# DBTAB: a Relational Storage Model for the YapTab Tabling System

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 have proved to be particularly effective in logic programs. However, when tabling is used for applications that store large answers and/or a huge number of answers, we can quickly run out of memory. In general, to recover space, we will have no choice but to delete some of the tables. In this work, we propose an alternative approach and instead of deleting tables, we store them externally using a relational database system. Subsequent calls to stored tables would import answers from the database, hence avoiding recomputation. To validate our approach, we have extended the YapTab tabling system to provide engine support for exporting and importing tables to and from the MySQL relational database management system.

## 1 Introduction

Tabling [1] is an implementation technique where intermediate answers for subgoals are stored and then reused when a repeated call appears. Resolution strategies based on tabling [2, 3] 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 [4].

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* [5]. Tries are trees in which there is one node for every common prefix [6]. Tries have proved to be one of the main assets of tabling implementations, because they are quite compact for most applications while having fast look up and insertion. The YapTab tabling system [7] uses tries to implement tables.

When tabling is used for applications that build many queries or that store a huge number of answers, we can build arbitrarily many and possibly very large tables, quickly filling up memory. In general, there is no choice but to throw away some of the tables (ideally, the least likely to be used next). The common control 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. 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, we can still use YapTab's memory management algorithm, but to decide what tables to move to database storage when the system runs out of memory, instead of using it 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 [8] as the RDBMS. The initial implementation of DBTAB only handles atomic terms such as integers, atoms and floating-point numbers.

The remainder of the paper is organized 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 found 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 and every generated answer for the subgoal is stored under this entry. Repeated calls to $\mathcal{S}$ or its *variants*[1] 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. Two levels of tries are used to implement tables, one for subgoal calls, other for computed answers. Each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the *subgoal trie*. Every subgoal call is represented in this trie as an unique path to a *subgoal frame* data structure, with argument terms stored within the internal nodes. Terms with

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

common prefixes branch off each other at the first distinguishing symbol. If free variables are present within the arguments, all possible bindings are stored into the *answer trie*, i.e., all possible answers to the subgoal are mapped to unique paths in this second trie. When inserting new answers, only the substitutions for the unbound variables in the subgoal call are stored. This optimization is called *substitution factoring* [5].

Tries are implemented by representing each trie node by a data structure with four fields each. The first field (`TrNode_symbol`) stores the symbol for the node. The second (`TrNode_child`) and third (`TrNode_parent`) fields store pointers respectively to the first child node and to the parent node. The fourth field (`TrNode_next`) 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.

An example for a tabled predicate $f/2$ is shown in Fig. 1. Initially, the subgoal trie only contains 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.* [9]. 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.

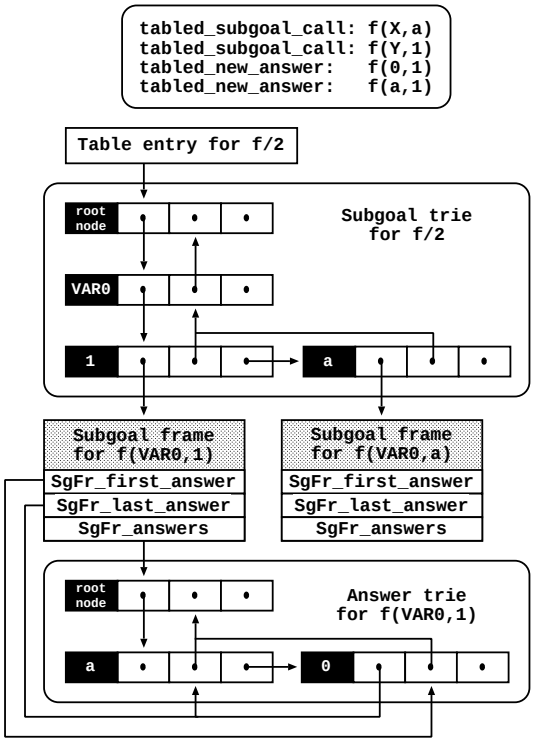


**Fig. 1.** Using tries to organize the table space

At last, the answers for $f(Y, 1)$ are stored in the answer trie as their values are computed. The leaf answer nodes are chained in a linked list in insertion time order (using the `TrNode_child` field), so that recovery may happen the same way. Finally, 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. Thus, 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

