Relational Storage Mechanisms for Tabled Logic Programs

Pedro Costa, Ricardo Rocha, and Michel Ferreira*

DCC-FC & LIACC
University of Porto, Portugal
c0370061@dcc.fc.up.pt {ricroc,michel}@ncc.up.pt

Abstract. Resolution strategies based on tabling are considerate to be particularly effective in Logic Programming. Unfortunately, when faced with applications that store large and/or many answers, memory exhaustion is a considerable problem. A common approach used to recover space is to delete some tables. In this work, we propose a different approach, storing these tables externally in a relational database. Subsequent calls to stored tables import answers from the database, rather than performing a complete re-computation. To validate this approach, we have extended the YapTab tabling system, providing engine support for exporting and importing tables to and from the MySQL relational database management system. Two different relational schemas for data storage and two data-set retrieval strategies are compared.

1 Introduction

Tabling is an implementation technique where intermediate answers for subgoals are stored and then reused when a repeated call appears. Resolution strategies based on tabling [1, 2] have proved to be particularly effective in logic programs, reducing the search space, avoiding looping and enhancing the termination properties of Prolog models based on SLD resolution [3].

The performance of tabling largely depends on the implementation of the table itself; being called upon very often, fast look up and insertion capabilities are mandatory. Applications can make millions of different calls, hence compactness is also required. Arguably, the most successful data structure for tabling is *tries* [4]. Tries are trees in which there is one node for every common prefix. Tries have proved to be one of the main assets of tabling implementations, because they meet the previously enumerated criteria of compactness and operability quite well. The YapTab tabling system [5] uses tries to implement tables.

Used in applications that pose many queries, possibly with a large number of answers, tabling can build arbitrarily many and very large tables, quickly filling up memory. In general, there is no choice but to throw away some of the

* This work has been partially supported by Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

tables, ideally, the least likely to be used next. The common control mechanism implemented in most tabling systems is to have a set of tabling primitives that the programmer can use to dynamically abolish some of the tables.

A more recent proposal is the approach implemented in YapTab, where a memory management strategy, based on a *least recently used* algorithm, automatically recovers space from the least recently used tables when the system runs out of memory [6]. With this approach, the programmer can still force the deletion of particular tables, but can also rely on the effectiveness of the memory management algorithm to completely avoid the problem of deciding what potentially useless tables should be deleted. Note that, in both situations, the loss of stored answers within the deleted tables is unavoidable, leading to the need of restarting the evaluation whenever a repeated call occurs.

In this work, we propose an alternative approach and instead of deleting tables, we store them externally using a relational database management system (RDBMS). Later, when a repeated call appears, we load the stored answers from the database, hence avoiding recomputing them. With this approach, the YapTab's memory management algorithm can still be used, this time to decide what tables to move to database storage when the system runs out of memory, rather than to decide what tables to delete. To validate this approach we thus propose DBTAB, a relational model for representing and storing tables externally in tabled logic programs. In particular, we will use YapTab as the tabling system and MySQL [7] as the RDBMS. The initial implementation of DBTAB handles only atomic terms such as integers, atoms and floating-point numbers.

The remainder of the paper is organised as follows. First, we briefly introduce some background concepts about tries and the table space. Next, we introduce our model and discuss how tables can be represented externally in database storage. We then describe how we extended YapTab to provide engine support for exporting and importing answers to and from the RDBMS. At the end, we present initial experimental results and outline some conclusions.

2 The Table Space

Tabled programs are evaluated by storing all computed answers for current subgoals in a proper data space, the *table space*. Whenever a subgoal $\mathcal{S}$ is called for the first time, a matching entry is allocated in the table space, under which all computed answers for the call are stored. *Variant calls*[1] to $\mathcal{S}$ are resolved by consumption of these previously stored answers. Meanwhile, as new answers are generated, they are inserted into the table and returned to all variant subgoals. When all possible resolutions are performed, $\mathcal{S}$ is said to be *completely evaluated*.

