The Term Retrieval Abstract Machine

Michael Ley *
University of Trier, FB IV – Informatik
P.O. Box 3825, W-5500 Trier, Germany, ley@uni-trier.dbp.de

Scans through large collections of complex objects often cannot be avoided. Even if sophisticated indexing mechanisms are provided, it may be necessary to evaluate simple predicates against data stored on disk for filtering. For traditional record oriented data models i/o and buffer management are the main bottlenecks for this operation, the interpretation of data structures is straightforward and usually not an important cost factor. For heterogeneously shaped complex objects it may become a dominant cost factor.

In this paper we demonstrate a technique to make data structure traversal inside of complex objects much cheaper than naive interpretation. We compile navigation necessary to evaluate condition predicates and physical schema information into a program to be executed by a specialized abstract machine. Our approach is demonstrated for the Feature Term Data Model (FTDM), but the technique is applicable to many other complex data models. Main parts of this paper are dedicated to the method we used to design the Term Retrieval Abstract Machine (TRAM) architecture by partial evaluation of a tuned interpreter.

1 Introduction

This is a paper about two standard themes of database system implementation: (1) Pushing critical operations down to the next architectural layer, and (2) avoiding unnecessary overhead by compilation and planning.

Perhaps unusual are the layer we deal with and the method we employ. Compilation and optimization on the operator level is a main theme in the database community. The design of efficient algorithms to implement operators and the mapping of complex objects to disk pages are the focus of many research papers (e.g. [19, 8, 32]).

The interpretation of these storage structures often is considered as minor detail of implementation and rarely described in literature [13]. For the conventional relational data model this abstinence may be justified, but for complex object data models interpretation and navigation inside of complex storage structures may become worth to consider because of their impact on system performance [32].

1992 ACM SIGMOD - 6/92/CA, USA
• 1992 ACM 0-89791-522-4/92/0005/0154...$1.50

*The work reported here was carried out within the LILOG-project of IBM Germany.

For the first prototype version of our complex feature term database system LILOG-DB [26, 3, 25, 28] we observed cpu boundness even for simple scan operations. A detailed analysis revealed that the database system kernel spent too much time for main memory management and for the evaluation of simple test predicates. To improve performance we proceeded in three steps:

1. First we replaced the standard C heap allocator by a main memory management based on object lifetimes. The result was an average 40% performance improvement of the entire system. The details of the new memory allocator are reported in [24].
2. The second step was to tune the predicate evaluation interpreter. Predicate evaluation was pushed down into the component which understands the layout of disk records. The performance gain in the interpreter was up to 100%.
3. The third step was to replace this interpreter by a compiled version. The target of compilation is the *Term Retrieval Abstract Machine* (TRAM). TRAM is implemented by an interpreter written in C, i.e. we only shifted the level of interpretation. Nevertheless we achieved a performance gain of up to 40% compared with the tuned interpreter by carefully designing the TRAM architecture.

The main part of this paper is a detailed description of the techniques we used in this third step.

The remainder of this paper is organized as follows. In Section 2 we briefly sketch some facts about LILOG-DB and the Feature Term Data Model (FTDM) to explain the context under which the work reported here was done. Term representation and internal schemas used in LILOG-DB are discussed in Section 3. The most important operations on disk resident data structures are the evaluation of conditions and the translation of qualifying terms to their in-memory representation. These are the tasks of the *Retrieval Manager*. In Section 4 we sketch the naive implementation of the Retrieval Manager and the tuning of it ("second step"). The prerequisite of the third step is a detailed analysis of this interpretative Retrieval Manager by partial evaluation to develop the TRAM architecture. The analysis is illustrated in Section 5, the resulting architecture is described in Section 6. The operations of the TRAM compiler, which translates queries and internal schemas to TRAM code, are the subject of Section 7. We con-

Note: This paper was shortened to meet the length limit by the Program Committee because the author was unavailable during the paper revision phase.

clude the paper with a performance evaluation section (8) and a section about perspectives for further elaboration of our concepts (9).

2 The LILOG-DB System

LILOG-DB is a deductive complex-object database system which was developed to support the natural language system LILOG [15] and to explore the abilities of logic-based database technology.

The *Feature Term Data Model* (FTDM) supported by LILOG-DB and its deductive user interface are direct outcomes of the analysis of the LILOG requirements. The complex objects stored in LILOG-DB are Prolog-like terms composed of atoms (integers, reals, symbols), functors, lists, sets and feature tuples. The feature term

```
<| name: 'Vancouver', hotels: [
   <| name:     'Granville Island Hotel',
      telephone:'683-7373'              |>,
   <| name:     'Hotel Vancouver',
      roof:     'green'                 |>,
   <| name:     'Pan Pacific Vancouver Hotel',
      address:  '999 Canada Place'      |> ] |>.
```

is an example we use for illustration in this section. In LILOG-DB collections of such feature terms are stored in persistent *feature term sets* (ft-sets). The members of an ft-set can be characterized by a sort. LILOG supports a very powerful language to specify sorts by syntactic descriptions, semantic constraints and multiple inheritance of characteristics from supersorts in the sort lattice [29, 3]. The main use of sorts is semantic processing in the logic programming and query processing levels of LILOG-DB, but the syntactic part of sort specifications is exploited by the kernel of LILOG-DB to achieve a compact object representation on disk [25].

This kernel, called the *Term Manager* (or *Fact Manager* in former papers [26, 27]), has a single ft-set scan interface for retrieval. The power of this interface is described by the γ-operator of EFTA (Extended Feature Term Algebra [28]), the algebra we translate DATALOG-like queries into.

