Journal of Intelligent Information Systems, 19:1, 111–134, 2002

Efficient Management of Persistent Knowledge

DIMITRIS G. KAPOPOULOS dkapo@di.uoa.gr
MICHAEL HATZOPOULOS mike@di.uoa.gr
PANAGIOTIS STAMATOPOULOS takis@di.uoa.gr
Department of Informatics and Telecommunications, University of Athens, Panepistimiopolis, Ilisia 157 84, Greece

Received March 27, 2000; Revised June 4, 2001; Accepted October 16, 2001

Abstract. Although computer speed has steadily increased and memory is getting cheaper, the need for storage managers to deal efficiently with applications that cannot be held into main memory is vital. Dealing with large quantities of clauses implies the use of persistent knowledge and thus, indexing methods are essential to access efficiently the subset of clauses relevant to answering a query. We introduce PerKMan, a storage manager that uses G-trees, and aims at efficient manipulation of large amounts of persistent knowledge. PerKMan may be connected to Prolog systems that offer an external C language interface. As well as the fact that the storage manager allows different arguments of a predicate to share a common index dimension in a novel manner, it indexes rules and facts in the same manner. PerKMan handles compound terms efficiently and its data structures adapt their shape to large dynamic volumes of clauses, no matter what the distribution. The storage manager achieves fast clause retrieval and reasonable use of disk space.

Keywords: G-trees, performance evaluation, persistent knowledge, storage manager

1. Introduction

Although computer speed has steadily increased and memory is getting cheaper, the need for storage managers to deal efficiently with applications that cannot be held into main memory is vital. An example of such application might be a logic-based data mining system which needs to access a very large telephone directory.

The efficient management of persistent knowledge in deductive database systems requires the adoption of effective indexing schemes in order to save disk accesses, while maintaining reasonable use of available space. *Deductive database systems* incorporate the functionality of both logic programming and database systems. They have four major architectures: ‘logic programming systems enhanced with database functionality,’ e.g., NU-Prolog (Ramamohanarao et al., 1988), ‘database access from Prolog,’ e.g., BERMUDA (Ioannidis et al., 1994), TERMdb (Cruickshank, 1994), ‘relational database systems enhanced with inferential capabilities,’ e.g., Business System 12 (Boas and Boas, 1986), and ‘systems from scratch,’ e.g., SICStus (Nilsson and Ellemtel, 1995), CORAL (Ramakrishnan et al., 1994), Aditi (Vaghani et al., 1994), Glue-Nail (Derr et al., 1994), XSB (Sagonas et al., 1994), ECL^{i}PS^e(ECL^{i}PS^e 3.7, 1998).

In multidimensional data structures, all attributes are treated in the same way and no distinction exists between primary and secondary keys. This seems to be suitable in a knowledge base environment, where queries are not predictable and clauses may be used in a variety of input/output combinations.

G-trees (Kapopoulos and Hatzopoulos, 1999; Kumar, 1994) are adaptable multidimensional structures that combines the features of B-trees and grid files. They have the ability to adapt their shape to high dynamic data spaces and to non-uniformly distributed data. G-trees divide the data space into a grid of variable size partitions. Only non-empty partitions, which span the data space, are stored in a B-tree-like organization. G-trees use a variable-length partition numbering scheme. Each partition is assigned a unique partition number. This number is a binary string of 0's and 1's, where leading zeros are significant. Each partition corresponds to a physical disk block, and data points are assigned to it until it is full.

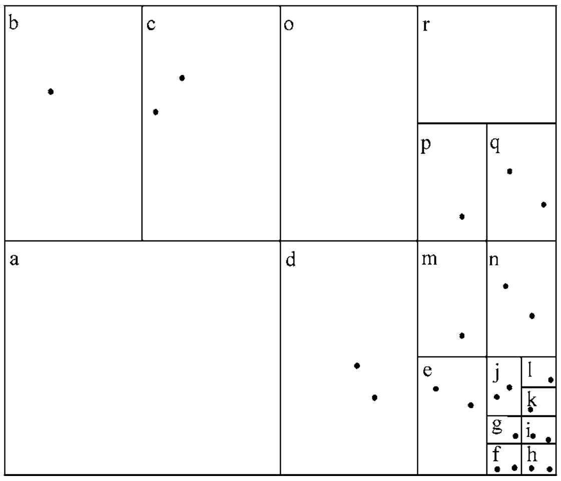
A full partition, P, is split into two equal sub-partitions $P0$ and $P1$, e.g., if $P = 10$, then $P0 = 100$ and $P1 = 101$. The points of P are moved to $P0$ and $P1$. P is deleted then from the G-tree, while $P0$ and $P1$, if non-empty, are inserted. The new entry is assigned to the appropriate child P_c. If there is room, the new entry is added to P_c. Otherwise, P_c must be split. The splitting dimension alternates with a period equal to the number of dimensions in a way that each dimension appears once in a cycle. No meta-information is stored regarding splitting. The splitting point is fixed because the attribute domains are split in the middle.

Figure 1 shows the partitioning of a two-dimensional space with non-uniform distribution and data block capacity $BC = 2$. The correspondence of partition numbers to letters is only for presentation. Internally, the partition numbers are stored as binary strings.

A leaf level entry in a G-tree consists of a partition number and a pointer to the block where the data points of this partition are stored. Figure 2 shows the internal and the leaf layer of the G-tree for the partitions of figure 1.

In Kumar (1994), the G-tree arithmetic is presented as well as algorithms for insertion, deletion and processing of range queries. Moreover, the advantages of the G-tree over similar data structures are examined.

This work discusses PerKMan, a new storage manager that makes database access from Prolog and aims at efficient manipulation of large amount of persistent knowledge. The



a -	00
b -	010
c -	011
d -	100
e -	10100
f -	10101000
g -	10101001
h -	10101010
i -	10101011
j -	1010110
k -	10101110
l -	10101111
m -	10110
n -	10111
o -	110
p -	11100
q -	11101
r -	1111

Figure 1. The partition of a two-dimensional space.

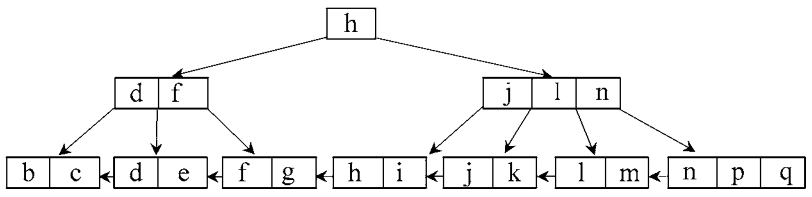


Figure 2. The G-tree for the partitions of figure 1.

rest of the paper is organized as follows: Section 2 deals with the syntax of predicates for the manipulation of persistent knowledge. Then, Section 3 presents the use of user-defined domains. Section 4 explains the data structures of the storage manager and Section 5 gives some formulae of its performance. Next, Section 6 exhibits experimental results and show that PerKMan achieves fast clause retrieval and good utilization of disk space. Section 6 concludes this work with a summary and a future research issue.

2. Syntax

PerKMan provides persistent storage of any size of knowledge and may be connected to Prolog systems offering an external C language interface. Throughout this paper, the word ‘Prolog’ stands for the ‘Edinburgh standard’. PerKMan is used for large volumes of clauses that are not possible to fit into main memory. Otherwise for smaller programs, normal Prolog should be used instead, because of the unnecessary overhead that PerKMan would introduce. The design motivation of PerKMan has been the organization of persistent clauses in a way that results in efficient update and retrieval operations.