The table space can be accessed in a number of ways: **(i)** to look up if a subgoal is in the table, and if not insert it; **(ii)** to verify whether a newly found answer is already in the table, and if not insert it; and, **(iii)** to load answers to variant subgoals. For performance purposes, tables are implemented using two

[1] Two calls are said to be variants if they are the same up to variable renaming.

levels of tries, one for subgoal calls, other for computed answers. In both levels, stored terms with common prefixes branch off each other at the first distinguishing symbol. The table space is organized in the following way. Each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the *subgoal trie*. Each unique paths in this trie represents a different subgoal call, with the argument terms being stored within the internal nodes. The path ends when a *subgoal frame* data structure is reached. When inserting new answers, substitution terms for the unbound variables in the subgoal call are stored as unique paths into the *answer trie*. This optimisation is called *substitution factoring* [4].

Each trie node consists of a data structure with four fields each. The first field stores the symbol for the node. The second and third fields store pointers respectively to the first child node and to the parent node. The fourth field stores a pointer to the sibling node, in such a way that the outgoing transitions from a node can be collected by following its first child pointer and then the list of sibling pointers.

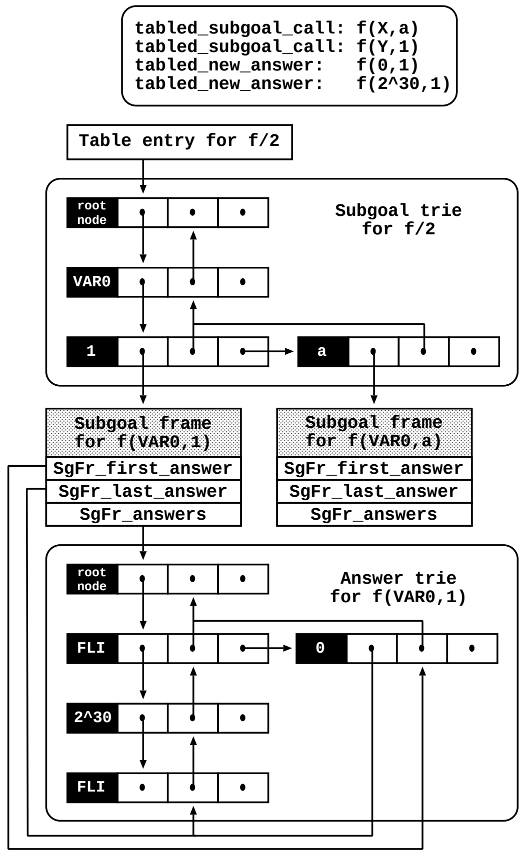


Fig. 1. Using tries to organise the table space

Among others, YapTab handles atomic terms such as integers, atoms and floating-point numbers. Terms are typed accordingly to the value of their mask bits, which cannot be used for data storage purposes. The non-mask part of a term is thus always less than the usual 32 or 64-bit C representation. In what follows we shall refer to integers larger than the maximum allowed non-tagged integer value as *long integer terms*. We shall also refer to integer and atom terms as *short atomic terms* and to floating-point and long integers as *long atomic terms*. Storing long term values requires the usage of additional nodes, consisting of special markers to delimit the data area. For this reason, while short term representation can be attained spending a single node, long atomic terms can use up to three and four nodes.

An example for a tabled predicate $f/2$ is shown in Fig. 1. Initially, the subgoal trie contains only the root node. When the subgoal $f(X, a)$ is called, two internal nodes are inserted: one for the variable X, and a second last for the constant a. Notice that variables are represented as distinct constants, as proposed by Bachmair *et al.* [8]. The subgoal frame is inserted as a leaf, waiting for the answers to appear. Then, the subgoal $f(Y, 1)$ is inserted. It shares one common node with $f(X, a)$, but the second argument is different so a different subgoal frame needs to be created. Next, the answers for $f(Y, 1)$ are stored in the answer trie as their values are computed. Notice how the 2^{30} long integer term is surrounded by two additional nodes tagged as `FLI` (*functor long int*). Finally, the leaf answer nodes are chained in a linked list in insertion time order (using the child field), so that recovery may happen the same way. The subgoal frame internal pointers `SgFr_first_answer` and `SgFr_last_answer` are set to point respectively to the first and last answer of this list. Therefore, when consuming answers, a variant subgoal needs only to keep a pointer to the leaf node of its last loaded answer, and consumes more answers just by following the chain. To load an answer, the trie nodes are traversed in bottom-up order and the answer is reconstructed.