The γ-operator is a generalization of the relational selection and projection operators for complex feature terms. The result of an application $\gamma_{cc}F$ is defined to be the set of feature terms we get if we evaluate the conditional constructor cc for all members of the feature term set F and eliminate duplicates. A conditional constructor is a (nested) if-then-else expression like

```
if (hotels.1.□ has telephone) then
    hotel(hotels.1.name.□,hotels.1.telephone.□)
else
    hotel(hotels.1.name.□,'unknown').
```

Conditions may be composed from the usual comparison operators, a set membership test, a sort-check, an arity test and the test whether a certain feature is specified in a tuple ('*has*'). The latter tests make sense, because unlike relational databases we do not force the members of an ft-set to be of identical shape. The navigation inside complex terms is expressed by *paths*. They are used inside of conditions and inside of constructors. The '*hotel*(...)' constructor in our example is a functor constructor, EFTA provides constructors to build lists, tuples, sets and atoms, too.

3 Term Representation

There are two principal methods to represent complex objects, like our feature terms, on disk: Either we use the same representation in memory and on disk to avoid "pointer swizzling", or we employ a 2-level storage model to gain more flexibility. We do not repeat the pros and cons of both approaches here, most arguments of this discussion can be found in the reports of the OOPSLA OODBS workshops [33, 16]. For us, the main reasons to use the 2-level storage approach were that the representation of terms given by our application, which is built in several logic programming environments, makes a one-level storage impossible and that we want to exploit schema information from sorts to achieve a more compact disk representation.

A peculiar property of our application domain is that schema information we have about the collections of terms to be stored in the database often is incomplete. For example, we may know that the terms of a collection are tuples, which usually have some known features (attributes), but they are allowed to have any number of other features not known in advance. The underlying reason for this situation is the style of modeling and processing incomplete knowledge in unification based natural language systems like LILOG [15]. Many items are represented by the structure of terms and not by values, i.e. processing often is done by meta-reasoning. E.g., a sort specification [29]

```
sort city =
    tuple <| name: symbol, hotels: listof hotel | >
    with □ has name
```

means: All tuples of sort *city* must have a *name*-feature with a symbol (string) value. *city* tuples may have a *hotels*-feature. If this feature is present, it must be a list. There may be any other features with arbitrary complex values.

The challenge we had to meet is to find a disk representation which allows to exploit parts of the sort information from the semantic level for compactness on the physical level.

In most object-oriented database systems (e.g. [5, 9, 10, 18, 19, 20]) and in NF^2 database systems (e.g. [32, 8]) complex objects are decomposed for disk representation. In most data models complex objects are trees, DAGs, or some more general class of graphs. The nodes of these graphs usually have known formats, they are classical records or nested records. The edges are used to link nodes and to construct complex objects of any shape and size. Decomposition is done along edges.

Nodes are represented by disk records, edges by appropriate physical or logical addresses. Because it would be very expensive to reconstruct large objects from records scattered on several files, sophisticated clustering techniques to map logical adjacency of nodes to physical adjacency of records have been developed [2, 5, 10, 34, 6].

The problem is that the logical nodes of our feature terms have no fixed formats, but are often very heterogeneous. We know two techniques to deal with this kind of variability:

The first technique we considered is to introduce a further level of decomposition. I.e. the logical nodes themselves are mapped to several physical records, which have standard formats.

The other approach is to represent a logical node by a single record in a self-identifying format [13].

We decided for the second technique, because only this gives us the chance to exploit incomplete schema information to compress disk representation and to minimize i/o. The idea is to use a hybrid of standard record formats and self-identifying formats. If we know a complete schema of a node, our method degenerates to the standard technique described above. With the first technique, i.e. with a further level of decomposition, we could not achieve this smooth degeneration and would pay more overhead.

In our actual implementation we store each term contiguously without any decomposition. At the moment this is sufficient for our application, but we plan to introduce partial decomposition in a later version of LILOG-DB (see Section 9).

To make processing of these partially self-identifying records efficient we proceeded in two steps: First we introduced a data structure, designed for fast processing, which contains the schema information we use for compact representation. In the second step we developed TRAM and the TRAM compiler, detailed in the next sections.

In the upper parts of LILOG-DB sort information is represented by data structures which have to cover the whole expressive power of sort specifications. This large expressive power is very desirable on the semantic level, but much of it is "useless" for term representation. We decided to extract "useful" information from sort specifications at ft-set creation time. The so called *internal schemas* are stored in the data dictionary. Internal schemas are and/or graphs. A simple example is shown in Fig. 3. For example, constraints, like the coreference condition *hotel.name.*□ = *name.*□ cannot be represented by this data structure. But internal schemas are straightforward to process. Internal schemas can be viewed as tree automata, which accept a superset of the terms described by the original sorts.

Inside the Term Manager, internal schemas are used during update and retrieval:

The *Update Manager* accepts a term in its main memory representation and writes all information from this term, it can not find in the internal schema, into the disk record.

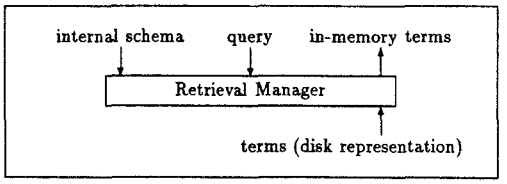

Figure 1: Retrieval Manager

The *Retrieval Manager* has the opposite task. It merges information from the internal schema and the disk record to build the self-identifying main-memory representation required by the query processor and the top level logic programming system.

4 Retrieval by Interpretation

In this section we sketch how the Retrieval Manager was implemented and how its performance was improved by stepwise refinement ("step two" in the introduction). Fig. 1 shows the inputs and outputs of the Retrieval Manager. It gets an internal schema and a (sub)query from an upper software layer. Terms in their disk representation are retrieved by a lower layer. The answer terms are expected to be in the main-memory representation.

Naive Interpreter

The Retrieval Manager has to translate terms from their disk representation to their main memory representation and to apply the γ-operator to them. In the first naive implementation these two tasks were done by two independent software layers: First the term under consideration was transformed to its main memory representation, then the conditional constructor of the γ-operator was evaluated.

The advantage of this architecture is its simplicity. The γ-operator evaluation procedures have not to deal with the internal schema because our main memory term representation is self-identifying like in most Prolog or Lisp systems. Only the term translation procedures have to know the disk representation and to interprete the internal schema. But they can work without knowledge about the query.