The arguments of each permanent predicate belong to predefined domains and this cannot be changed dynamically at run time. The user provides the declaration of these arguments. User-defined domains are also declared, where they exist. From the Prolog point of view, the definition and manipulation of knowledge may be achieved through appropriate built-in predicates. These predicates have to be defined, using the external C language interface, in terms of the functions provided by PerKMan. In the following, we describe the proposed syntax only for few predicates due to space limitation.

PerKMan allows users to define custom domains built up from simple (*sdomain*) or complex (*cdomain*) domains. Domains are created with `cr_dom/2`. Its syntax is

$\texttt{cr\_dom}(\textit{cdomain}, \textit{domain}\ \{; \textit{domain}\})$
$\textit{domain} = \textit{cdomain} \mid \textit{sdomain} \mid \langle\textit{functor}\rangle\ (\textit{domain}\ \{, \textit{domain}\}) \mid \texttt{udom}.$

The symbol ; denotes disjunction. The universal domain `udom` incorporates any structure including lists. The meta-symbol | is used for the declaration of alternatives, the pair ⟨ ⟩ stands for 0 or 1 instances of the included term and the pair { } for 0 or more instances. A basic unit *sdomain* is one of the types `atom`, `integer` and `real`. The system grants as much space as needed for the storage of *sdomains*.

```
?- cr_dom(supp_name,supplier(atom,atom)).
yes.
```

Apart from storing the data as an unsorted sequence of clauses (heap organization), PerKMan supports the G-tree to store and retrieve clauses. A predicate definition is added to a knowledge base with `cr_pred/2`. Its syntax is

> `cr_pred`(*predicate*, ((*argument, domain*, y | n)
> {, (*argument*, *domain*, y | n)})).

The value `y(n)` means the participation (or not) of the argument in the index.

```
?- cr_pred(supplies, ((supplier,supp_name, y),
                      (product, atom,      y),
                      (price,   integer,   n)))
yes.
```

Although PerKMan does not exclude the definition of recursive domains, such as a list, it does not employ them in indices due to their unpredictable number of elements.

A knowledge base is queried through PerKMan either with set-oriented or clause-oriented operations. In clause-oriented operations further solutions are found through backtracking. The retrieval of clauses can be transparent if a permanent predicate `pred(X,Y,...)` is defined as `pred(X,Y,...):-sel_c(pred(X,Y,...))`. The predicate `sel_c/1` selects clauses in a clause-oriented mode.

```
?- sel_c(supplies(supplier(stafford,terry),Product,Price)).
Product = television
Price = 230    more? -- ;

Product = telephone
Price = 40    more? --
yes.
```

Hence, from the user's point of view, there is no difference between permanent (disk) and temporary (main memory) predicate access.

3. User-defined domains

In this Section, we deal with the use of User-Defined Domains (UDDs) built up from simple or complex domains. Appendices A and B contain the symbols in alphabetical order and their corresponding definitions used throughout this paper. Appendix B summarizes symbols used in formulae, whereas Appendix A includes the other symbols. Only for variables do we use italics. For predicates, clauses and code we use courier fonts.

We can distinguish between two kinds of UDDs: Non-Recursive Domains (NRD) and Recursive Domains (RD). The maximum number of elements for an argument defined on a NRD is known in advance, whereas an argument defined on a RD has an unpredictable number of elements. Compound terms or disjunction(s) of compound terms represent custom domains. A compound term consists of sub-terms. A functor may lead sub-terms. The following program udd_ex is an example of UDDs. The predicate ins_c/1 inserts clauses in a clause-oriented mode.

```
?- cr_dom(q,a(z,c)).
?- cr_dom(z,(d(atom);e(integer))).
?- cr_dom(c,(f(s,t);w(integer))).
?- cr_dom(s,s(integer)).
?- cr_dom(t,(l(atom);v(atom))).
?- cr_pred(pr,((name,q,y))).
?- ins_c(pr(a(d(atm1),f(s(9),v(atm2))))).
```

3.1. *Domain trees*

PerKMan flattens UDDs in order to handle them efficiently. Domain Trees (DTs) of UDDs include functors and sub-terms and helps understanding. Figure 3 shows the DT of q. We use dashed lines for disjunction and continuous for conjunction. Functors are inside ‘’.

DTs are unbalanced AND/OR-trees. Simple domains reside on the leaf nodes of DTs and the way we traverse them (the path) gives the form of the clauses. All possible paths are constructed by successive replacements of UDDs. Because the hierarchical structure of complex domains is flattened, they can be organized into G-trees. As an example, we decompose the UDD q of the program udd_ex. The symbols + and · denote disjunction and conjunction, respectively.

```
q=z.c
  =(atom+integer)·(s.t+integer)
  =(atom+integer)·(integer·(atom+atom)+integer)
  = atom·integer·atom+atom·integer·atom+atom·integer+
    integer·integer·atom+integer·integer·atom+integer·integer
```

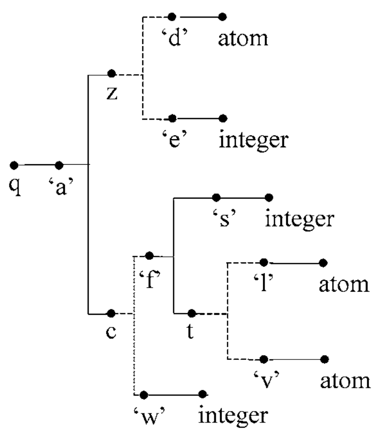


Figure 3. The tree of the user-defined domain q.

The six components of the last equation span the space of the alternative expressions and each one represents the ordered leaf nodes (simple domains) of a possible path.

3.2. Path identity

For the storage of clauses involving UDDs, PerKMan uses a prefix *PI* (Path Identity) that declares the correspondence between arguments and used terms. In other words, *PI* declares the path of the DT that corresponds to the selected terms. Its use is necessitated by the existence of disjunctions between sub-terms. We use the depth-first method to traverse a DT, because this method is both simple and efficient. If there are alternatives, we select a node according to a number that declares its position amongst the other nodes of the disjunction. A *PI* is a sequence of these numbers. The order of terms corresponds to the place of terms in DTs and therefore there is no confusion in the creation of *PIs*.

We examine the *PI* of the clause of the program `udd_ex`. The first choice occurs at node `z`. The `d(atom)` is chosen and thus 1 is the first element of *PI*. Next, at node `c`, we have the selection of `f(s,t)`, and so the second element of *PI* is 1. The last decision concerns node `t` and the selection of `v(atom)` corresponds to number 2. We have $PI = 1, 1, 2$.

The storage of a clause includes its *PI* and the used terms. Only the arguments that participate in indices are involved in *PIs*. The length of *PIs* is limited, for RDs are not included in indices.

4. Data structures

The data structures that PerKMan uses to organize persistent knowledge form four areas: User-Defined Domains (UDDA), Predicate Declarations (PDA), Index (IA) and Clauses (CA) Area. The first two are loaded into main memory when a knowledge base is opened, while the other two remain on disk. Figure 4 shows the above areas.

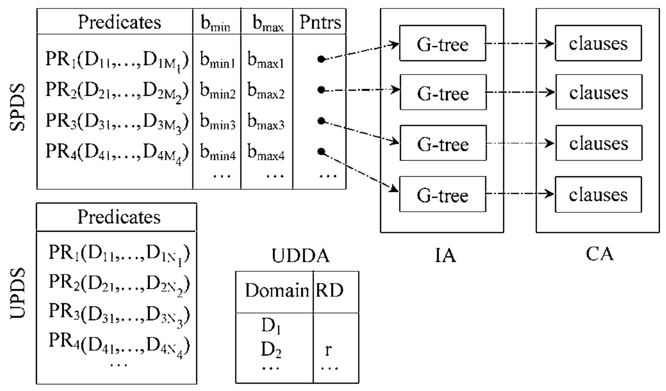


Figure 4. Data structures of PerKMan.