3 The Relational Storage Model

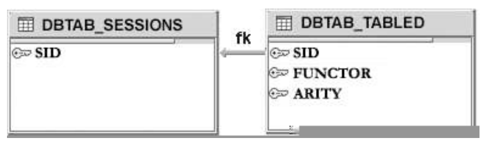


Fig. 2. The DBTAB system tables

DBTAB is expected to handle multi-user concurrency. Storing run-time data, of possible multiple sources, into the same database requires each running instance of YapTab to uniquely identify its own tabled predicates and its computed answers. To tackle this problem, DBTAB introduces the notion of *session*. The system tables used to keep control status are depicted in Fig. 2. Table DBTAB_SESSIONS keeps track of initialised sessions, while DBTAB_TABLED is used to identify each session YapTab's tabled predicates.

The choice of an effective representation model for the tables is a hard task to fulfil, mainly due to the high variability of the tries internal representation of different types of terms. The relational model is expected to quickly store and retrieve answers, thus minimizing the impact on YapTab's performance. With this concern in mind, two different database schemes were developed, each one focusing on a specific step of execution.

Multiple Table Predicate Representation To take full advantage of the relational model, data regarding the computed subgoal's answers is stored in several tables, aiming to keep the table space representation as small as possible in the database. Figure 3 shows the multiple table relational schema for the $f/2$ tabled predicate introduced back in Fig. 1.

The tabled predicate $f/2$ is mapped into the relational table SESSIONk_F2, where k is the current session id. Predicate arguments become the ARGi table integer fields[2], with i ranging between 1 and 2. SESSIONk_F2 is destined to hold

[2] The internal representation of terms can be thought of as 32 or 64-bit integers, so MySQL INTEGER or BIGINT types are used accordingly to store these values.

all the computed answers for each completely evaluated subgoal of $f/2$, along with a meta-representation of the subgoal itself. The table's integer field META is used to tell apart these two kinds of records: a zero value signals an answer trie branch; an one value signals a full bound subgoal trie branch and a positive value greater than one signals a meta-information record for a subgoal with unbound variables within its arguments. The uniqueness of each stored answer is ensured by the definition of a primary key involving every field of the table.

Opposite to short atomic terms, who's values are directly stored within the corresponding ARGi record fields of the main predicate table, long atomic terms are stored in the SESSIONk_LONGINTS and SESSIONk_FLOATS auxiliary tables. Each of these terms is stored only once and is uniquely identified by a sequential number masked as a YapTab functor term to simplify the loading algorithm. Mind that a *foreign key* relation between main table's ARGi fields and auxiliary table's TERM fields may not be defined, because short atomic terms are also stored within the first.

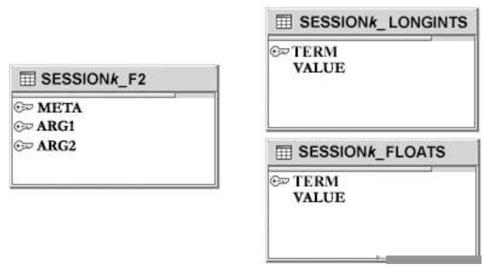


Fig. 3. The DBTAB multiple table schema

Single Table Predicate Representation The previous schema may require several transactions to store a single subgoal answer. For instance, for a subgoal such as $f/2$ with two floating-point bindings, five transactions may be required if the floating-point values have not been previously stored. To avoid over-heading in the storage operation, a simpler database schema as been devised.