The disadvantage of this naive Retrieval Manager architecture is its inefficiency for γ-operators, which allow to pass only a small percentage of the data read from disk. This may either be due to a selective condition or to a narrow projection which gets only small subterms. The Retrieval Manager's problem is that it translates all terms completely to their main memory representation regardless of the fact that most of them never are used.

Direct Path Evaluation

To avoid this useless work we implemented a path evaluation interpreter which operates on the disk representa-

tion. It directly interpretes the disk data and navigates to the subterm specified by the path the caller submitted. The γ-operator evaluation procedures now use this new path evaluator to navigate to the subterms they need and only trigger the translation of these subterms to the main memory representation.

The main idea of this refined Retrieval Manager is to translate only subterms actually needed into their main memory representation. For most queries it is faster than the naive interpreter, but it still translates terms not used by upper layers of LILOG-DB. Many small subterms are translated to their main memory representation only to be used in the condition evaluator.

Direct Condition Evaluation

Most conditions are composed of simple comparisons with atomic constants. In the next refinement step we additionaly extended the condition interpreter to evaluate conditions of the form *"path* Θ *constant"* ($\Theta \in \{=, \neq, <, \leq, \geq, >\}$) by direct interpretation of the disk representation. For many queries this change resulted in a considerably performance improvement. The better performance is not only due to the translation cost we avoid. But the avoidance of main memory allocations and deallocations, which are indirect costs of the translation, contributes much to the performance gain.

We did not push the evaluation of all conditions to the disk representation because this would make the interpreter very complex. For many queries there is a simple way to achieve performance nevertheless. The EFTA optimizer should transform conditions which include complex constructors to conditions the Retrieval Manager can evaluate very fast. If the Term Manager user does not optimize, he still gets the correct answer, but he has to accept additional execution costs.

The third Retrieval Manager is much faster than the first naive interpreter. Even for simple conditions we could demonstrate performance improvements of up to 100% [21]. But for complex queries sequential scans are still too expensive. By analyzing the Term Manager with the gprof profiler [12], we could identify the path evaluation procedure as a main bottleneck.

The task of the path evaluation procedure is to navigate through the disk representation of complex terms. The procedure gets a path, an internal schema and access to the disk representation of a term. It has to return the position inside the disk representation and the position inside the internal schema which correspond to the input path. It has to interpret the internal schema and the disk representation according to the path in parallel. A statement by statement analysis revealed that the execution costs of both intermingled interpreters are similar. However, there is an important difference between the inputs: The disk representation to be analyzed changes in every step of a scan loop, but the internal schema and the path are loop invariants.

[1] HIGH() gets the upper boundary of an array variable.

The key idea, which led to the development of TRAM, is to move these invariants outside of the loop, i.e. to *compile* path applications and internal schemas. In the following section we shall describe the design of TRAM in detail.

5 How to Invent a TRAM

The title of this section is an allusion to a paper of Kursawe [23], who describes the method we applied to develop an abstract machine for term retrieval. He shows, that there is a very natural way to derive the Warren Abstract Machine [36, 1], the standard target architecture for Prolog compilation. The idea is to specialize an abstract Prolog interpreter together with a set of Prolog programs to be interpreted by partial evaluation [11, 31]. The resulting program carefully is analyzed to extract a set of elementary "key algorithms", which are the heart of the WAM.

Applied to the retrieval interpreter we described in the section above, this means that we have to specialize this interpreter for a set of queries to get an idea how the target language of a query compilation should look like. Because of the size of the retrieval interpreter, we shall demonstrate this here by example. The actual implementation was done in C. To make the presentation more compact and readable, we use Modula-2 like pseudo-code in this paper.

The Path Evaluation Interpreter

In Fig. 2 we have sketched the *eval_path* interpreter. It has three arguments: The path to be evaluated is represented by an array, the internal schema is represented using pointers, and the disk representation is accessed with a set of *get_...* methods using the 'read pointer' *rp* as an object handle. In the figure we introduced parenthesized labels we now use to refer to individual statements:

(1) The main loop of *eval_path* traverses the argument path element by element[1].

(2) The first statement inside the loop treats disjunctive schemas. For OR-nodes in the internal schema graph the Update Manager selects the appropriate ancestor node. To indicate its selection the Update Manager stores the edge number in the disk representation. We read this number by *get_variant_number* and skip to the appropriate ancestor of the OR-node by *skip_to_child*. Because our internal schema build-up procedure guarantees that there are no consecutive OR-nodes, we do not need to loop the treatment of disjunctive schemas.

(3) Our schema variable now points to an AND-node. Instruction (3) demonstrates the main idea of the use of internal schemas in the LILOG-DB Term Manager. A certain kind of information might be known in the schema or it must be read from the disk representation. If the tag field of the current term node is known in the schema, we extract it from the schema node, otherwise we get the tag value from disk by *get_tag*.

(4) The next IF-statement distinguishes between symbolic path elements and numeric path elements. In Fig. 2 we

```
PROC eval_path(path; VAR schema; VAR rp): BOOLEAN;
(1)   FOR i := 0 TO HIGH(path) DO
(2)     IF is_or_node(schema) THEN
            get_variant_number(rp, v);
            skip_to_child(schema, v)
(3)     IF tag_known(schema)
            THEN tag = schema↑.tag;
            ELSE get_tag(rp, tag) ;
(4)     IF is_symbol(path[i]) THEN
(5)       IF tag ≠ tuple
              THEN RETURN FALSE;
(6)       IF arity_known(schema)
              THEN arity := schema↑.arity
              ELSE get_arity(rp, arity) ;
(7)       pos := pos_in_schema(path[i], schema);
(8)       IF pos > 0 THEN
              IF NOT jump_to(rp, arity, pos) THEN
                  RETURN FALSE
(9)       ELSE
              IF NOT search_feature_name(rp,
                        arity, schema, path[i]) THEN
                  RETURN FALSE
        ELSE (* handle numeric path element ...*)
      RETURN TRUE
```

Figure 2: The Path Application Interpreter

show only the code of the former case.