The chosen RDBMS for DBTAB is MySQL 4.1 [8], running a InnoDB storage engine. This is a transaction-safe engine with commit, rollback and crash-recovery capabilities. InnoDB tables present no limitation in terms of growth and support FOREIGN KEY constraints with CASCADE abilities. The MySQL C API for prepared statements is used to manipulate the record-sets, benefiting of several advantages in terms of performance: the statement is parsed only once, when it is first sent; network traffic is substantially reduced since statement invocation require only the respective input parameters; a binary protocol is used to transfer data between the client and the server.

### 3.1 Representing the Table Space

Figure 2 shows the ER diagram for the relational representation of the table space, the DBTAB database. The diagram is divided in two types of tables: system tables, identified by the prefix DBTAB, and predicate tables, identified by the prefix SESSION. System tables basically maintain control status, while predicate tables are meant to hold look-up values and run-time data, such as computed answers and meta-information about known subgoals.

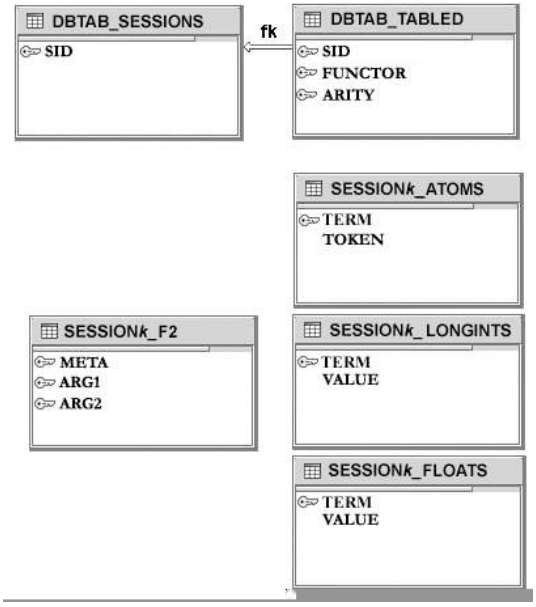


**Fig. 2.** The DBTAB relational schema

Storing run-time data, of possible multiple sources, raises the important issue of multi-user concurrency. Since the same database is to be used as a final repository of data, each running instance of YapTab must be uniquely identified in order to refer to its own found answers. To tackle this problem, DBTAB introduces the notion of *session*. A specific predicate, `tabling_init_session/1`, is introduced to initialize sessions. It takes one argument that is considered to be a *session id*, that can be either a free variable or a ground term. In the first case, a new identifying integer is attained from the DBTAB_SESSIONS table and unified with the variable. On the other hand, if the argument is a ground integer, its value is searched in the sessions table, and should it be found, the indicated session is reestablished[2]. The

[2] Currently, this means only that the same session id is reused.

`tabling_kill_session/0` predicate can be used to finish the opened session and clean-up all of its dependent information.

Tabling begins by the identification of the predicates to handle. YapTab's directive '`:- table p/n.`' is used for this purpose, generating a new table entry data structure for the specified predicate and inserting it into the table space. DBTAB extends the previously described behaviour by registering functor $p$ and arity $n$ into DBTAB_TABLED. The SID field acts as a foreign key to DBTAB_SESSIONS, establishing the dependency net for the session. DBTAB then dynamically creates a new relational table SESSION$k$_PN to hold all completed answer tries for $p/n$, together with three auxiliary relational tables SESSION$k$_ATOMS, SESSION$k$_LONGINTS and SESSION$k$_FLOATS, with $k$ being the current session id. Along with all computed answers, a meta-representation for every $p/n$'s completely evaluated subgoal is to be stored within the SESSION$k$_PN table. The integer field META is used to tell apart these two kinds of records: a zero value signals an answer; a positive non-zero value signals a meta-information record. The arguments of $p/n$ are mapped into integer fields[3] named ARG$i$, with $i$ being an index between 1 and $n$.