Table SESSIONk_F2's design now considers the possibility of storage for long atomic terms (see Fig. 4). For that purpose, specifically typed fields are placed after each ARGi argument field. Regardless of this, each triplet is still considerate a single argument for record-set manipulation purposes, hence a single field may be initialised to a value other than NULL; the others must remain unset. This extended table approach requires no special masking for long atomic terms, albeit the appearance of NULL values in most field makes the definition of a primary key impossible.

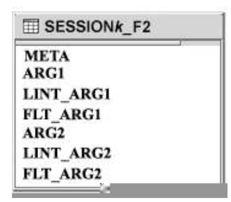


Fig. 4. The DBTAB single table schema

4 Extending the YapTab Design

DBTAB's model is meant to trigger the dumping of a completed tabled subgoal to the database when the corresponding table is chosen by YapTab's memory management algorithm to be abolished. The present version, although not yet suitable for fully safe usage, already implements the required features to correctly export and import tables, therefore allowing us to study and evaluate the potential and weaknesses of the proposed model.

4.1 Prepared Statements

Data exchange between the YapTab engine and the RDBMS is mostly done through the use of the MySQL C API for prepared statements. The SQL statements, used to retrieve/store information from/to the database, are sent to the database for parsing, and, on success, the returned handle is kept within a `PreparedStatement` data structure. This specialised buffer is responsible for keeping additional information about the statement's execution, including possible used parameters and/or possible returning result-set and respective fields meta-information.

Pointers to `PreparedStatement` structures are placed inside two major table space data structures: table entries and subgoal frames. Table entries are augmented with a pointer to an INSERT prepared statement. This statement is prepared to insert a full record at a time into the predicate's relational table, so that all subgoals hanging from the same table entry may use the same INSERT statement when storing their computed answers, therefore reducing the number of statements required for data submission transactions. On the other hand, subgoal frames are augmented with a pointer to a SELECT prepared statement. This statement is used to speed up the data retrieval transaction, while reducing the resulting record-set at the same time. Ground terms stored in the respective subgoal trie branch are used to refine the statement's WHERE clause - the corresponding fields in the relational representation need not to be selected for retrieval since their values are already known.

4.2 The DBTAB API

We next present the list of developed functions for the DBTAB's API and briefly describe their actions.

`dbtab_init_session(MYSQL *handle, int sid)` uses the database handle to initialise the session identified by the `sid` argument.

`dbtab_kill_session(void)` kills the currently opened session.

`dbtab_init_table(TableEntry tab_ent)` initialises the INSERT prepared statement associated with `tab_ent` and creates the corresponding relational table.

`dbtab_free_table(TableEntry tab_ent)` frees the INSERT prepared statement associated with `tab_ent` and drops the corresponding table if no other instance is using it.

`dbtab_init_view(SubgoalFrame sg_fr)` initialises the specific SELECT prepared statement associated with `sg_fr`.

`dbtab_free_view(SubgoalFrame sg_fr)` frees the SELECT prepared statement associated with `sg_fr`.

`dbtab_store_answer_trie(SubgoalFrame sg_fr)` traverses both the subgoal trie and the answer trie, executing the INSERT prepared statement placed at the table entry associated with the subgoal frame passed by argument.

`dbtab_fetch_answer_trie(SubgoalFrame sg_fr)` starts a data retrieval transaction executing the SELECT prepared statement for `sg_fr`.

4.3 Top-Level Predicates

Two new predicates were added and two pre-existing ones were slightly changed to act as front-ends to the developed API functions. To start a session we must call the `tabling_init_session/2` predicate. It takes two arguments, the first being a database connection handler and the second being a *session identifier*. This identifier can be either a free variable or an integer term meaning, respectively, that a new session is to be initiated or a previously created one is to be reestablished. These arguments are then passed to the `dbtab_init_session()` function, which will return the newly (re)started session identifier. The `tabling_kill_session/0` terminates the currently open session by calling `dbtab_kill_session()`.