(5) If the path element actually is a symbol, the current term node should be a tuple. If we have navigated to any other kind of term node, the path evaluation fails.

(6) This instruction gets the arity of the tuple node either from the schema or from the disk representation.

(7) A feature name may be known in the schema, or it may be an optional feature allowed by our open tuple semantics. By *pos_in_schema* the schema node is searched for the current path element. In the case of success a positive position number is returned. On the disk representation this position number is the index in an array of offsets.

(8) The selected array element points to the feature value we are looking for. Like for attribute names in conventional relational database systems, the feature name is not stored on disk. In our pseudo code we hide the offset array lookup operation in *jump_to*, which may fail for absent features or for out of boundary positions.

(9) If a feature is not known in the internal schema, its name and its value are stored on disk. These optional features are located just behind the offset array in alphabetical order. *search_feature_name* performs a binary search for the current path element in the disk record.

The *eval_path* fragment should give an idea on the mode of operation of the whole retrieval interpreter. Next we shall demonstrate how to specialize this interpreter for a certain query by partial evaluation.

Partial Evaluation Example

In our example we assume an ft-set stored using the schema shown in Fig. 3. The sort *tree* is described as a disjunction of a tuple and an one-element enumeration sort. Inside the tuple sort the features *left* and *right* recursively are defined to be of sort *tree*. The internal schema shown in Fig. 3 is the translation of the sort *tree* into an internal schema and/or graph.

The query, we use in our example, is a simple projection to the path *left.right.color*.□.

The first obvious transformation to get the specialized procedure *eval_path_left_right_color* shown in Fig. 4 from the general path application interpreter *eval_path*

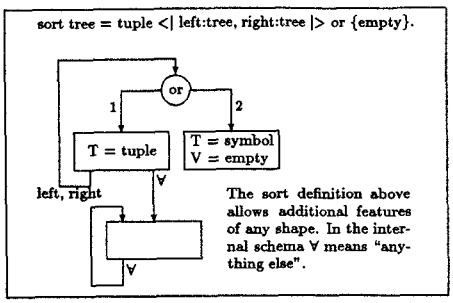

Figure 3: Sort and Internal Schema of our Example

(Fig. 2) is to unfold the for-loop. We simply concatenate the body of the loop three times. In the first copy we replace *path*[*i*] by *left*, in the second copy by *right* and in the final copy by *color*. We replace the *is_symbol* tests (4) by *true* and eliminate the unreachable code for handling numeric path elements. Next we propagate all internal schema information which is available into the procedure:

- When entering each of the former loop bodies, the schema variable points to the OR-node of the *tree*-schema. We can delete the tests *is_or_node*(*schema*) and make the instruction *get_variant_number* unconditional. Because in all three cases the remaining path evaluation only can succeed for the first variant, we insert IF-statements to test the variant number variable *v* to be equal to 1. The calls of *skip_to_child* are evaluated now at 'compile-time'.
- Instruction (2) is evaluated at compile-time, too. We know that the tag is *tuple*, the run-time tests (5) for this are deleted.
- There is no arity information in the internal schema. The *arity_known* tests (6) are eliminated, the *get_arity* instructions remain in the procedure.
- The *pos_in_schema* search operations (7) and the attached tests (8) are the next things we can evaluate now. The first occurence of *pos_in_schema* is evaluated to 1, the second occurence to 2 and the third occurence to 0. These values are propagated along the *pos* variable to the third argument of *jump_to*. The unreachable branches of instruction (8) are deleted.

The resulting specialized path application procedure *eval_path_left_right_color* shown in Fig. 4 does not contain any reference to the path or to the internal schema. We performed the partial evaluation

eval_path_left_right_color(rp) ←
    eval_path(left.right.color.□, 'tree', rp).

Implementation Techniques

We have demonstrated a technique to derive fast specialized path evaluation procedures. This technique works well for the other parts of the retrieval interpreter, too. It would be possible to stop our example here and to develop a compiler which produces C procedures of the style of Fig. 4. In a second step these procedures are translated by a C compiler to cpu-instructions. This binary code then is linked to the database system kernel either by a static or by a dynamic loader.

```
PROC eval_path_left_right_color(VAR rp): BOOLEAN;
    get_variant_number(rp, v);
    IF v = 2 THEN RETURN FALSE;
    get_arity(rp, arity);
    IF NOT jump_to(rp, arity, 1) THEN
        RETURN FALSE;
    get_variant_number(rp, v);
    IF v = 2 THEN RETURN FALSE;
    get_arity(rp, arity);
    IF NOT jump_to(rp, arity, 2) THEN
        RETURN FALSE;
    get_variant_number(rp, v);
    IF v = 2 THEN RETURN FALSE;
    get_arity(rp, arity);
    RETURN search_feature_name(rp, arity, 2, 'color');
```

Figure 4: A Specialized Path Application Procedure

The problem of producing C procedures is that it would commit us to a certain implementation technique. There are at least two other styles of target code, which might be attractive:

- An extreme approach is to generate machine code for the hardware cpu directly. The very large overhead of calling the C compiler is avoided and the code can be optimized beyond the level possible in C. But the main disadvantage of generating cpu instructions is the bad portability of the compiler.
- Another technique is to generate code which is executed by a software interpreter. The main advantage of interpretation is high portability. The disadvantage is the interpretation overhead.

Nevertheless we decided to explore interpretation first. We will justify this below.

The critical parameter of interpretation is its level. If the level of the instructions known by the interpreter is to low, to much instructions have to be executed to solve the problem. The execution of a single instruction might be cheap, but the interpretation overhead kills performance. This is especially a problem when simulating hardware architectures by software interpreters [30].

The level of interpretation becomes to high if the whole power of instructions often is not utilized and the instructions themselves are expensive. In the area of hardware interpreters this problem was a reason to develop RISC architectures.

The benefit of the top-down partial evaluation technique used in our example is that it gives us good directions to design the instruction set of an interpreter. The specialized evaluation procedures produced by partial evaluation are composed of building blocks. These are good candidates for interpreter instructions.