The `abolish_table(p/n)` predicate is used by YapTab to remove $p/n$'s entry from the table space. DBTAB's expansion of this predicate deletes the corresponding record from DBTAB_TABLED and drops all the session tables associated with $p/n$. Both in YapTab and DBTAB, the `abolish_all_tables/0` predicate can be used to dispose of all table entries: the action takes place as if `abolish_table/1` was called for every tabled predicate.

### 3.2 Handling Primitive Types

YapTab handles atomic terms such as integers, atoms and floating-point numbers. YapTab determines the type of each term by reading its mask bits. The non-mask part of a term is thus less than the usual 32 or 64-bit representation, so an additional term is used to represent floating-point values and integers greater than the maximum masked integer allowed. In what follows we call these integers *long integer terms*. Due to this difference in sizes, internal representation at trie level may require more than one node. While integer and atom terms use only one node, long integer and floating-point terms use 3 and 4 nodes.

DBTAB explores this idea and handles answer terms dividing them in two categories: *atomic terms* have their values directly stored within the corresponding ARG$i$ record fields; *non-atomic terms* are substituted in the ARG$i$ record fields by unique sequential values that work as a foreign key to the TERM field of the auxiliary tables for the predicate. These sequential values are masked as YapTab terms in order to simplify the loading algorithm[4]. Atomic terms com-

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

[4] The `YAP_MkApplTerm()` macro is used to create *dummy* application terms and then the non-tag part of these terms is used to hold the sequential value.

prise integers and atoms, while floating-point and long integer terms are considered non-atomic. Two auxiliary tables are defined to hold non-atomic values: SESSION$k$_FLOATS stores floating-point values used to build floating-point terms; SESSION$k$_LONGINTS is used to store long integer values.

Aside from storing atoms into the predicate tables, every session stores their YapTab's internal representation as well as their string values, respectively in the TERM and TOKEN fields of its SESSION$k$_ATOMS table. When reestablishing a session, this table is used to rebuild the previous internal symbol addressing space.

### 3.3 Manipulating Data Through Prepared Statements

Data exchange between the database and YapTab is done through the MySQL C API along with a *prepared statement* data structure (see Fig. 3 for details). The SQL statements used to store/retrieve information are sent to the database for parsing, and, on success, the returned handle is used to initialize the `statement` pointer. Additional information about possible used parameters is stored within the sub-structure `params`.

If a record-set is to be returned upon the statement's execution, the result-set pointer `records.metadata` and the sub-structure `fields` are initialized with information regarding it. For predicate tables, the `stmt_buffer` pointer will be initialized with an integer array, sized to hold an entire record. Since the `params` and `fields` sub-structures are never used at the same time, they can share the `stmt_buffer`, `bind`, `null`, and `length` arrays - these arrays are sized accordingly to the largest requirement in terms of size. After record storing, the `records.num_rows` sub-structure holds the count of retrieved rows.

```
typedef struct prepared_statement {
  MYSQL_STMT     *statement;
  my_ulonglong   affected_rows;
  void           *stmt_buffer;
  struct {
    int          count;
    MYSQL_BIND   *bind;
    my_bool      *null;
    my_ulong     *length;
  } params;
  struct {
    int          count;
    MYSQL_BIND   *bind;
    my_bool      *null;
    my_ulong     *length;
  } fields;
  struct {
    MYSQL_RES    *metadata;
    my_ulonglong num_rows;
  } records;
} *PreparedStatement;
```

**Fig. 3.** The prepared statement structure

The table entry data structure is augmented with a pointer to a generic INSERT prepared statement. All subgoals branches hanging from this table entry share the same prepared statement. Computed answers are stored by instantiating its input parameters as required and executing it. The subgoal frame data structure is augmented with a pointer to a specific SELECT prepared statement. Ground terms in the subgoal trie are used in the refinement of the WHERE clause; the corresponding fields are not selected for retrieval since their values are already known.

