BC 72/ Bayer R., McCreight E.M.: *Organization and Maintenance of Large Ordered Indices*, Acta Informatica, vol.1, Springer Press (1972), 173−189

/BDK 92/ Bancilhon F., Delobel C., Kanellakis P. (eds.): *Building an Object-Oriented Database System*, Morgan Kaufmann Publ. (1992)

/Be 87/ Bernstein P.: *Database System Support for Software Engineering*, Proc. of the 9th Int. Conf. on Software Engineering, IEEE Computer Society Press (1987), 166−178

/BL 85/ Brandes T., Lewerentz C.: *GRAS : A Non-Standard Database System within a Software Development Environement*, Workshop for Software Engineering Environments for Programming in the Large, Harwichport, Massachusetts (1985), 113−121

/BM 91/ Beeri, C., Milo, T.: *A Model for Active Object Oriented Database*, Proc. 17th Int. Conf. on Very Large Data Bases, IEEE Computer Society Press (1991), 337−349

/BOS 91/ Butterworth P., Otis A., Stein J.: *The GemStone Object Dabase Management System*, Communications of the ACM, vol. 34−10, ACM Press (1991), 64−77

/Br 89/ Broekmanns M.: *Change Management in a Non-Standard Database System* (in German), Diploma Thesis, Technical University of Aachen, Germany (1992)

/CD 86/ Coulouris G.F., Dollimore J.: *Distributed Systems—Concepts and Design*, Addison-Wesley Publishing Company (1986)

/CDR 89/ Carey M., DeWitt D.J., Richardson J.E., Shekita, E.J.: *Storage Management for Objects in EXODUS*, in: Kim, Lochovsky (Eds.): Object-Oriented Concepts, Databases, and Applications, ACM Press, Frontier Series (1989), 341−369

/Ch 76/ Chen P.P.: *The Entity-Relationship Model—Toward a Unified View of Data*, ACM Transactions on Database Systems, vol. 1−1, ACM Press (1976), 9−36

/Ch 89/ Chakravarthy S.: *Rule Management and Evaluation: An active DBMS prospective*, ACM SIGMOD, vol. 18−3, ACM Press (1989), 20−28

/Co 89/ Cohen E.S., Sari D.A., Genecker R., Hasling W.M., Schwanke R.W., Wagner M.C.: *Version Management in Gypsy*, in: /He 89/, 201−215

/CS 92/ Cattell R.G.G., Skeen J.: *Object Operations Benchmark*, ACM Transactions on Database Systems, vol. 17−1, ACM Press (1992), 1−31

/Da 88/ Dayal U. et al: *The HiPAC Project: Combining Active Databases and Timing Constraints*, ACM SIGMOD, vol. 17−1, ACM Press (1988), 51−70

/De 91/ Deux O.: *The O_2 System*, Communications of the ACM, vol. 34−10, ACM Press (1991), 34−48

/DG 90/ Deiters W., Gruhn V.: *Managing Software Processes in the Environment MELMAC*, Proc. of the 4th Int. Symposium on Practical Software Development Environments, SIGSOFT Notes, vol. 15−6, ACM Press (1990), 193−205

/DGL 86/ Dittrich K.R., Gotthard W., Lockemann P.C.: *DAMOKLES, A Database System for Software Engineering Environments*, Proc. of the Int. Workshop on Advanced Programming Environments 1986, LNCS 244, Springer Press (1986), 353−371

/DKM 86/ Dittrich, K.R., Kotz, A.M., Mülle, J.A.: *An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases*, ACM SIGMOD vol. 15−3, ACM Press (1986), 22−36

/DM 90/ DeWitt D.J., Maier D.: *A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems*, Proc. 16th Int. Conf. on Very Large Data Bases, IEEE Computer Society Press (1990), 107−121

/ECMA 90/ European Computer Manufacturers Association: *ECMA PCTE Standard*, ECMA−149 (1990)

/ED 88/ Enbody R., Du H.: *Dynamic Hashing Schemes*, ACM Computing Surveys, vol. 20−2, ACM Press (1988), 85−113

/EHH 89/ Engels G., Hohenstein U., Hülsmann U. et al.: *CADDY—Computer Aided Design of Non-Standard Databases*, in: Madhavji, Schäfer, Weber (Eds.): Proc. of the 1st Int. Conf. on System Development Environments & Factories, Pitman Press (1989), 151−158

/ELN 92/ Engels G., Lewerentz, L., Nagl M. et al.: *Building Integrated Software Development Environments Part I: Tool Specification*, ACM Transactions on Software Engineering and Methodology, vol 1−2, ACM Press (1992), 135−167

/Fl 83/ Flajolet Ph.: *On the Performance Evaluation of the Extendible Hashing and Trie Searching*, Acta Informatica, vol. 20, Springer Press (1983), 345−367