YapTab's directive `table/1` is used to set up the predicates for tabling. The DBTAB expanded version of this directive calls the `dbtab_init_table()` function for the corresponding table entry data structure. Figure 5 shows, labeled as **(1)** and **(2)**, the INSERT statements generated, respectively, to each storage schema by the `dbtab_init_table()` function for the call `':- table f/2'`.

(1) INSERT IGNORE INTO SESSIONk_F2(META,ARG1,ARG2) VALUES (?,?,?);
(2) INSERT IGNORE INTO SESSIONk_F2(META,ARG1,LINT1,FLT1,ARG2,LINT2,FLT2)
VALUES (?,?,?,?,?,?,?);
(3) SELECT F2.ARG1 AS ARG1, L.VALUE AS LINT1 FROM SESSIONk_F2 AS F2
LEFT JOIN SESSIONk_LONGINTS AS L ON (F2.ARG1=L.TERM)
WHERE F2.META=0 AND F2.ARG2=22;
(4) SELECT DISTINCT ARG1,LINT1 FROM SESSIONk_F2 WHERE META=0 AND ARG2=22;
(5) SELECT ARG1 FROM SESSIONk_F2 WHERE META>1 AND ARG2=22;

Fig. 5. Prepared statements for $f(Y, 1)$

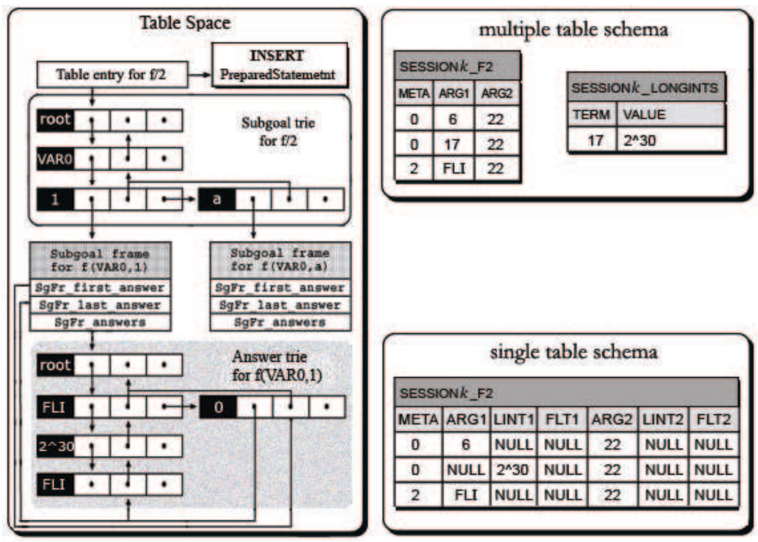
The `abolish_table/1` built-in predicate can be used to abolish the tables for a tabled predicate. The DBTAB expanded version of this predicate calls the `dbtab_free_table()` function for the corresponding table entry and the `dbtab_free_view()` function for each subgoal frame under this entry.

4.4 Exporting Answers

Whenever the `dbtab_store_answer_trie()` function is called, a new data transaction begins. Given the subgoal frame to store, the function begins to climb the subgoal trie branch, binding every ground term it finds along the way to the respective parameter in the INSERT statement. When the root node is reached, all parameters consisting of variable terms will be left NULL. The attention is then turned to the answer trie. For ground subgoal calls, no such structure is found and all parameters already hold a value different than NULL. For all other calls, control proceeds cycling through the terms stored within the answer trie nodes. The remaining NULL parameters are bound repeatedly, and the prepared statement is executed for each present branch.

Next, a single record of meta-information is stored. The META field value is set to a bit field structure that holds the total number of variables in the subgoal call. The least significative bit is reserved to differentiate answers generated by full ground subgoal trie branches from answer trie branches. The ARGi fields standing for variable terms present in the subgoal trie branch are bitwise masked with special markers. These markers identify each one of possible types of long terms that were found in the answer trie and were meant to be unified with the original variable.