Fig. 5 shows the instructions we derived from the *eval_path_left_right_color* procedure. A simple pattern we find in this procedure three times is the *get_variant_number* call and some branching behind it. The analysis of more complex situations than the example we presented, showed, that is is appropriate to merge the *get_variant_number* call and a simple jump table mechanism into one instruction. The *variant_switch* instruction has a variable number of arguments, which are labels. If the variant number read from the disk representation is 1, the interpreter jumps to the

```
        variant_switch left, fail / fail
left:   sym_path_tu_xx_fe 1 / fail
        variant_switch right, fail / fail
right:  sym_path_tu_xx_fe 1 / fail
        variant_switch color, fail / fail
color:  sym_path_tu_xx_xx 2, 'color' / fail
        ...
fail:   ...
```

Figure 5: A TRAM Program for Path Application

first argument label, if it is 2, it jumps to the second label, etc. If something goes wrong or the variant number is larger than the number of arguments, the last label is used. In our notation we separate this failure-label by '/'.

When analyzing the remaining statements in Fig. 4, we could be tempted to use two instructions for the treatment of each path element: a *get_arity* instruction and a *jump_to* or *search_feature_name* instruction combined with some failure exit. But it is more appropriate to raise the level of interpretation and to design instructions which are able to handle the application of a path element completely. To avoid too powerful instructions, we enumerate all situations which can occur when translating a symbol path element:

tag	arity	feature
tuple	known	known
tuple	unknown	known
tuple	known	unknown
tuple	unknown	unknown
unknown	known	unknown
unknown	unknown	unknown

In all other situations the application of the path element must fail. The instructions used in Fig. 5 are simply mnemos for the lines in the table above. *sym_path_tu_xx_fe* means: *"treat a symbol path element for a term known to be a tuple, with unknown arity, but known feature."* The only arguments we need for this instruction are the position of the feature in the offset array and a failure label the interpreter should escape to for problems.

The Term Retrieval Abstract Machine instruction set completely was designed using the methods sketched in our example. We call it an *abstract* machine, because it is a model useful for all implementation techniques sketched above. Direct interpretation, we assumed in our argumentation and we actually used in our first TRAM implementation, only is the most obvious technique. For machine code or C code compilers the TRAM instruction set is a very good intermediate level code. The TRAM compiler, described in Section 7, then becomes the front-end of an EFTA to machine or EFTA to C compiler.

6 The TRAM Instruction Set

TRAM is a *semantics-directed machine architecture* [31]. Like WAM [36], it combines very specialized and very general instructions in one set. This section surveys the TRAM instructions. We can not describe all details here, but a lot of them will become more distinct in the next section about compilation.

The instruction set is organized in six groups:

(1) The first group covers *miscellaneous instructions*. A *no operation* (*nop*) instruction is included for

convenience only, it is used during compilation. The *exit_null* instruction is used to indicate failure to the caller of TRAM. If TRAM succeeds, it returns the results of its computation by *exit_term*. The two *exit_* instructions are the only way to leave the TRAM. *_rp* instructions are needed to manipulate the *rp*-register (read pointer), which walks through the disk representation of terms. *reset_rp* restores *rp* to the beginning of the disk representation. By using *store_rp* and *load_rp* it is possible to remember *rp*-states. *variant_switch* is the jump-table instruction to handle disjunctive schemas. The unconditional jump instruction *goto* is the necessary glue to patch up code parts.

(2) Group 2 and 3 contain the most important instructions of the TRAM. These instructions are specialized to evaluate path applications as efficient as possible. The instructions of group 2 handle *symbolic path elements*. The compiler chooses among the 6 instructions according to the knowledge he got from the internal schema.

(3) Group 3 are path application instructions for *numeric path elements*. We consider group 2 and 3 to be the core of the TRAM instruction set because these instructions are responsible for much of TRAMs performance gain against the interpretation of conditional constructors.

(4) There are only four instructions to *load terms* from the disk representation into in-memory data structures. The most powerful of them is *get_xx*, which directly interprets internal schemas to load complex terms.

(5) Instructions to evaluate *constructors* are more elaborated. TRAM provides specialized instructions for all atomic and complex constructors of EFTA. The universal constructor interpreter instruction *construct_xx* presently is only used to handle complex constants, but it was included to facilitate experimental EFTA expansions.

(6) The instruction group to handle *conditions* is the largest. These TRAM instructions are used to test EFTA conditions against secondary storage terms. The central role of condition testing for the performance of scans justifies the large number of very specialiced instructions in this group.

TRAM has only a small number of registers. The most important registers are the read pointer register *rp*, which points to the current position in the disk record, and the program counter *pc*. The *get_*, *construct_* and *exit_* instructions work on a stack, which is controled by the stack pointer register *sp*.

Additionally TRAM has some pseudo registers, which are initialized when TRAM is started to analyze a term. These pseudo registers are never changed during a TRAM computation. Pseudo registers are used to define the environment, TRAM needs to work, like the base address of the stack, the beginning of the disk record, and the location of the internal schema.

TRAM is build on top of the Retrieval Manager interpreter from section 4. We do not compile the translation of complex terms from disk representation to in-memory representation. The reason is, that this would not change the level of interpretation, as long as TRAM is implemented by an interpreter. A TRAM extended for term translation would perform exactly the same steps as the interpreter (*"get_term"*), which alternatively can be described as a machine reading the internal schema as its program. The term translation by TRAM would require additional TRAM registers to implement a *call/return* mechanism for handling schemas with recursion.

A guideline for the design of TRAM was to compile the tree shaped query expression as long as it seems profitable, but never to compile the recursion of internal schemas. *get_term* and utility procedures to skip disk terms loop into internal schemas, they are called by the *get_xx* instruction, or by the path application instructions, respectively.

This policy does not only make TRAM simpler and faster, but it additionaly makes the TRAM instruction set independent of the term representations on disk and in memory. The TRAM architecture is dependent on our data model and on our query language, but the instruction set does not change if we use different tuple representations or another functor node layout. The main memory representation is only required to provide constructors to build terms bottom-up.