/FM 86/ Fraser C.W., Myers E.W.: *An Editor for Revision Control*, ACM Transactions on Programming Languages and Systems, vol. 9−2, ACM Press (1986), 277−295

/FNPS 79/ Fagin R., Nievergelt J., Pippinger N., Strong H.R.: *Extendible Hashing: A Fast Access Method for Dynamic Files*, ACM Transactions on Database Systems, vol. 4−3, ACM Press (1979), 315−344

/GB 82/ Goldstein I.P., Bobrow D.G.: *A Layered Approach to Software Design*, Technical Report, XEROX PARC, Palo Alto (1982)

/GKK 92/ Gerlhof C., Kemper A., Kilger Ch., Moerkotte G.: *Clustering in Object Bases*, Technical Report 6/92, Fakultät für Informatik, University of Karlsruhe, Germany (1992)

/GJ 91/ Gehani N., Jagadish H.V.: *Ode as an active Database: Constraints and Triggers*, Proc. 17th Int. Conf. on Very Large Data Bases, IEEE Computer Society Press (1991), 327−336

/Gr 91/ Gray J. (ed.): *The Benchmark Handbook*, Morgan Kaufmann Publ. (1991)

/GRD 90/ Giavitto J., Rosuel G., Devarenne A., Mauboussin A.: *Design Decisions for the Incremental Adage Framework*, Proc. of the 12th Int. Conference on Software Engineering, IEEE Computer Society Press (1990), 86−95

/He 89/ Henderson P. (ed.): *Proc. ACM 3rd Symposium on Practical Software Development Environments*, ACM SIGPLAN Notices, vol. 24−2, ACM Press (1989)

/HPR 90/ Hormann H., Platz D., Roschewski M. et al.: *The Hypermodel Benchmark*, *Description, Execution, and Results*, SWT Memo Nr. 53, Lehrstuhl für Software-Technologie, University of Dortmund, Germany (1990)

/HK 88/ Hudson S.E., King R.: *The Cactis Project: Database Support for Software Environments*, IEEE Transactions on Software Engineering, vol. 14−6, IEEE Computer Society Press (1988), 709−719

/HK 89/ Hudson S.E., King R.: *Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System*, ACM Transactions on Database Systems, vol. 14−3, ACM Press (1989), 291−321

/HN 86/ Habermann N., Notkin D. : *Gandalf : Software Development Environments*, IEEE Transactions on Software Engineering, vol. 12−2, IEEE Computer Society Press (1986), 1117−1127

/Hö 92/ Höfer F.: *Incremental Evaluation of Attributed Graphs—Extensions and Realization* (in German), Diploma Thesis, Technical University of Aachen, Germany (1992)

/Hu 87/ Hudson S.E.: *Incremental Attribute Evaluation: An Algorithm for Lazy Evaluation in Graphs*, Technical Report 87−20, University of Arizona (1987)

/IDL 87/ *Special Issue on the Interface Description Language IDL*, ACM SIGPLAN Notices, vol. 22−11, ACM Press (1987)

/Ka 85/ Kawagoe K.: *Modified Dynamic Hashing*, in: Navathe (Ed.): Proc. of the ACM SIGMOD 1985 Int. Conf. on Management of Data, ACM Press (1985), 201−213

/KD 92/ Keßler U., Dadam P.: *Evaluation of Complex Queries on Hierarchically Structured Objects by Path Indexes*, in: Appelrath (Ed.): Proc. Datenbanksysteme in Büro, Technik und Wissenschaft, IFB 270, Springer Press (1992), 218−237

/Ki 91/ Kilian M.F.: *A Note on Type Composition and Reusability*, OOPS Messenger, vol. 2−3, ACM Press (1991), 24−32

/KM 92/ Kemper A., Moerkotte G.: *Access Support Relations: An Indexing Method for Object Bases*, Information Systems, vol. 17−2, Pergamon Press Ltd (1992), 117−145

/La 80/ Larson P.A.: *Linear hashing with Parital Expansions,* Proc. of the 6th Int. Conf. on Very Large Databases, IEEE Computer Society Press (1980), 224−232

/La 88/ Larson P.A.: *Dynamic Hash Tables*, Communications of the ACM, vol. 31−4, ACM Press (1988), 446−457

/LC 86/ Lehmann T.J., Carey M.: *A Study of Index Structures for Main Memory Database Management Systems*, Proc. of the 12th Int. Conf. on Very Large Databases, IEEE Computer Society Press (1986), 294−303

/Li 78/ Litwin W.: *Virtual Hashing: Dynamically Changing Hashing*, Proc.of the 4th Conf. on Very Large Databases, IEEE Computer Society Press (1978), 517−523

/LLOW 91/ Lamb O., Landis G., Orenstein, J. Weinreb D. : *The Objectstore Database System*, Communications of the ACM, vol. 34−10, ACM Press (1991), 50−63