Figure 6 illustrates the final result of the described process using both storage schemas. When the subgoal trie is first climbed, ARG2 is bound to the integer term of value 1 (internally represented as 22). All values for ARG1 are then bound cycling through the leafs of the answer trie. The branch for the integer term of value 0 (internally represented as 6) is stored first, and the branch for the long integer term 2^{30} is stored next. Notice how, in the multiple table schema, the ARG1 field of the second record holds the key for the auxiliary table record. At last, the meta-information is inserted. This consists of a record holding in the META field the number of variables in the subgoal call (1 in this case, internally represented by 2) and in the ARGi fields the different terms found in the answer trie for the variables in the subgoal call along with the other ground arguments.



multiple table schema

SESSIONk_F2

META	ARG1	ARG2
0	6	22
0	17	22
2	FLI	22

SESSIONk_LONGINTS

TERM	VALUE
17	2^30

single table schema

SESSIONk_F2

META	ARG1	LINT1	FLT1	ARG2	LINT2	FLT2
0	6	NULL	NULL	22	NULL	NULL
0	NULL	2^30	NULL	22	NULL	NULL
2	FLI	NULL	NULL	22	NULL	NULL

Fig. 6. Exporting $f(Y, 1)$ using both storage schemas

4.5 Importing Answers

When importing answers from the database, the first step consists in calling the `dbtab_init_view()` function in order to construct the specific SELECT statement used to fetch the set of answers for the subgoal. The `dbtab_init_view()` function first retrieves the meta-information from the database and then it uses the ground terms in the meta-information record to refine the search condition within the WHERE clause for the SELECT statement. This shortens the retrieved fields list, thus reducing the amount of data returned by the server.

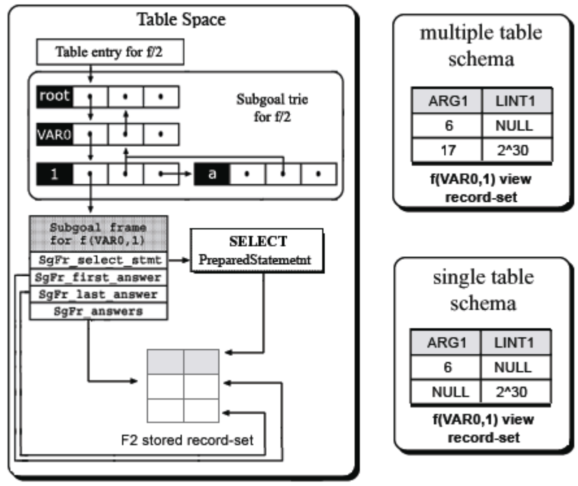


Fig. 7. Importing $f(Y, 1)$ using both storage schemas

The storage schemes differ somewhat in the way the returned result-set is interpreted. The multiple table schema sets the focus on the ARGi fields, where no NULL values can be found. Additional columns, placed immediately to the right of the ARGi fields, are regarded as possible placeholders of answer terms only when these *main* fields convey long atomic term markers. In such a case, the non-NULL additional field value is used to create the specific YapTab term. The single table schema, on the other hand, requires no sequential markers for long atomic terms, hence, it makes no distinction what so ever between ARGi and its possibly following auxiliary fields. For each argument (single field, pair or triplet), the first non-NULL value is considerate to be the correct answer term. Figure 7 shows, in the right boxes, the resulting *views* for each storage schema.

Figure 5 shows, labeled as **(3)** and **(4)**, the SELECT statements generated, respectively, to each storage schema by the call to `dbtab_init_view()`. Notice that statement **(4)** bears a DISTINCT option. This is the chosen way to prune repeated answers from the record-set, since no primary key is created for predicate tables in the single table schema. The statement labeled as **(5)** is used by both schemes to obtain the meta-information record. Notice how the search condition over the META field is established for subgoals calls with free variables.