4.1. User-defined domains area

It includes the definitions of UDDs. An example of entries in the user-defined domains area is

```
supp_name = supplier(atom,atom).
expert_fields = fields(atom,expert_fields);atom.
```

UDDs and predicates are declared with the predicate `cr_dom/2` and `cr_pred/2`, respectively. Clauses are inserted with the predicate `ins_c/1`, e.g.,

```
?- cr_dom(expert_fields,(fields(atom,expert_fields);atom)).
?- cr_pred(experts,((exp_fld,expert_fields, n),
                    (name,   atom,          y))).
?- ins_c(experts(fields(databases,systems_analysis),peterson)).
```

The value `r` in the field RD of the user-defined domains area declares recursion in the corresponding domain. Recursions are found through DTs. From the above example only the entry for the UDD `expert_fields` is accomplished with the indicator `r`.

4.2. Predicate declarations area

It includes the predicate declarations as they are provided by the users and subsequently converted by the system. It is divided into the 'User Predicate Declarations Segment' (UPDS) and the 'System Predicate Declarations Segment' (SPDS).

UPDS: This contains for each persistent predicate, its name and arity, the domains of its arguments and the participation of each argument in the index. Users give the above declarations through `cr_pred/2`.

SPDS: For the persistent predicates it contains their names, arity, domains, the number of bits of the largest and the smallest partition ($b_{\min}$, $b_{\max}$) and the G-tree addresses in IA.

If at least one non-recursive UDD with compound term has been declared in the index of a predicate, then the predicate arity and domains in SPDS are different from the ones in UPDS. If a predicate PR_i has in its index at least one argument defined on a compound term, in general, it is $N_i \neq M_i$. In SPDS, the predicate dimensions are related to simple domains. When a clause is inserted, each of its arguments corresponds to the first non-occupied dimension of the index that has the same domain type.

4.3. Index area

The IA is composed of G-trees and is used for the fast retrieval of clauses.

Each G-tree corresponds to one predicate. This means that clauses are organized in parts according to the relation they implement. These parts are accessed easily because the addresses of G-tree roots are held in main memory. The above organization of predicates avoids accumulation in a big segment and speeds up the procedures of update and retrieval.

A G-tree dimension can be shared by two or more predicate arguments that belong to different paths. These arguments have to be of the same type, e.g., the argument with domain atom of the first path of the UDD `q` and the argument with the same domain type of the second path can share an index dimension with data type atom. This means that the required dimensions to index the domain `q` are integer, integer, atom and atom. Consequently, the predicate `pr` of the program `udd_ex` is declared in SPDS as,

```
pr(integer,integer,atom,atom).
```

If some G-tree dimensions are left without arguments (the opposite never happens), they take a default value from their domain. This does not influence G-trees, for they are adaptable structures. The number of the required dimensions to index a predicate is greater than or equal to the number of the attributes that participate in the index, and less than or equal to the number of the leaves of the DTs that the index includes. If the number of paths in a DT is n and there are k different simple domain types in it and l_{ij} is the number of i-domains in the j-path, $1 \leq i \leq k$, then the number of required G-tree dimensions with i-domain type is:

$$\max_{j=1,..,n} l_{ij}$$

The number of required dimensions to index a UDD is:

$$\sum_{i=1}^{k} \max_{j=1,..,n} l_{ij}$$

We examine the insertion of the clause of the program `udd_ex`. The first argument (atom) corresponds to the third dimension, which has the same type. Likewise, the second argument (integer) corresponds to the first dimension and the last argument (atom) to the fourth dimension. The second dimension takes the default value that is not necessary to be stored in the CA due to the existence of *PIs*. The clause is stored as '`1,1,2 (atm1,9,atm2)`'.

The retrieval procedure is analogous to the insertion one, e.g., the goal

```
?- pr(a(e(X),f(s(8),l(atm3)))).
```

triggers a search for clauses with $PI = 2, 1, 1$ and data `(_,8,atm3)`. When a query does not have an explicit functor declaration, it is replaced with the set of goals that corresponds to the paths of the DT, e.g., the query

```
?- pr(a(X,w(7))).
```

is analyzed into the goals `pr(a(d(Y),w(7))` and `pr(a(e(Y),w(7))`. The first corresponds to clauses with $PI = 1, 2$ and data `(_,7)` while the second to clauses with $PI = 2, 2$ and data `(_,7)`. The auxiliary variable `Y` does not appear in answers that have the form `X = d(atm4)` or `X = e(9)`. The most general query

```
?- pr(X).
```

is replaced by six calls.

The storage manager handles rules and facts in the same manner. Indexing the head of rules is achieved by inserting them into the G-tree. In order to achieve that, PerKMan relates the declaration of variables in rules head, since we only match with the head of rules, to the lower values of their domains. For integer and real numbers 'lower' means the minimum value that the variable could have, e.g., for integer it is −2147483647. For atoms the lower value is 'NULL'. Lower values are reserved by the manager and cannot be regarded as data. For example, to retrieve rules with head `goodprice(X,Y)`, the index is searched for the partition where the entry `goodprice(NULL,-2147483647)` belongs. The clauses block of this partition is accessed and rules like the `goodprice(X,Y):-supplies(_,X,Y),Y < 10` are found where they exist. Similarly, the rule `goodprice(radio,Y):-Y < 60` is stored in the clauses block of the partition of the clause `goodprice(radio,-2147483647)`. Rules in secondary storage are interpreted after their retrieval; that is, no compilation is needed at run time. Non-ground facts are treated as rules, e.g., the `goodprice(_,20)` is indexed as `goodprice(NULL,20)`. The lower values of complex domains are constructed from the lower values of the simple domains that reside in the leaf nodes of their domain tree.

In order to make things clearer, we consider the following example program that deals with courses attended by students in an informatics department. Courses are divided into two groups. A course may be compulsory or optional. All students take `databases` as the one of the compulsory courses. If a student has chosen the course `analysis_2`, then `analysis_1` is compulsory. The persistent predicate `st_cr/2` stores the courses of the students. The letters in the comments of the program are used for better presentation of clauses in figure 5. These are shown in italics.

