An Efficient Storage Manager

Dimitris G. Kapopoulos, Michael Hatzopoulos, and Panagiotis Stamatopoulos

Department of Informatics, University of Athens,
Panepistimiopolis, Ilisia 157 84, Greece
{dkapo, mike, takis}@di.uoa.gr

Abstract. When dealing with large quantities of clauses, the use of persistent knowledge is inevitable, and indexing methods are essential to answer queries efficiently. We introduce PerKMan, a storage manager that uses G-trees and aims at efficient manipulation of large amount 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 their distribution. The storage manager achieves fast clause retrieval and reasonable use of disk space.

1 Introduction

Efficient management of persistent knowledge in deductive database systems requires the adoption of effective indexing schemes in order to save disk accesses, whilst maintaining reasonable use of available space. *Deductive database systems* incorporate the functionality of both logic programming and database systems. As referred to [6], they have four major architectures: *logic programming systems enhanced with database functionality* (NU-Prolog [11]), *database access from Prolog* (BERMUDA [6], TERMdb [1]), *relational database systems enhanced with inferential capabilities* (Business System 12 [4]), and *systems from scratch* (SICStus [9], CORAL [10], Aditi [13], Glue-Nail [2], XSB [12], ECL^{i}PS^e [3]).

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.

The *G-tree* [8] is an adaptable multidimensional structure that combines the features of B-trees and grid files. It divides the data space into a grid of variable size partitions and adapts its shape to high dynamic data spaces and to non-uniformly distributed data. Only non-empty partitions are stored in a B-tree-like organization. The G-tree uses a variable-length partition numbering scheme. Each partition is assigned a unique binary string of 0's and 1's. In [8], the G-tree arithmetic, algorithms for update and search operations and the advantages of the G-tree over similar data structures are examined. The *G^r_tree* [7] combines the features of metric spaces and G-trees. It considers every data space as a metric space through the use of the Euclidean norm and an algorithm that transforms strings to unsigned long integers.

J. Štuller et al. (Eds.): ADBIS-DASFAA 2000, LNCS 1884, pp. 331–338, 2000.

Although the G^r_tree requires distance computations and has the overhead of a small amount of storage space, due to the introduction of active regions inside the partitions of the data space, it reduces the accesses of partial match and range queries.

This work introduces PerKMan, a new storage manager that uses G^r_trees, makes database access from Prolog and aims at efficient manipulation of large amount of persistent knowledge. The rest of the paper is organized as follows: Section 2 deals with the use of user-defined domains. Section 3 explains the data structures of the storage manager. Section 4 gives experimental results and shows that PerKMan achieves fast clause retrieval and good utilization of disk space. Section 5 concludes this work with a summary and a future research issue.

2 User-Defined Domains

PerKMan provides persistent storage of any size of knowledge and may be connected to Prolog systems that offer an external C language interface. The arguments of each permanent predicate belong to predefined domains and this cannot be changed at run time. From the Prolog point of view, the definition and manipulation of knowledge may be achieved through appropriate built-in predicates that have to be defined, using the external C language interface, in terms of the functions provided by PerKMan.

User-Defined (or *custom*) *Domains (UDDs)* are built up from simple (*sdomain*) or complex (*cdomain*) domains. Domains are created with `cr_dom/2`. Its syntax is

`cr_dom`(*cdomain*, *domain* {; *domain*})

domain = *cdomain* |*sdomain* |⟨*functor*⟩ (*domain* {, *domain*}) |*udom*.

The universal domain *udom* incorporates any structure including lists. A basic unit *sdomain* is one of the types atom, integer and real.

Apart from storing the data as an unsorted sequence of clauses (heap organization), PerKMan supports the G^r_tree to store and retrieve clauses. A predicate definition is added to a knowledge base with `cr_pred/3`. 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.

UDDs are distinguished between Non-Recursive Domains (NRD) and Recursive Domains (RD). PerKMan does not employ RDs in indices due to their unpredictable number of elements. The following program `udd_ex` is an example of UDDs.

```
?- 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)), b(300)).
?- ins_c(pr(a(d(atm1),f(s(9),v(atm2))))).
```

A knowledge base is queried through PerKMan either with set-oriented or clause-oriented operations. Clauses retrieval can be transparent if a permanent predicate `p(X,..)` is defined as `p(X,..):-sel_c(p(X,..))`. The predicates `sel_c/1` and `ins_c/1` selects and inserts clauses in a clause-oriented mode respectively.