4.6 Handling the Resulting Record-Sets

After the SELECT statement execution, two possible strategies may be used to supply the stored record-set with the answers back to the YapTab engine.

Rebuilding the Answer Trie In this scenario, the stored record-set is used only for answer trie rebuilding purposes. Traversing the records in a sequentially top-to-bottom fashion, the retrieved values are used to create the substitution factoring of the respective subgoal call, exactly as when the `tabled_new_answer` operation occurred. By the end of the cycle, the entire answer trie resides in the table space, as observed back in Fig. 1, and the record-set can then be released from memory. This approach requires no alteration to the YapTab's implemented API, safe for the call to `dbtab_fetch_answer_trie()`.

Browsing the Record-Set This approach aims at reducing the required memory space required to store completed subgoal calls by keeping data in this binary form. This is expected to lead to gains in performance since: **(i)** retrieval transaction occurs only once; **(ii)** no time and memory are spent rebuilding the answer trie; and **(iii)** long atomic term representation required down to one fourth of the usually occupied memory. Since the answer tries will not change once completed, all subsequent subgoal calls may fetch their answers from the record-sets obtained by `dbtab_fetch_answer_trie()`.

Figure 7 illustrates how the ancillary YapTab constructs are used to implement the idea. The left side box presents the state of the subgoal frame after answer collection for $f(Y, 1)$. The internal pointers are set to the first and last rows of the record-set. When consuming answers, the `SgFr_answers` value is tested to decide if the subgoal should fail, proceed or load answers from the database. If loading answers, the first record's offset along with the subgoal frame address are stored in a *loader choice point*[3]. The fetched record and its field values are then used to bind the free variables found for the subgoal in hand. If backtracking occurs, the choice point is reloaded and the last recorded offset is used to step through to the next answer. When, at the end of the record-set, an invalid offset is reached, the loader choice point is discarded and execution fails, thus terminating the ongoing evaluation.

5 Initial Experimental Results

A batch of tests using a simple path discovery algorithm over a graph (see Fig. 8) were performed in an Intel Pentium®4 XEON 2.6GHz processor with 2 GBytes of main memory and running the Linux kernel-2.6.18.

```
% connection handle creation stuff...
:- tabling_init_session(my_connection,Sid).
:- consult('graph.pl').
:- table path/2.

path(X,Z) :- path(X,Y), path(Y,Z).
path(X,Y) :- edge(X,Y).
```

Fig. 8. The test program

For comparison purposes, three main series of tests were performed both in YapTab and DBTAB environments (DBTAB with MySQL 5.0 running a InnoDB engine [7]). For each one of these series, the external file that holds the `edge/2` facts was generated with a different number of edges, aiming to achieve between 10,000 and 100,000 possible combinations among nodes. In each sub-series, two types of nodes were considered: integer and floating-point terms. The query `'?- path(X,Y).'` was executed 10 times for each setup and the mean of measured times, in milliseconds, is presented next in Table 1.

The table shows two columns for YapTab, measuring the generation and browsing times when using tries to represent the table space, two columns for

[3] A loader choice point is a WAM choice point augmented with a pointer to the subgoal frame data structure and with the offset for the last consumed record.

Answers	Terms	YapTab		DBTAB				
		Generate	Browse	Multiple Table		Single Table		Browse
				Export	Import	Export	Import	
10,000	integers	65	1	1055	16	1048	34	2
	floats	103	2	10686	44	1112	47	6
50,000	integers	710	6	4911	76	5010	195	12
	floats	1140	8	83243	204	5012	282	27
100,000	integers	1724	11	9576	153	9865	392	20
	floats	1792	14	215870	418	11004	767	55

Table 1. Execution times, in milliseconds, for YapTab and DBTAB

each of DBTAB storage schemes, measuring the times to export and import the respective number of answers and one last column, measuring the time to recover answers when using the approach that browses through the stored data-set. Some preliminary observations: **(i)** export and import times exclude the table generation time; **(ii)** when the trie is rebuilt after importing, this operation duration is augmented with generation time; **(iii)** when using tries, YapTab and DBTAB spend the same amount of time browsing them.