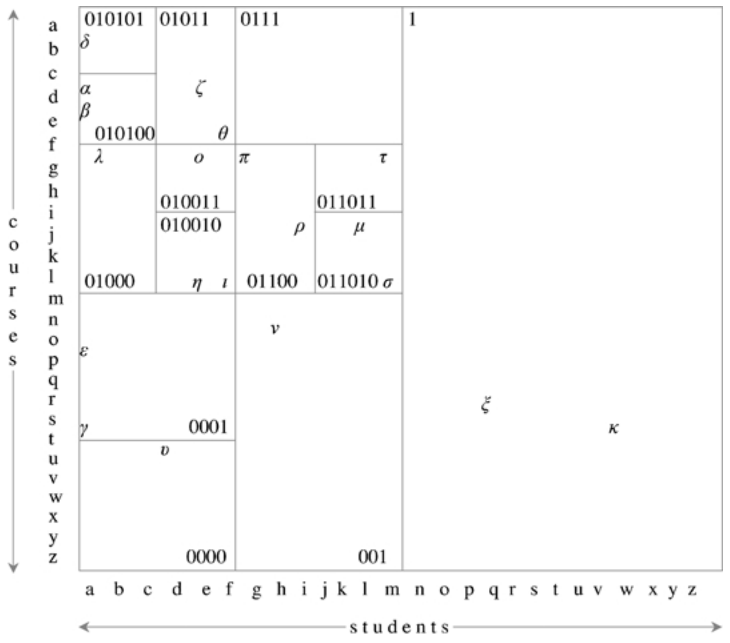


Figure 5. The partition scheme of the program `st_cr`.

```
?- cr_pred(st_cr, ((course,atom,y), (student,atom,y))).

?- ins_c(st_cr(_,compilers)).                                   % α
?- ins_c(st_cr(_,databases)).                                   % β
?- ins_c(st_cr(_,software_engineering)).                        % γ
?- ins_c((st_cr(X,analysis_1) :- st_cr(X,analysis_2))).         % δ
?- ins_c((st_cr(X,os_1) :- st_cr(X,os_2))).                     % ε
?- ins_c(st_cr(dimas,computer_networks)).                       % ζ
?- ins_c(st_cr(dimas,linear_algebra)).                          % η
?- ins_c(st_cr(eframidis,expert_systems)).                      % θ
?- ins_c(st_cr(eframidis,logic_design)).                        % ι
?- ins_c(st_cr(vassileiou,speech_processing)).                  % κ
?- ins_c(st_cr(alexiou,funcional_programming)).                 % λ
?- ins_c(st_cr(konstandinou,image_processing)).                 % μ
?- ins_c(st_cr(gregoriou,novel_architectures)).                 % ν
?- ins_c(st_cr(petrou,robotics)).                               % ξ
?- ins_c(st_cr(dimitriou,files_organization)).                  % ο
?- ins_c(st_cr(fotiou,files_organization)).                     % π
?- ins_c(st_cr(hatzis,image_processing)).                       % ρ
?- ins_c(st_cr(lazarou,logic_design)).                          % σ
?- ins_c(st_cr(lazarou,files_organization)).                    % τ
?- ins_c(st_cr(coutris,teory_of_linear_circuits)).              % υ
```

Figure 5 shows the partition scheme of the above example.

Figure 6 shows the G-tree for the partitions of the program `st_cr`.

In each recursive step of a rule application, PerKMan retrieves the first block that includes at least one matching clause. The first matching clause is used for the next step. Backtracking uses the second matching clause from the buffer and so on until all the matching clauses are exhausted. Then, a second matching block comes into main memory. We do not support the presentation of answers according to the insertion order of clauses, in order to avoid additional cost. Query answers are stored in buffers. If an answer cannot fit in buffers, it is cached and stored in a file. Garbage collection removes old answers. We use the LRU (Least Recently Used block) policy to replace the data that has not been referenced for the longest time.

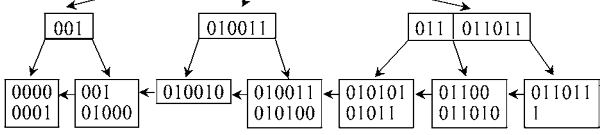


Figure 6. The G-tree for the partitions of figure 5.

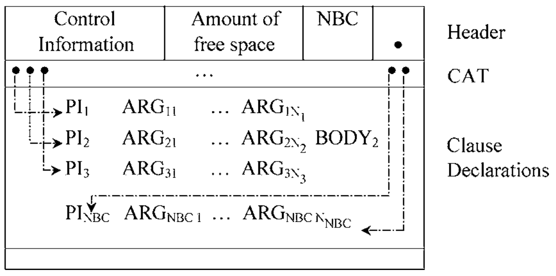


Figure 7. A clauses block structure.

4.4. Clauses area

The clauses of all persistent predicates compose the CA. Each block of clauses is composed of a header, a clause allocation table (CAT), the clause declarations and the free space. The header includes control information, the amount of free space, the number of clauses in the block (*NBC*) and one pointer that is used to connect blocks in case of overflow. A full block may overflow when its clauses have their arguments that participate in the index identical. This means that the block cannot be split. This may occur when the index includes only attributes that do not identify its predicate or there are many rules with variables in the same indexing arguments. The size of a clause cannot be larger than the size of its block.

A CAT contains $NBC + 1$ pointers. The first *NBC* pointers indicate the beginning of clauses whereas the last one indicates the end of the last clause. A clause declaration consists of the prefix *PI*, the arguments ARG_{ij}, $1 \leq i \leq NBC$, $1 \leq j \leq N_k$, $1 \leq k \leq NBC$ and the $BODY_i$, when the clause is a rule. The use of CAT is necessary due to the variable length of clauses. The order of clauses within a block is not significant. Figure 7 shows a block of clauses.

A split occurs when the free space cannot accommodate a new clause. The splitting point is fixed, as the attribute domains are split in the middle. For integer and real numbers, the splitting point is the half of the sum of the upper and lower value of the considered interval. For atoms, we use an algorithm that transforms them to unsigned long integers. This makes possible an arithmetic comparison between arguments and splitting points for all data types. There is a small possibility for two different atoms to be transformed to the same number and a slight possibility not to find a dimension to split due to identical transformations. If this happens, we use overflow blocks.

5. Cost formulae

In this Section we present some formulae for clauses insertion, deletion and simple queries on predicates organized as G-trees. The creation of general formulae presents difficulties due to the existence of the many parameters and complex rules.

5.1. Clause insertion

The index is searched for P_L and if it does not exist, the search continues to find an ancestor of P_L (up to $P_{L'}$), moving backward in the leaf level. When the proper partition is found, the corresponding clause block is accessed. If there is room, the clause is inserted in it. Otherwise, we have a case of overflow and we split the partition. The old partition number is deleted from the index, whereas the new partitions are inserted into it. If the partition of the inserted clause does not exist in the index, then it is inserted into it and the clause is stored into a new block. Let *PF* be the probability that a block cannot store any more clauses. The insertion cost is

$$\begin{aligned} PBAI &= h\,rba + (NPB - 1)sba + PE * (1rba + (1 - PF)\,sba \\ &\quad + PF * ([delete\_one\_partition] + 2 * [insert\_one\_partition] + 2\,rba + 1sba)) \\ &\quad + (1 - PE) * ([insert\_one\_partition] + 1rba) \\ &= (h + 1 + 2 * PE * PF)rba + (NPB - 1 + PE)\,sba \\ &\quad + PE * PF * [delete\_one\_partition] \\ &\quad + (2 * PE * PF - PE + 1) * \ [insert\_one\_partition] \end{aligned}$$

Due to the possibility of successive splits or merges in the index it is

$$\begin{aligned} 1sba &\le [delete\_one\_partition] \le (2 * h - 1)rba \\ 1sba &\le [insert\_one\_partition] \le (2 * h + 1)rba \end{aligned}$$

we have

$$\begin{aligned} PBAI &\ge (h + 1 + 2 * PF * PE)rba + (NPB + 3 * PF * PE)sba \\ PBAI &\le (h + 1 + 2 * PE * PF)rba + (NPB - 1 + PE)sba \\ &\quad + (2 * h - 1) * PE * PFrba + (2 * PE * PF - PE + 1) * 2 * (h + 1)rba \\ &= (3h + 2 + 3 * PE * PF * (2 * h + 1) - PE * (2 * h + 1))\,rba \\ &\quad + (NPB - 1 + PE)sba \end{aligned}$$