The representation of TRAM instructions in our implementation is straightforward. Each TRAM instruction is coded in consecutive elements of a code array. The first word of a instruction always contains its operation-code. The number of arguments stored in the code words following the opcode depends on the kind of instruction. Most instructions have a fixed number of arguments, only *variant_switch* and *construct_tu* are of variable length.

The TRAM code representation is not very compact. For example it would be sufficient to allow 7 bits for the opcode or 16 bits for jump addresses. But some arguments need 32 bits. Because TRAM is simulated by software written in C, we consider a simple and fast to decode instruction format more important than a very compact code and decided to use only 32 bit words to prevent alignment problems. For TRAM interpreters written in assembler or microcode this design trade-off should be reconsidered.

Our TRAM simulator has two arguments: The first is the initialization value of the *rp* register, the second is the code array. The code array is divided into three segments:

- The *header segment* contains initialization values of registers and pseudo registers. *rp* is not initialized from this header, because it is the only input which changes for each term of a scan. All other initialization values are calculated at compile time.
- The second segment is the *program segment* with the TRAM code.
- The data segment is located at the end of the code

array. It is used to remember read pointer states to avoid evaluating the same path repeatedly. The stack resides in the data segment, too. Because the nesting of complex constructors is known at compile-time, we can predict exactly the stack size required for a program and allocate à code array of the appropriate size in advance.

In the next section we briefly explain how the TRAM code is constructed by the TRAM compiler.

7 The TRAM compiler

The design of the TRAM compiler was strongly influenced by some experiments in implementing compilers in Prolog using the techniques described by Warren and Cohen [35, 7]. Parts of the TRAM compiler first were sketched in Prolog, but we decided to implement it in C because we do not want to limit the portability of our Term Manager by the availability of a certain logic programming environment.

The compiler specification was refined simultaneously with the TRAM instruction set design to guarantee that the abstract machine and the compiler match in every detail.

The TRAM compiler works in three phases: (1) The first phase translates the conditional constructor of the query and the internal schema into intermediate code. (2) This intermediate code is improved by the peephole optimizer. (3) Finally the intermediate code data structures are transformed to build the code array.

#	sort	file size
1	term	5,128,192
2	ts2 = tuple <| x100:int, x:ts2 |>	4,104,192
3	ts3 = tuple <| x100:int, x50:int, x:ts3 |>	4,104,192
4	ts4 = tuple <| x100:int, x50:int, x10:int, x:ts4 |>	3,424,256
5	ts5 = tuple <| x100:int, x50:int, x10:int, x5:int, x:ts5 |>	3,424,256
6	ts6 = tuple <| x100:int, x50:int, x10:int, x5:int, x3:int, x:ts6 |>	2,936,832
7	ts7 = tuple <| x100:int, x50:int, x10:int, x5:int, x3:int, x2:int, x:ts7 |>	2,936,832

Figure 6: Sorts and Sizes of the Test FT-Sets

8 Performance Evaluation

The expense to implement the TRAM simulator and compiler only is justified if we can demonstrate that TRAM increases the Retrieval Manager performance. As described in Section 4 our starting point to design TRAM was an interpreter, which already had been tuned. Therefore we can not expect performance improvements of the order reported above, again. First ad hoc experiments with a literature database indicated a performance gain of 15 – 30 % compared with the fast interpreter for simple queries. For complex queries better results could be obtained.

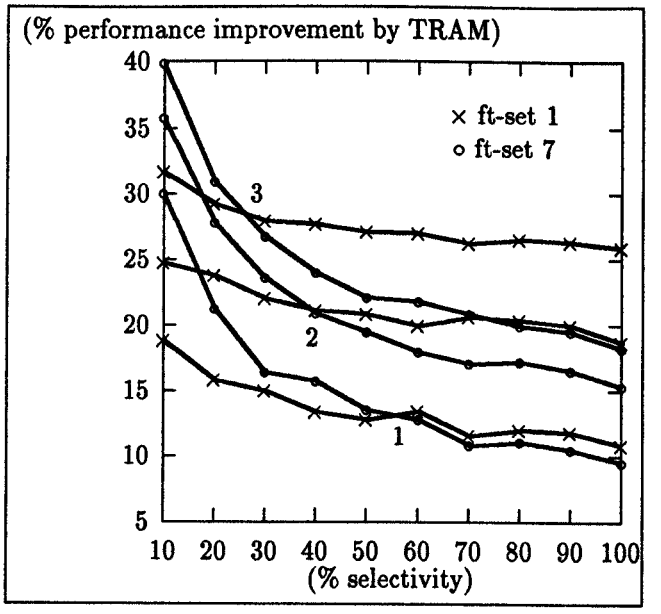

Figure 7: Selectivities vs. Performance Gains

To establish these results and to get more exact values we performed extensive performance evaluations. Our experiments were done on a standalone workstation exclusively used for our measurements. A database generated by the tool described in [21] was loaded on the disk of the workstation.

We used the *gprof* utility [12] to generate procedure level profiles of the LILOG-DB system. The profiles were cumulated to the module level to get times of the two Retrieval Manager versions and the underlying modules they use.

With *gprof* we faced the same problem as reported by the ORION group [19]: The results are rather inaccurate and each measurement has to be repeated several times to get more reliable values. The experiments, reported here, were repeated 10 times. Altogether, Fig. 9 and 10 are based on 10800 runs of LILOG-DB.

The database we used in our experiments, contains 7 ft-sets each with the same set of 5000 feature tuples. Each tuple has 6 integer value features and a feature with a nested tuple:

```
<| x100:36, x50:18, x10:4, x5:0, x3:2, x2:1, x:
    <| x100:27, x50:12, x10:5, x5:4, x3:0, X2:1, x:
        ... |> |>
```

All terms are of identical shape, the level of nesting is 9. The integers are random values to allow tests with different selectivities like in the Wisconsin Benchmark [4].