As expected, most of DBTAB's execution time is spent in data transactions (export and import). Long atomic terms (floats) present the most interesting case. For storage purposes, the single table approach is clearly preferable. Due to the extra search and insertion on auxiliary tables in the multiple table approach, the export time of long atomic terms (floats) when compared with their short counter-part (integers) increases as the number of answers also increases. For 10,000 answers the difference is about 10 times more, while for 1000,000 the difference increases to 20 times more. On the other hand, the single table approach seems not to improve the import time, since it is, on average, the double of the time spent by the multiple table approach. Nevertheless, the use of LEFT JOIN clauses in the retrieval SELECT statement (as seen in Fig. 5) may become a heavy weight when dealing with larger data-sets. Further experiments with larger tables are required to provide a better insight on this issue.

Three interesting facts emerge from the table. First, the browsing times for tries and record-sets are relatively similar, with the later requiring, on average, the double of time to be completely scanned. Secondly, when the answer trie becomes very large, re-computation requires more time, almost the double, than the fetching (import plus browse) of its relational representation. DBTAB may thus become an interesting approach when the complexity of re-calculating the answer trie largely exceeds the amount of time required to fetch the entire answer record-set. Third, an important side-effect of DBTAB is the attained gain in memory consumption. Recall that trie nodes possess four fields each, of which only one is used to hold a symbol, the others being used to hold the addresses of parent, child and sibling nodes (please refer to section 2). Since the relational representation dispenses the three pointers and focus on the symbol storage, the size of the memory block required to hold the answer trie can be reduced by a factor of four. This is the worst possible scenario, in which all stored terms are integers or atoms. For floating-point numbers the reducing factor raises to eight because, although this type requires four trie nodes to be stored, one floating-point requires most often the size of two integers. For long integer terms, memory gains go up to twelve times, since three nodes are used to store them in the trie.

6 Conclusions and Further Work

In this work, we have introduced the DBTAB model. DBTAB was designed to be used as an alternative approach to the problem of recovering space when the tabling system runs out of memory. By storing tables externally instead of deleting them, DBTAB avoids standard tabled re-computation when subsequent calls to those tables appear. Another important aspect of DBTAB is the possible gain in memory consumption when representing answers for floating-point and long integer terms. Our preliminaries results show that DBTAB may become an interesting approach when the cost of recalculating a table largely exceeds the amount of time required to fetch the entire answer record-set from the database.

As further work we plan to investigate the impact of applying DBTAB to a more representative set of programs. We also plan to improve the quality of the developed model. A first goal is to cover all possibilities for tabling presented by YapTab. Early stages of implementation shown that lists and application terms can be represented as record trees. Auxiliary tables, similar to those used for long atomic terms, are added to hold the internal terms of these complex types. The complex terms can be stored/retrieved from/to YapTab's addressing space using an algorithm that, starting from the root, descends the record tree one level at a time, recovering possible child records of each node.

References

1. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: International Conference on Logic Programming. Number 225 in LNCS, Springer-Verlag (1986) 84–98
2. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43** (1996) 20–74
3. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag (1987)
4. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38** (1999) 31–54
5. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
6. Rocha, R.: On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In: International Symposium on Practical Aspects of Declarative Languages. Number 4354 in LNCS, Springer-Verlag (2007) 155–169
7. Widenius, M., Axmark, D.: MySQL Reference Manual: Documentation from the Source. O'Reilly Community Press (2002)
8. Bachmair, L., Chen, T., Ramakrishnan, I.V.: Associative Commutative Discrimination Nets. In: International Joint Conference on Theory and Practice of Software Development. Number 668 in LNCS, Springer-Verlag (1993) 61–74