5.2. Clause deletion

A clause deletion requires the following steps. The proper partition, say P, where the clause may exist, is obtained as happens in case of insertion. Then the corresponding clauses block is accessed. If after the deletion of a clause, the block does not underflow, the procedure stops. Otherwise, P and its complement are merged into their parent and the index is updated. If the complement partition is empty and, thus, it does not exist in the index, the merge is propagated upwards. *PU* is the underflow probability. We have

$$\begin{aligned} PBAD &= h\,rba + (NPB - 1)sba + 1rba + (1 - PU)sba \\ &\quad + PU * (2 * [delete\_one\_partition] + [insert\_one\_partition] + 2rba) \end{aligned}$$

From the formulae of the previous Subsection, we have

$$\begin{aligned} PBAD &\geq (h + 1 + 2 * PU)rba + (NPB + 2 * PU)sba \\ PBAD &\leq (h + 1 + (6 * h + 1) * PU)rba + (NPB - PU)sba \end{aligned}$$

5.3. *Queries without variables*

The answer to a query without variables is ‘true’ or ‘false’. We consider a query referring to a single predicate. *PBANV* is the physical block access cost of a query without variables. If *PBANVF* is the cost of searching the predicate facts, $PBANVR_q$ is the corresponding cost of the qth rule of the predicate and r is the number of rules in the query, then

$$PBANV = PBANVF + \sum_{q=1}^{r} PBANVR_q$$

The number of the descendants in a G-tree node depends on the size of the partition numbers. Because this size is variable, we can only deal with the average order m of a G-tree. If *PNS*, *PRS* and *BS* are the average size of partition numbers, the pointer size and the block size in bytes respectively, we have

$$(m - 1) * PNS + m * PRS \leq BS$$

Since the order m is the maximum integer in the above inequality, we have

$$m = \left\lfloor \frac{BS + PNS}{PNS + PRS} \right\rfloor$$

h is the height of the G-tree and p is the number of partitions that have to be accessed in the leaf level. This number is at least one (P_H) and at most all entries in $[P_{L'}, P_H]$. *NPB* is the number of blocks occupied by the above partitions and *NP* is the number of partitions in the leaf level of the G-tree. We have

$$2 * (\lceil m/2 \rceil - 1) * \lceil m/2 \rceil^{h-2} \leq NP \leq (m - 1) * m^{h-1}$$

or

$$1 + \log_m \frac{NP}{m - 1} \leq h \leq 2 + \log_{\lceil m/2 \rceil} \frac{NP}{2 * (\lceil m/2 \rceil - 1)}$$

Because h is an integer, we have

$$\left\lceil \log_m \frac{NP}{m - 1} \right\rceil + 1 \leq h \leq \left\lfloor \log_{\lceil m/2 \rceil} \frac{NP}{2 * (\lceil m/2 \rceil - 1)} \right\rfloor + 2$$

In addition, because partition numbers are stored in a B$^+$-tree like organization, we have

$$\left\lceil \frac{p}{m-1} \right\rceil \leq NPB \leq \left\lceil \frac{p}{\lceil m/2 \rceil - 1} \right\rceil$$

The first access to clauses blocks is random (rba) and the remaining sequential (sba). *h* rba are needed to traverse top-down the tree. The leaf level of the tree is accessed sequentially as it is kept in a contiguous area. *PE* reflects the probability that the partition of the searched fact exists in the index. It is

$$\begin{aligned} PBANVF &= h\, rba + (NPB - 1) sba + PE\, rba \\ &= (h + PE) rba + (NPB - 1) sba \end{aligned}$$

The evaluation of $PBANVR_q$ depends on the rules form and complexity. We examine the simple case where in the body of a rule there are no other variables except those existing in its head. If the body of a rule consists of *k* conjunctions, we have

$$\begin{aligned} PBANVR_q &= PBANV_{q0} + PBANV_{q_1} + PE_{q_1} * PBANV_{q2} \\ &\quad + PE_{q_1} * PE_{q_2} * PBANV_{q3} + \cdots + PE_{q_1} * \cdots * PE_{q_{k-1}} * PBANV_{qk} \\ &= PBANV_{q0} + PBANV_{q_1} + \sum_{i=2}^{k} PBANV_{q_i} * \prod_{j=1}^{i-1} PE_{q_j} \end{aligned}$$

$PBANV_{q0}$ is the cost to find the qth rule, $PBANV_{q_i}$, $1 \leq i \leq k$, is the cost to match the ith sub-goal of its body and PE_{q_j}, $1 \leq j \leq k-1$, the probability of a match for the jth sub-goal of the qth rule. Recalling that variables are related to the lower values of their domain, we have

$$PBANV = PBANVF + \sum_{q=1}^{r} \left(PBANV_{q0} + PBANV_{q_1} + \sum_{i=2}^{k_q} PBANV_{q_i} * \prod_{j=1}^{i-1} PE_{q_j} \right)$$

5.4. Queries with variables

Let *PBAV* be the cost to find all the answers. If a query refers to one predicate that has only facts, we have

$$hrba + (NPB - 1) sba \leq PBAV \leq (h + p) rba + (NPB - 1) sba$$

The lower bound of the above formulae is reached when none of the *p* partitions overlaps with the query region, and the upper bound when all of them overlap with the query region.

6. Experimental results

In this Section we provide experimental results on the performance of PerKMan and compare them to the corresponding performance of *ECL^{i}PS^e*.

ECL^{i}PS^e is a Prolog-based system, whose aim is to provide a platform for integrating various extensions of logic programming. One of these extensions is the persistent storage of clauses through the DB and the KB module (ECL^{i}PS^e 3.7, 1998). The BANG file (Freeston, 1987), a multiattribute indexing structure that is a variant of the BD-tree, is used in both modules to index on the attributes that the user indicates. In the DB (DataBase) module, attributes of type term cannot be included in the index. This module does not manipulate rules and non-ground facts. The KB (Knowledge Base) module supports the persistent storage of any clause. The storage of terms does not need any type of size declaration. ECL^{i}PS^{e}allocates space as required. The KB version is less efficient than the DB version and the second should be preferred when possible. None of the modules supports the coexistence of DB and KB relations.

The statistics of ECL^{i}PS^{e}inform us about the size of the page buffer area for the DB handling, the pages of the relations that are currently in buffers, the real I/O and buffer access. This allows a comparison between the access efficiency of ECL^{i}PS^e and PerKMan, on the base of disk reads. We present experiments with four dimensions, all included in the index and attribute size of 4 bytes. We used 8 Kbytes page size because this is the default for the BANG file in ECL^{i}PS^e. The data followed the normal distribution with mean value 0 and standard deviation $5 * 10^6$. We choose to present our experiments with data following the normal distribution because as well as the fact that this distribution is common in real world measurements, it approximates many other distributions well. We used non-duplicate facts. Their range was $[10^5, 2 * 10^6]$ and the step of increment 10^5. Similar experiments with other distributions showed that results depend very slightly upon the nature of the distribution from which the data is drawn. Our implementation was made in C and the performance comparison on a SUN4u Sparc Ultra 5/10 under SunOS 5.8.