PerKMan flattens UDDs in order to handle them efficiently. *Domain Trees (DTs)* of UDDs include functors and sub-terms and assists understanding. Figure 1 shows the DT of q. We use dashed lines for disjunction and continuous for conjunction.

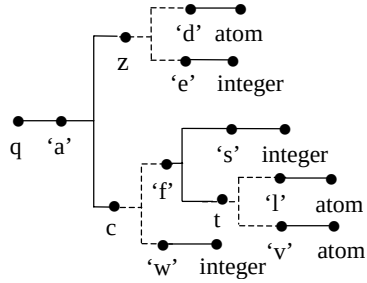

Fig. 1. The tree of the user-defined domain q.

DTs are unbalanced AND/OR-trees. *sdomains* reside on the leaf nodes of DTs and the way we traverse them gives the form of the clauses. Possible paths are constructed by successive replacements of UDDs. Because the hierarchical structure of *cdomains* is flattened, they can be organized into G^r_trees. As an example, we decompose the UDD q. The symbols + and · denote disjunction and conjunction respectively.

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

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

For the storage of clauses involving UDDs, we use a *Path Identity* (*PI*) prefix that declares the correspondence between arguments and used terms. In other words, *PI* is 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. If there are alternatives, we select a node according to a number that declares its position among the other nodes of the disjunction. 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.

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 select the 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.

3 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, whilst the other two remain on disk. Figure 2 shows the above areas.

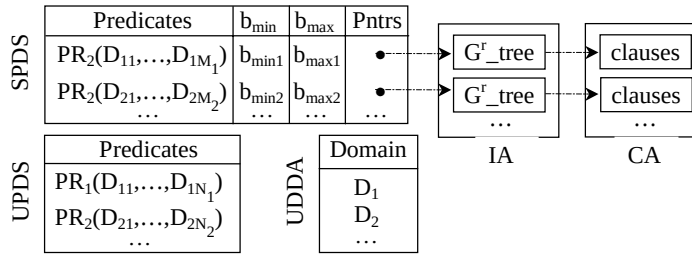

Fig. 2. Data structures of PerKMan.

3.1 Predicate Declarations Area

It includes in two segments the predicate declarations as they are provided by the users (UPDS) and subsequently converted by the system (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.
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 addresses in IA.

If at least one UDD with *cdomain* has been declared in the index of a predicate PR_i, then the predicate arity and domains in SPDS are different from the ones in UPDS. In general, it is $N_i \neq M_i$. In SPDS, the predicate dimensions are related to *sdomains*. 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.

3.2 Index Area

The IA is composed of G^r_trees and is used for the fast retrieval of clauses. Each G^r_tree corresponds to one predicate. A G^r_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. Thus, the required dimensions to index the `q` have types integer, integer, atom and atom. Consequently, the `pr` is declared in SPDS as

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

If some index dimensions are left without arguments, they take a default value from their domain. If the number of paths in the DT of a UDD is n, k are the different *sdomain* 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^r_tree dimensions to index a UDD is

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

We examine the insertion of the clause of the `udd_ex`. The first argument (atom) corresponds to the third dimension, which has the same type. Likewise, the second

argument corresponds to the first dimension and the last argument to the fourth. 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 `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), whilst the second to clauses with *PI* = 2,2 and data (_,7). Answers that have the form `X=d(atm4)` or `X=e(9)`.

Indexing the head of rules is achieved by inserting them into the G^r_trees. In order to do that, PerKMan relates the declaration of variables in rules head 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' is `NULL`. Lower values are reserved by the manager and cannot be regarded as data. For example, to retrieve rules with head `gp(X,Y)` the index is searched for the partition where the entry `gp(NULL,-2147483647)` belongs (*X* stands for atoms and *Y* for integers). The clauses block of this partition is accessed and rules like the `gp(X,Y):-sp(_,X,Y),Y<10` are found where they exist. 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 `gp(_,20)` is indexed as `gp(NULL,20)`. The lower values of *cdomains* are constructed from the lower values of the *sdomains* that reside in the leaf nodes of their DT.