We used 7 different sort specifications to build the ft-sets. Fig. 6 shows the sort specifications and the resulting file sizes in bytes. The sort specifications range from the universal sort *term*, which forces the Term Manager to use a totally self-identifying disk representation, to *ts7*. This sort describes all features, which occur in our test data, but it allows missing features or

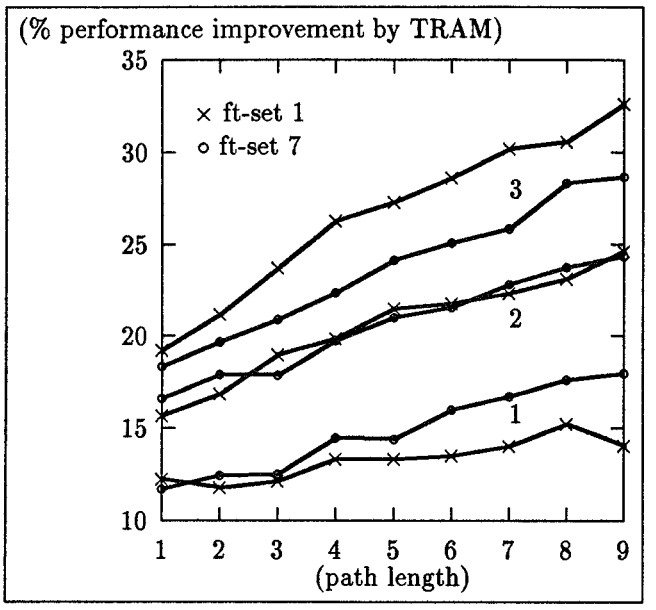

Figure 8: Path Lengths vs. Performance Gains

additional features.

The file sizes impressively demonstrate the high costs of missing schema information. The steps in the size function are a result of fragmentation at the mapping of the records to 4KByte disk blocks.

The queries, we used to evaluate TRAM performance, are composed of simple comparisons of a path with an integer constant. They were varied in three dimensions:

- The number of elementary comparisons connected by *and* were 1, 2 or 3.
- The selectivity was controled by choosing appropriate integer constants. In our tables the second columns show the selectivities in percent.
- The length of the paths used in the conditions range from 1 to 9. In composed conditions we always used identical paths for all comparisons.

An analysis of the performance improvements of the TRAM Retrieval Manager compared to the fast interpreter Retrieval Manager reveals that the variation caused by the inaccuracy of *gprof* cannot hide some significant tendencies:

- The performance gain of TRAM is higher for low selectivities than for high ones.
- If the path length is increased, the performance gain becomes slightly larger.
- The reuse of intermediate results makes TRAM superior for conditions which contain the same path several times.

In Fig. 7 the average performance gains for paths with $len(path) \in [1 \ldots 9]$ has been plotted as a function of the selectivity. The condition complexities are indicated in the figure by 1, 2 or 3, respectively.

In Fig. 8 we calculated the average over all selectivities and show the performance gains as a function of the path length.

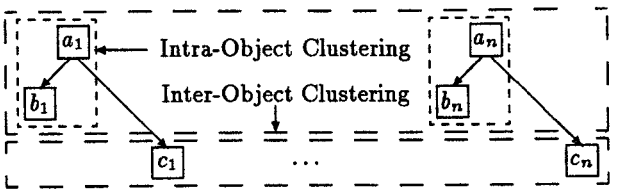

Figure 9: Inter-Object and Intra-Object Clustering

The absolute Retrieval Manager CPU times to process the 5000 terms ranged from about 1.5 seconds for 10% selectivity to 12 seconds for 100% selectivity. In the compiled version only term translation is a relevant for the costs.

Because the TRAM simulator only has to interprete a very small number of instructions to treat one term, we expect only small shifts of the performance functions for a system which generates machine code to realize TRAM. To exploit this may be necessary for a production system, but we do not intend to deal with this standard technique in our prototype system, because it would limit portability.

9 Perspectives

In our current implementation each feature term is represented by exactly one disk record. This works well for our applications, where the terms are either of moderate size or are always retrieved and updated in its entirety. But if terms are very large and usually only small portions of them are loaded or updated, doubtless a not clustered decomposed representation would be better. Because of the problems discussed in Section 3, we think that decomposition of terms should not be done on a node level, but on a more coarse granulation. Ideally the database system would learn which subterms usually are retrieved or updated and it would optimize i/o and cpu costs in the tradeoff between a representation which is more clustered and faster to manage terms in its entirety, or a representation which is more decomposed and better to handle small subterms. In reality the points of fracture may be commited at design time by the user.

A technique to assemble decomposed complex objects during retrieval recently was described by Keller, Graefe and Maier [17]. The idea of their *assembly operator* is to assemble w objects in parallel like on an assembly line to avoid to much jumping between inter-object clusters (Fig. 9 , adapted from [17]). w is the window size or assemly line capacity, which limits the number of objects under construction. TRAM is a technique to handle intra-object clustering, the assembly operator is a technique to profit from inter-object clustering, i.e. these techniques are orthogonal and should be combined. The assembly operator would be the manager and a collection of TRAM programs attached to the inter-object clusters would be the workers.

Another important aspect of complex object representations is the identity of objects and subobjects. Conceptual object identity and physical object identity are to a large extend independent [22]. Our data model is value-based and does not support conceptual object identity. Nevertheless persistent objects have some kind of address or identity used in access paths. For large objects fast intra-object access paths become important. An obvious implementation technique is to force decomposition at large indexed intra-object collections.

We plan to explore the potentialities of partially decomposed representations. Our first impression is that decomposition may be very beneficial if carefully directed, but that we would pay too much when doing it by default for our objects with incomplete schemas.