Figure 8 shows the total insertion time in minutes versus the number of facts. We repeated the insertion procedure three times in a dedicated machine. We present the average insertion

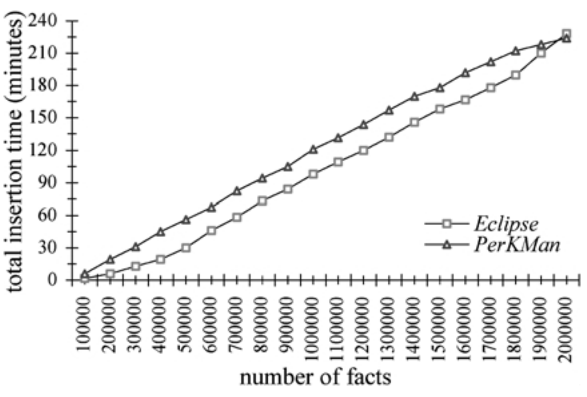


Figure 8. Total insertion time versus number of facts.

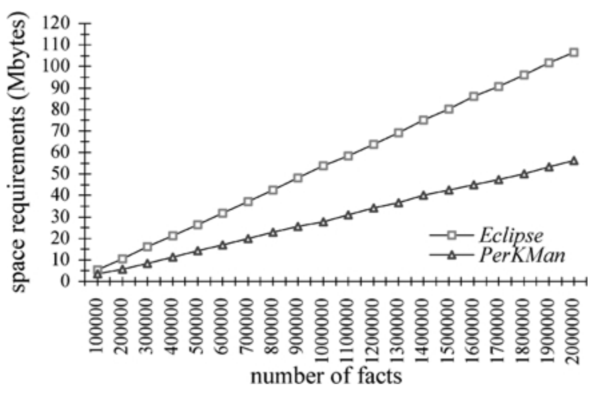


Figure 9. Space requirements versus number of facts.

times. The insertion time of PerKMan becomes lower than the one of ECL^{i}PS^e in a volume of $2 * 10^6$ facts.

Figure 9 shows the space requirements of the two systems in Mbytes compared with the number of facts. We present the total storage space, as ECL^{i}PS^e does not inform us about the storage space of index and data separately. As shown in this figure, PerKMan needs much smaller storage space than ECL^{i}PS^e to organize its data.

The following results correspond to the average disk block accesses using 1000 queries of the same type. The queries are taken uniformly from the insertion file. That is, the constant values of a partial match query over a file of *NC* clauses were taken from the places $\lfloor NC/1000\rfloor * j$, $1 \leq j \leq 1000$, of the insertion file. We decided to take average values from 1000 queries in order to have more accurate results. The average value of disk accesses from a high number of queries reduces substantially the possibility of obtaining decreasing segments in the curve of disk accesses versus the number of facts. Such segments are justified by the fact that queries were taken uniformly from the insertion file and, consequently, the queried facts were not the same for all steps. The insertion file consists of facts in order to have a comparison with ECL^{i}PS^e that provides statistics for the DB handling.

Figure 10 shows that the two systems need the same number of disk access for exact match queries. An example of these queries is

```
?- pr(874608,595741,-1371850,-1115738).
```

Figures 11–13 present the accesses for partial match queries with one, two and three variables, respectively. Examples of these queries are the following:

```
?- pr(X,595741,-1371850,-1115738).
?- pr(X,Y,-1371850,-1115738).
?- pr(X,Y,Z,-1115738).
```

For PerKMan there are two curves that represent the number of accesses to find the first and all matching facts. ECL^{i}PS^e statistics give us the same number of accesses for both

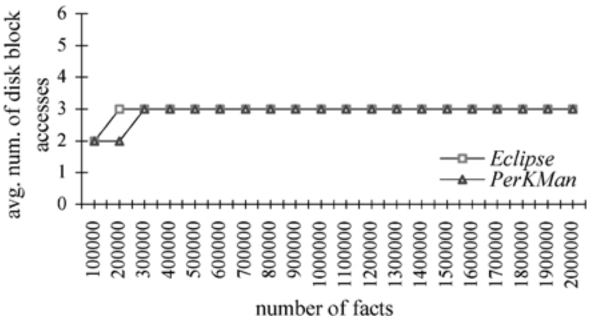


Figure 10. Average number of disk block accesses per exact match query versus number of facts.

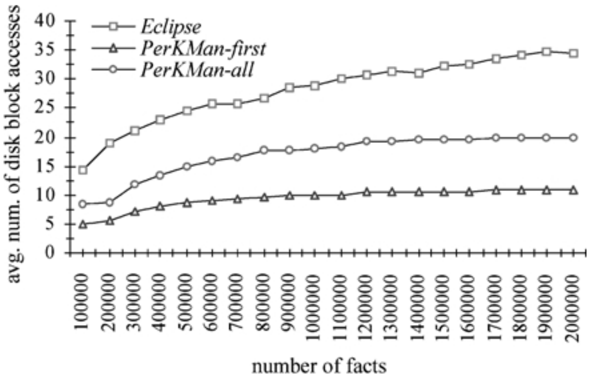


Figure 11. Average number of disk block accesses per partial match query of one variable versus number of facts.

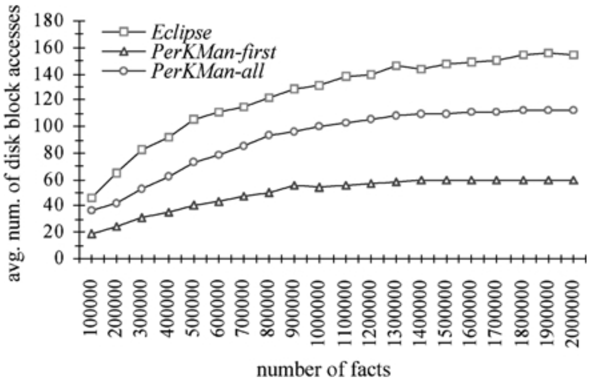


Figure 12. Average number of disk block accesses per partial match query of two variables versus number of facts.

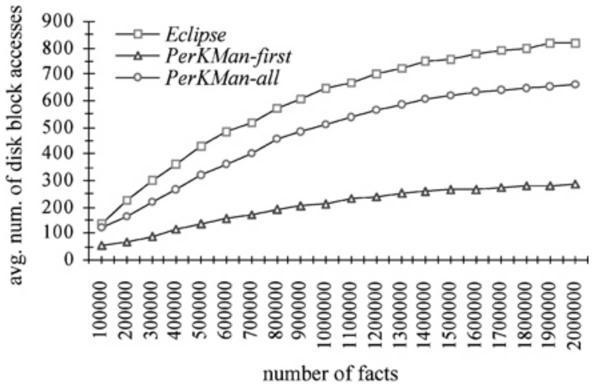


Figure 13. Average number of disk block accesses per partial match query of three variables versus number of facts.

cases. The figures show that the disk accesses required by PerKMan are less than the ones required by ECL^{i}PS^e. This difference is due to the underlying indexing schemes of the two systems.

Figure 14 presents the average elapsed query time in milliseconds for the exact match queries of figure 10. Figures 15–17 present the average elapsed query time in milliseconds required to obtain all the answers for the partial match queries of the figures 11–13, respectively. During the experiments, the machine was dedicated. These figures indicate that the efficiency of PerKMan, in terms of elapsed query time, increases with larger numbers of facts. We can also see that for large volumes of data, the elapsed time of PerKMan is much less than that of ECL^{i}PS^e, as far as its comparison with the number of disk accesses is concerned. We also notice that there is no identical behaviour of the curves of block accesses and their coresponding processing time. This is justified by the fact that elapsed time is sensitive to many more factors than disk accesses (implementation, replacement policy, etc.).