/LS 88/ Lewerentz C., Schürr A.: *GRAS, a Management System for Graph-Like Documents*, in: Beeri, Schmidt, Dayal (Eds.): Proc. of the 3rd Int. Conf. on Data and Knowledge Bases, Morgan Kaufmann Publ. (1988), 19−31

/MK 88/ Müller H.A., Klashinsky K.: *Rigi—A System for Programming-in-the-large,* Proc. 10th Int. Conf. on Software Engineering, IEEE Computer Society Press (1988), 80−85

/MR 86/ Meyer A.R., Reinhold M.B.: *'Type' is not a type*, in: Proc. of the 13th ACM Symp. on Principles of Programming Languages, ACM Press (1986), 287−295

/Na 90/ Nagl, M.: *Characterization of the IPSEN Project*, in: Madhavji, Schäfer, Weber (eds.): Proc. of the 1st Int. Conf. on Systems Development Environments & Factories 1989, Pitman Press (1990), 141−150

/Ne 91/ Newbery Paulisch F.: *The Design of an Extendible Graph Editor*, Dissertation, University of Karlsruhe, Germany (1991)

/NS 91/ Nagl M., Schürr A.: *A Specification Environment for Graph Grammars*, in: Ehrig, Kreowski, Rozenberg (Eds.): Graph Grammars and Their Application to Computer Science, Proc. of the 4th Int. Workshop 1990, LNCS 532, Springer Press (1991), 599−609

/PS 92/ Peuschel B., Schäfer W.: *Concepts and Implementation of a Rule-Based Process Engine*, Proc. of the 14th Int. Conf. on Software Engineering, IEEE Computer Society Press (1992), 262−279

/Re 91/ Reichenberger C.: *Delta Storage for Arbitrary Non-Text Files*, Proc. of the 3rd Int. Workshop on Software Configuration Management 1991, ACM Press (1991), 144−152

/Ro 75/ Rochkind M.J.: *The Source Code Control System*, IEEE Transactions on Software Engineering, vol. 1−4, IEEE Computer Society Press (1975), 364−370

/RT 88/ Reps T., Teitelbaum T.: *The Synthesizer Generator*, Springer Press (1988)

/Sc 89/ Schürr A.: *Introduction to PROGRES, an Attribute Graph Grammar Based Specification Language*, in: Nagl (Ed.) : Proc. of the WG '89 Workshop on Graph-Theoretic Concepts in Computer Science, LNCS 411, Springer Press (1989), 151−165

/Sc 91a/ Schürr A.: *PROGRES, A VHL-Language Based on Graph Grammars*, in: Ehrig, Kreowski, Rozenberg (Eds.): Graph Grammars and Their Application to Computer Science, Proc. of the 4th Int. Workshop 1990, LNCS 532, Springer Press (1991), 641−659

/Sc 91b/ Schürr A.: *Operational Specification with Programmed Graph Rewriting Systems: Formal Definitions, Applications, and Tools* (in German), Deutscher Universitäts Verlag (1991)

/SG 86/ Scheifler R., Gettys J.: *The X Window System*, ACM Transactions on Graphics, vol. 5−2, ACM Press (1986), 79−109

/St 92/ Stonebraker (Ed.): Proc. of the ACM SIGMOD Int. Conf. on Management of Data, vol. 21−2, ACM Press (1992),

/SZ 91/ Schürr A., Zündorf A.: *Non-Deterministic Control Structures for Graph Rewriting Systems*, in: Schmidt, Berghammer (Eds.): Proc. of the WG '91 Workshop on Graph-Theoretic Concepts in Computer Science, LNCS 570, Springer Press (1991), 48−62

/SZd 87/ Smith K.E., Zdonik St.B.: *Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems*, Proc. of OOPSLA '87, Object-Oriented Programming Systemes, Languages and Applications, SIGPLAN Notices, vol. 22−12, ACM Press (1987), 452−465

/ThN 92/ Thomas I., Nejmeh B.: *Definitions of Tool Integration for Environments*, IEEE Software, IEEE Computer Society Press (1992), 29−35

/Ti 85/ Tichy W.F.: *RCS—A System for Version Control*, Software: Practice and Experience, vol. 15−7, John Wiley & Sons (1985), 637−654

/TN 92/ Tsangaris M.M., Naughton J.F.: *On the Performance of Object Clustering Techniques*, in: /St 92/, 144−153

/We 89/ Westfechtel B.: *Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control*, Technical Report AIB 89−8, Technical University of Aachen, Germany (1989)

/We 91/ Westfechtel B.: *Revision and Consistency Control in an Integrated Software Development Environment* (in German), Informatik Fachberichte 280, Springer Press (1991)

/Zo 92/ Zohren St.: *The GRAS Server* (in German), Diploma Thesis, Technical University of Aachen, Germany (1992)