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.

3.3 Clauses Area

CA includes the clauses of all persistent predicates. Each block in it 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 connects blocks in case of overflow. A 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. The use of CAT is necessary due to the variable length of clauses.

4 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 DataBase (DB) and the Knowledge Base (KB) module [3]. The BANG file [5], a variant of the BD-tree, is used in both modules to index on the attributes that the user indicates. The DB module does not manipulate rules and non-ground facts, and attributes of type term cannot be included in the index. The KB module supports the persistent storage of any clause. 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 variation $5*10^6$. We chose 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 SUN Ultra 5/10 under SunOS 5.6.

Figure 4 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 times. The insertion time of PerKMan is linear and becomes lower than the one of ECL^{i}PS^e in a volume of $2*10^6$ facts.

Figure 5 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 100 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 $j*\lfloor NC/100 \rfloor$, $1 \le j \le 100$, of the insertion file.

Figure 6 shows that the two systems need the same number of disk accesses for exact match queries. Figures 7, 8 and 9 concern partial match queries with one, two and three variables, respectively. They show that the disk accesses required by PerKMan are fewer than the ones required by ECL^{i}PS^e. 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 cases. In some steps of these figures we observe some decrements in the number of disk accesses, despite the corresponding increment in the number of data. This is justified by the fact that

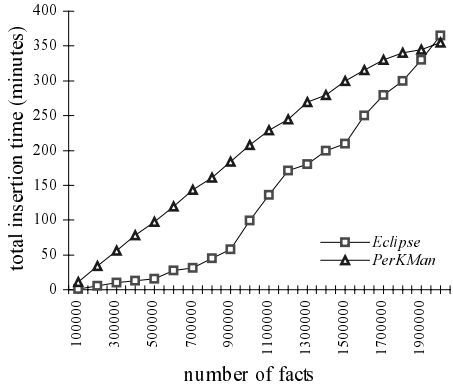

Fig. 3. Total insertion time.

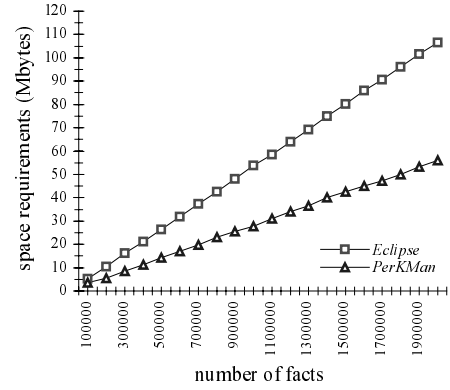

Fig. 4. Space requirements.

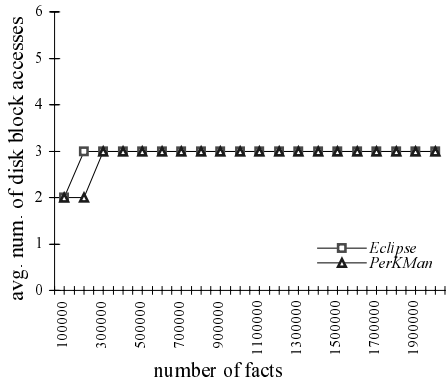

Fig. 5. Average number of accesses per exact match query.

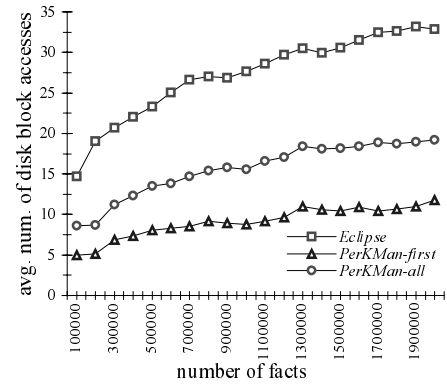

Fig. 6. Average number of accesses per partial match query of one variable.