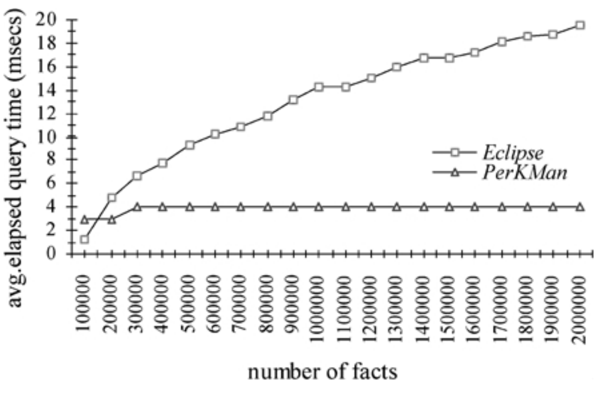


Figure 14. Average elapsed query time per exact match query versus number of facts.

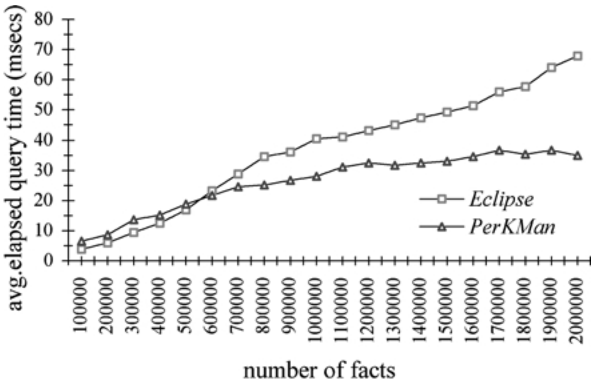


Figure 15. Average elapsed query time per partial match query of one variable versus number of facts.

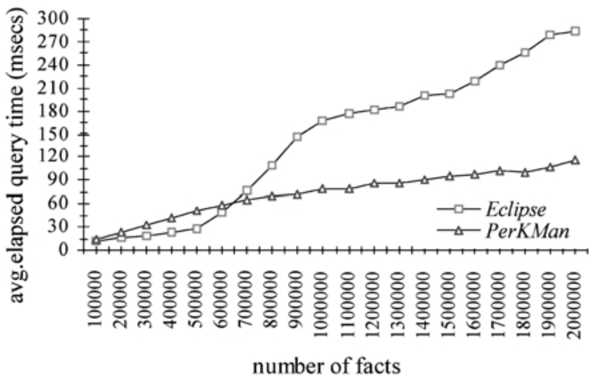


Figure 16. Average elapsed query time per partial match query of two variables versus number of facts.

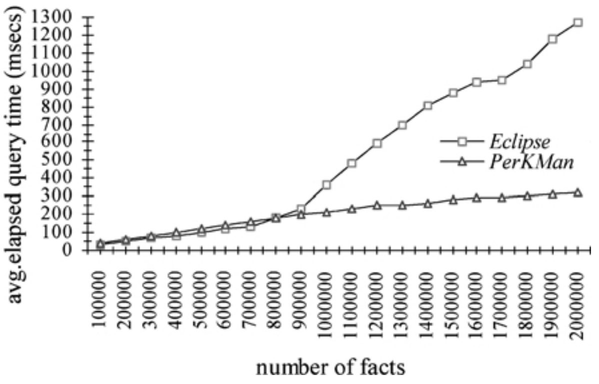


Figure 17. Average elapsed query time per partial match query of three variables versus number of facts.

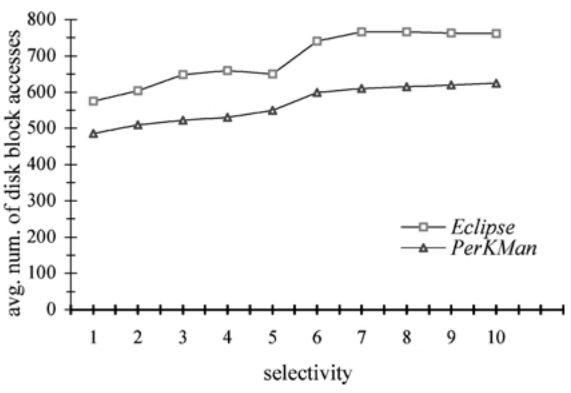


Figure 18. Average number of disk block accesses per partial match query of three variables versus selectivity.

The selectivity of the above exact match and partial match queries was low (no more than 5 facts), because data followed the normal distribution with a high value of standard deviation. Experiments with higher values of selectivity showed that PerKMan keeps the amount of saving accesses shown in previous figures, for all the selectivity values. In figure 18, we present the number of physical block accesses versus selectivity for partial match queries with three variables. We use a file of 10^6 facts. The selectivity varies from 1% to 10% of the insertion file.

We proceed with a discussion with regard to the indexing schemes of PerKMan and ECL^{i}PS^e that justifies, theoretically, the results shown in the above diagrams.

The G-tree maps a partition directly into a block and avoids the complex solution, where blocks are defined by including some and excluding other sub-partitions from a partition, as does the BANG file. The partition algorithm of the BANG file is more complex than that of the G-tree. It uses a recursive procedure to achieve the best balance of the regions resulting from the partition of a logical region (a block region minus the block regions it encloses). Neither the partition scheme nor the recombination of a region is unique. The first depends on the order in which data is added, and for the second, algorithms provide the best recombination. The partition scheme of the G-tree, on the other hand, does not depend on the order of insertions and deletions. Moreover, when a partition becomes empty, it is recombined efficiently with its complement.

The directory maintenance of the BANG file is more expensive than that of the G-tree. When a new entry is added to the directory of the BANG file due to the split of a logical region, it is possible for a logical region which does not span its enclosed space to exist in an upper directory level. To avoid this problem, the logical region is forced to split before it overflows, but the union of the regions which it encloses may not span the new logical region. This situation may be propagated recursively up to the leaf level (data pages). The splitting procedure incorporates an algorithm that ensures the minimum size of regions after the split. The existence of this algorithm is essential because it avoids the creation of empty directory pages. The maintenance of the G-tree consists of insertions and deletions of partition numbers.

Our experimental results verified the cost formulae of Section 5. In the following, we examine the cost formulae with regard to a file of 10^6 facts, which took part in our experiments.

For the insertion cost we had $h = 2$, $NPB = 1$. As the data block capacity was 511 facts and the utilization of data blocks 68%, we have $PF = \frac{100}{511*68} = 0.0029$. The capacity and the utilization of the leaf nodes of G-tree were found to be 628 and 70%, respectively. For the last 10000 facts of the insertion file the G-tree created 30 new partition numbers. This means that the probability for a partition to exist is $PE = 1 - \frac{30}{10000} = 0.997$. Replacing the variables in the last two inequalities of Subsection 5.1 we have $4.014 \le PBAI \le 4.055$. From the experiments we found that on average the insertion cost of the last 10000 facts of the insertion file was 4.021 accesses. This value falls between the upper and lower limit of the above inequality.

We proceed with the estimation of the deletion cost. Based on the above values of data block capacity and utilization and because an underflow occurs when utilization falls under 50%, we estimate the underflow probability as

$$PU = \frac{1}{\left(511 * \frac{68}{100} - \frac{511}{2}\right) + 1} = 0.01076$$

As we have $h = 2$ and $NPB = 1$, the inequalities of Subsection 5.2 give, $4.043 \le PBAD \le 4.129$. For the deletion of the first 10000 facts from the file of 10^6 facts we needed on average 4.07 block accesses. This value is compatible with the above inequality.

As we can see from the above application of formulae, both the insertion and deletion inequalities approximate very tightly to the corresponding costs.