References

[1] H. Aït-Kaci: Warren's Abstract Machine, A Tutorial Reconstruction. MIT Press, 1991

[2] J. Banerjee, W. Kim, S.-J. Kim, J.F. Garza: *Clustering a DAG for CAD Databases.* IEEE Trans. Software Eng., vol.14, no.11, Nov. 1988, 1684–1699

[3] S. Benzschawel, E. Gehlen, M. Ley, T. Ludwig, A. Maier, B. Walter: *LILOG-DB: Database Support for Knowledge Based Systems.* In [15], 501–594

[4] D. Bitton, D.J. DeWitt, C. Turbyfill: *Benchmarking Database Systems, a Systematic Approach.* VLDB 1983, 8–19

[5] E.E. Chang, R.H. Katz: *Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS.* SIGMOD Conf. 89, 348–357

[6] J.-b. R. Cheng, A.R. Hurson: *Effective Clustering of Complex Objects in Object-Oriented Databases.* SIGMOD Conf. 1991, 22–31

[7] J. Cohen, T.J. Hickey: *Parsing and Compiling Using Prolog.* ACM Trans. Programming Languages and Systems, vol.9, no.2, April 1987, 125–163

[8] P. Dadam et al.: *A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies.* SIGMOD Conf. 1986, 356–367

[9] O. Deux et al.: *The Story of O^2.* IEEE Trans. Knowledge and Data Eng., vol.2, no.1, March 1990, 91–108

[10] P. Drew, R. King: *The Performance and Utility of the Cactis Implementation Algorithms.* VLDB 1990, 135–147

[11] A.P. Ershov et al. (eds.): *Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, Avernæs, Denmark, October 1987.* Special Issue of: New Generation Computing, vol.6, nos.2,3, 1988

[12] S.L. Graham, P.B. Kessler, M.K. McKusick: *gprof: a Call Graph Execution Profiler.* Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, SIGPLAN Notices, vol.17, no.6, 120–126, June 1982

[13] J. Gray, A. Reuter: *Transaction Processing: Concepts and Techniques.* To be published by Morgan-Kaufman, San Mateo, CA, 1992

[14] C.T. Haynes: *Logic Continuations.* Journal of Logic Programming, vol.4, no.2, June 1987, 157–176

[15] O. Herzog, C.-R. Rollinger (Eds.): *Text Understanding in LILOG.* Lecture Notes in Computer Science, vol.546, Springer 1991

[16] J. Joseph, S. Thatte, C. Thompson, D. Wells: *Report on the Object-Oriented Database Workshop, held in conjunction with OOPLSA'88.* SIGMOD Record, vol.18, no.3, Sept. 1989, 78–101

[17] T. Keller, G. Gräefe, D. Maier: *Efficient Assembly of Complex Objects.* SIGMOD Conf. 1991, 148–157

[18] W. Kim, J. Banerjee, H.-T. Chou, J.F. Garza, D. Woelk: *Composite Object Support in an Object-Oriented Database System.* OOPSLA'87 Proceedings, SIGPLAN Notices, vol.22, no.12, Dec. 1987, 118–125

[19] W. Kim, H.-T. Chou, J. Banerjee: *Operations and Implementation of Complex Objects.* IEEE Trans. Software Eng., vol.14, no.7, July 1988, 985–996

[20] W. Kim, J.F. Garza, N. Ballou, D. Woelk: *Architecture of the ORION Next-Generation Database System.* IEEE Trans. Knowledge and Data Eng., vol.2, no.1, March 1990, 109–124

[21] M. Klein: *Leistungsanalyse des Datenbanksystems LILOG-DB.* IBM, IWBS Report, 1991 (in German)

[22] S. Khoshafian, M.J. Franklin, M.J. Carey: *Storage Management for Persistent Complex Objects.* Information Systems, vol.15, no.3, 1990, 303–320

[23] P. Kursawe: *How to Invent a Prolog Machine.* Third Int. Conf. on Logic Programming, London 1986, Lecture Notes in Computer Science, vol.225, Springer 1986, 134–148

[24] M. Ley: *Low Level Main Memory Management in LILOG-DB — notes from the implementation underground.* Submitted for publication, 1991

[25] M. Ley, B. Walter: *Der LILOG-DB-Fact-Manager: Ein Datenbankkern zur Speicherung variabel strukturierter komplexer Objekte.* Informatik Forsch. Entw., vol.5, 1990, 188–201 (in German)

[26] T. Ludwig: *A Brief Overview of LILOG-DB.* in: Proc. of 6th Data Eng. Conf., L.A. 1990, 420–426

[27] T. Ludwig: *Query Processing in LILOG-DB: What It Is and Where it Goes.* Datenbanksysteme in Büro, Technik und Wissenschaft, GI-Fachtagung Kaiserslautern, März 1991, Informatik-Fachberichte, Bd.270, 271–287, Springer

[28] T. Ludwig, B. Walter: *EFTA: a database retrieval algebra for feature-terms.* Data & Knowledge Engineering, vol.6, 1991, 125–149

[29] A. Maier, M. Ley, E. Gehlen: *Sort Processing in a Deductive Database System.* IBM, IWBS Report 154, 1991

[30] C. Mills, S.C. Ahalt, J. Fowler: *Compiled Instruction Set Simulation.* Software — Practice and Experience, vol.21, no.8, August 1991, 877–889

[31] *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91, Yale University, New Haven, CT, USA, June 17–19, 1991.* SIGPLAN Notices, vol.26, no.9, Sept. 1991

[32] H.-J. Schek, H.-B. Paul, M.H. Scholl, G. Weikum: *The DASDBS Project: Objectives, Experiences, and Future Prospects.* IEEE Trans. Knowledge and Data Eng., vol.2, no.1, March 1990, 25–43

[33] S. Thatte: *Report on the Object-Oriented Database Workshop: Implementation Aspects, held in conjunction with OOPSLA'87.* SIGPLAN Notices, vol.23, no.5, May 1988, 73–87

[34] M.M. Tsangaris, J.F. Naughton: *A Stochastic Approach for Clustering in Object Bases.* SIGMOD Conf. 1991, 12–21

[35] D.H.D. Warren: *Logic Programming and Compiler Writing.* Software — Practice and Experience, vol.10, 1980, 97–125

[36] D.H.D. Warren: *An Abstract Prolog Instruction Set.* Technical Note 309, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1983