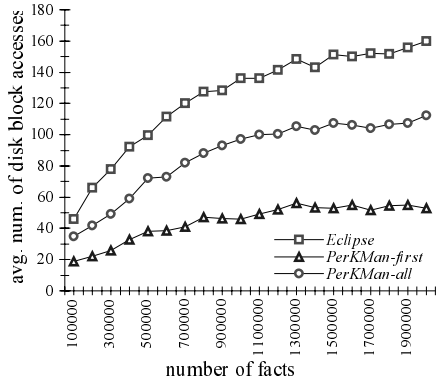

Fig. 7. Average number of accesses per partial match query of two variables.

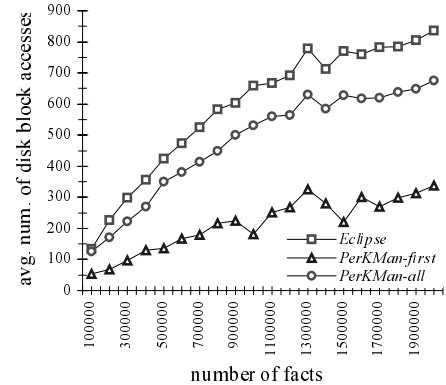

Fig. 8. Average number of accesses per partial match query of three variables.

queries were taken uniformly from the insertion file and, consequently, the queried facts were not the same for all steps.

5 Summary

The increasing need for large knowledge bases and efficient handling of 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. 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. From the performance of PerKMan, we believe that it achieves its design motivation, which is to handle efficiently large quantities of persistent knowledge.

Planned research work includes sophisticated methods that relate rules to data on a scheme that is based on the distribution of query types.

References

1. Cruickshank, G.: Persistent Storage Interface for Prolog – TERMdb, System Documentation. Draft Revision:1.5, Cray Systems (1994)
2. Derr, M.A., Morishita, S., Phipps, G.: The Glue-Nail Deductive Database System: Design, Implementation, and Evaluation. The VLDB Journal, vol.3, no.2 (1994) 123-160
3. ECL^{i}PS^{e} 3.7: Knowledge Base User Manual. ECRC GmbH (1998)
4. Van Emde Boas, G., Van Emde Boas, P.: Storing and evaluating Horn-clause rules in a relational database. IBM J Res.Develop. vol. 30, no. 1 (1986) 80-92
5. Freeston, M.: The BANG file: a new kind of grid file. Proc. of ACM, SIGMOD Conf. (1987) 260-269
6. Ioannidis, Y.E., Tsangaris, M.: The Design, Implementation, and Performance Evaluation of BERMUDA. IEEE Trans. on Knowledge and Data Eng., vol. 6, no. 1 (1994) 38-56
7. Kapopoulos D.G., Hatzopoulos, M.: The G^{r}_Tree: The Use of Active Regions in G-Trees. In: J.Eder, I. Rozman and T. Welzer (eds.): Advances in Databases and Information Systems. Lecture Notes in Computer Science, vol. 1691. Springer-Verlag, (1999) 141-155
8. Kumar, A.: G-Tree: A New Data Structure for Organizing Multidimensional Data. IEEE, Trans. on Knowledge and Data Eng., vol. 6, no. 2 (1994) 341-347
9. Nilsson, H., Ellemtel, A.: The External Storage Facility in SICStus Prolog. R:91:13, Swedish Institute of Computer Science (1995)
10. Ramakrishnan, R., Srivastava, D., Sudarshan, S., Seshadri, P.: The CORAL Deductive System. The VLDB Journal, vol.3, no.2 (1994) 161-210
11. Ramamohanarao, K., Shepherd, J., Balbin, I., Port, G., Naish, L., Thom, J., Zobel, J., Dart, P.: The NU-Prolog deductive database system. In: Gray, P.M.D., Lucas, R.J. (eds.): Prolog and Databases. West Sussex, Ellis Horwood limited (1988) 212-250
12. Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. Proc. of ACM SIGMOD Conf. (1994) 442-453
13. Vaghani, J., Ramamohanarao, K., Kemp, D.B., Somogyi, Z., Stuckey, P.J., Leask T.S., Harland, J.: The Aditi Deductive Database System. The VLDB Journal, vol.3, no.2 (1994) 245-288