Next, we examine some formulae of Subsection 5.3. For the file of 10^6 facts the G-tree created $NP = 3112$ partition numbers with average length $PNS = 17$ bits, that is 2.125 bytes. Since $PRS = 4$ and $BS = 8192$, we have

$$m = \left\lfloor \frac{8192 + 2.125}{2.125 + 4} \right\rfloor = 1337$$

From the formula that estimates the height of the G-tree, we have

$$\left\lceil \log_{1337} \frac{3112}{1336} \right\rceil + 1 \le h \le \left\lfloor \log_{669} \frac{3112}{2 * (669 - 1)} \right\rfloor + 2$$

or

$$2 \le h \le 2.$$

This estimates the height of the G-tree properly, because for 10^6 facts it is $h = 2$.

On average for 1000 queries we had $p = 192$. Thus,

$$\left\lceil \frac{192}{1336} \right\rceil \le NPB \le \left\lceil \frac{192}{669 - 1} \right\rceil$$

or

$$1 \leq NPB \leq 1$$

The value of PE is the same as in the insertion cost, as queried facts were taken from the insertion file. Thus, $PBANVF = 2 + 0.997 + 1 - 1 = 2.997$. This value is very close to that of figure 10 (exact match query versus 10^6 facts), which is 3.

For a partial match query with one variable we had $h = 2$, $NPB = 2$, $p = 25$ and 17 disk accesses. These values are compatible with the formulae of Subsection 5.4 as, $3 \leq 17 \leq 28$.

7. Summary

The increasing need for large knowledge bases and efficient handling of non-ad hoc queries implies the adoption of effective data structures. We presented PerKMan, a storage manager that may be connected to Prolog systems that offer an external C language interface. This means that in order for PerKMan to be usable, an intermediate interface has to be developed that connects the host Prolog system with the functions provided by the storage manager. PerKMan handles facts and rules uniformly and allows different arguments of a predicate to share an index dimension in a novel manner. It indexes compound terms efficiently and its data structures are not only independent of the data distribution but also adapts well to dynamic large volumes of clauses. We presented cost formulae and reported experimental results.

In order to find all answers in partial match queries with one, two and three variables in four-dimensional predicates, PerKMan achieves 42%, 28% and 19%, respectively, savings on average in disk block accesses compared to the indexing method of ECL^{i}PS^e. Moreover, the elapsed query time of PerKMan is increasingly less, versus the number of data, compared to that of ECL^{i}PS^e. From the performance of PerKMan, we believe that it achieves its design motivation, which is to handle efficiently large quantities of persistent knowledge.

Work in progress includes sophisticated methods that relate rules to data on a scheme that is based on the distribution of query types.

Appendix A: Symbols and definitions

Symbol	Definition
BC	Capacity of data block
$b_{\max}$	Number of bits of the smallest partition
$b_{\min}$	Number of bits of the largest partition
CA	Clauses area
CAT	Clause allocation table
cdomain	Complex domain
`cr_dom/2`	Creates domains
`cr_pred/2`	Creates predicates

(*Continued on next page*.)

(*Continued*).

Symbol	Definition
DT	Domain tree
IA	Index area
`ins_c/1`	Inserts clauses in a clause-oriented mode
LRU	Least recently used
NBC	Number of clauses in a block
NC	Number of predicate clauses
NRD	Non-recursive domain
P	Partition
PDA	Predicate declarations area
PI	Path identity
RD	Recursive domain
sdomain	Simple domain
`sel_c/1`	Selects clauses in a clause-oriented mode
SPDS	System predicate declarations segment
UDD	User-defined domain
UDDA	User-defined domains area
`udom`	Universal domain
UPDS	User predicate declarations segment

Appendix B: Symbols and definitions in formulae

Symbol	Definition
BS	Block size in bytes
h	G-tree height
m	G-tree average order
NP	Number of partitions in the leaf level of the G-tree
NPB	Number of blocks occupied by p partition numbers
p	Number of searched partition numbers
$PBAD$	Physical block accesses to delete a clause
$PBAI$	Physical block accesses to insert a clause
$PBANV$	Physical block accesses for a query without variables
$PBANVF$	Physical block accesses for retrieving facts in a query without variables
$PBANVR_q$	Physical block accesses for the qth rule execution in a query without variables
$PBANVR_{q_0}$	Physical block accesses to find the qth rule

(*Continued on next page.*)

(*Continued*).

Symbol	Definition
$PBANVR_{q_i}$	Physical block accesses to match the ith sub-goal of the rule body
$PBAV$	Physical block accesses for a query with variables
PE	Probability for a partition to exist in a G-tree
PE_{qJ}	Probability of a match for the jth sub-goal of the qth rule
PF	Probability for a block to be full
P_{H}	$b_{\max}$-bit long partition number of the rightmost point of a query
P_{L}	$b_{\max}$-bit long partition number of the leftmost point of a query
$P_{\mathrm{L}'}$	Ancestor of $\mathrm{P_L}$with length at least $b_{\min}$-bit, existing in a G-tree
PNS	Average partition numbers size in bytes
PRS	Pointer size in bytes
PU	Probability for a block to underflow
rba	Random block access
sba	Sequential block access

References

Cruickshank, G. (1994). Persistent Storage Interface for Prolog—TERMdb System Documentation. Draft Revision:1.5, Cray Systems.

Derr, M.A., Morishita, S., and Phipps, G. (1994). The Glue-Nail Deductive Database System: Design, Implementation, and Evaluation. *The VLDB Journal*, 3(2), 123–160.

ECL^{i}PS^e 3.7 (1998). Knowledge Base User Manual. ECRC GmbH.

Emde Boas, G. and Emde Boas, P. (1986). Storing and Evaluating Horn-Clause Rules in a Relational Database. *IBM J Res. Develop.*, 30(1), 80–92.

Freeston, M. (1987). The BANG File: A New Kind of Grid File. In *Proc. ACM SIGMOD Conf.* (pp. 260–269).

Ioannidis, Y.E. and Tsangaris, M. (1994). The Design, Implementation, and Performance Evaluation of BERMUDA. *IEEE Trans. on Knowledge and Data Eng.*, 6(1), 38–56.

Kapopoulos, D.G. and Hatzopoulos, M. (1999). The G-Tree: The Use of Active Regions in G-Trees. In *Proc. 3rd Intern. Conf. on Advances in Databases and Information Systems*, Maribor (pp. 141–155).

Kumar, A. (1994). G-Tree: A New Data Structure for Organising Multidimensional Data. *IEEE Trans. on Knowledge and Data Eng.*, 6(2), 341–347.

Nilsson, H. and Ellemtel, A. (1995). The External Storage Facility in SICStus Prolog. R:91:13, Swedish Institute of Computer Science.

Ramakrishnan, R., Srivastava D., Sudarshan, S., and Seshadri, P. (1994). The CORAL Deductive System. *The VLDB Journal*, 3(2), 161–210.

Ramamohanarao, K. , Shepherd, J., Balbin, I., Port, G., Naish, L., Thom, J., Zobel, J., and Dart, P. (1988). The NU-Prolog Deductive Database System. In P.M.D. Gray and R.J. Lucas (Eds.), *Prolog and Databases*, Reading, West Sussex: Ellis Horwood.

Sagonas, K., Swift, T. and Warren, D.S. (1994). XSB as an Efficient Deductive Database Engine. In *Proc. of ACM SIGMOD Conf.* (pp. 442–453).

Vaghani, J., Ramamohanarao, K., Kemp, D.B., Somogyi, Z., Stuckey, P.J., Leask, T.S., and Harland, J. (1994). The Aditi Deductive Database System. *The VLDB Journal*, 3(2), 245–288.