Figure 4 shows the prepared statements generated to store and recover the $f(Y, 1)$ subgoal call introduced back at Fig. 1. The first statement is the generic INSERT statement that is used for all insertions into table SESSION$k$_F2. The

second statement is one of the specific SELECT statements that are used to retrieve the records that contain the answer trie. It takes no input parameters and returns only one field, ARG1. Note that value 22 is the YapTab's internal representation of the integer term of value 1. The third statement presents a similar query that will retrieve answers if floating-point values are expected to unify with the variable term $Y$. Finally, the forth statement collects the meta-data for the subgoal.

```
(1) INSERT IGNORE INTO SESSIONk_F2(META,ARG0,ARG1) VALUES(?,?,?);
(2) SELECT ARG1 FROM SESSIONk_F2 WHERE META=0 AND ARG2=22;
(3) SELECT F2.ARG1, FLOATS.VALUE AS FLT_ARG1
    FROM   SESSIONk_F2 AS F2 LEFT JOIN SESSIONk_FLOATS AS FLOATS
           ON (F2.ARG1=FLOATS.TERM)
    WHERE META=0 AND ARG2=22;
(4) SELECT ARG1 FROM SESSIONk_F2 WHERE META=1 AND ARG2=22;
```

**Fig. 4.** Prepared statements for $f(Y, 1)$

### 3.4 The DBTAB API

We next present the list of developed functions and briefly describe their actions.

`dbtab_init_session(int sid)` - initializes the session passed by argument;
`dbtab_kill_session(void)` - kills the currently opened session;
`dbtab_init_table(TableEntry tab_ent)` - creates the relational table and initializes the generic INSERT prepared statement associated with the table entry passed by argument;
`dbtab_free_table(TableEntry tab_ent)` - frees the INSERT prepared statement, dropping the table if no other instance is using it;
`dbtab_init_view(SubgoalFrame sg_fr)` - initializes the specific SELECT prepared statement associated with the passed subgoal frame;
`dbtab_free_view(SubgoalFrame sg_fr)` - frees the SELECT prepared statement;
`dbtab_export(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. The answer trie is deleted at the end of the transaction;
`dbtab_import(SubgoalFrame sg_fr)` - starts a data retrieval transaction, executing the SELECT prepared statement for the subgoal frame passed as argument.

## 4 Extending the YapTab Design

When a predicate is declared as tabled, the `dbtab_init_table()` function is called, starting the table creation process and generation of the INSERT clause, sending it afterwards to the database for parsing. If preparation succeeds, the returned handle is placed inside the corresponding table entry data structure.

DBTAB's final model is meant to trigger the dumping of a tabled subgoal to the database when the corresponding table is chosen by YapTab's memory management algorithm to be abolished. Currently, DBTAB is still not yet fully integrated with YapTab's memory management algorithm. However, the present version already implements all the required features to correctly export and import tables, therefore allowing us to study and evaluate the potential and weaknesses of the proposed model. The current version of DBTAB triggers the dumping of a tabled subgoal to the record-set upon its *completion* operation, removing it from memory afterwards - it is to be replaced by a record-set storing the same answer terms. This operation is delayed up to this point in execution in order to prevent unnecessary memory consumption, both at client and server sides, while only incomplete tables are known. Variant calls to completed subgoals always import answers from the database.

### 4.1 Exporting Answers

Figure 5 shows the pseudo-code for the `dbtab_export()` function. Initially, the function starts a new data transaction. It then begins to climb the subgoal trie branch, binding the ground terms to the respective statement parameters along the way. 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, cycling through the terms stored within the answer nodes. The remaining NULL parameters are bound repeatedly, and the prepared statement is executed for each present branch. Next, the meta-information about variables is stored. For each variable term present in the subgoal trie branch, a new unassigned variable term is created. The non-tag part of this variable is used to store a bit-mask containing information about all the possible types of terms that will be unified with the original variable. The total number of variables is stored in the META

```
dbtab_export(SubgoalFrame sg_fr) {
  dbtab_start_transaction()
  insert_stmt = TabEnt_insert_stmt(SgFr_tab_ent(sg_fr))
  n_vars = bind_subgoal_terms(SgFr_parent(sg_fr))
  answer = SgFr_first_answer(sg_fr)
  while (answer != NULL) {
    bind_answer_terms(answer)                          // prepare record
    commit &= exec_prep_stmt(insert_stmt)
    answer = TrNode_child(answer)
  }
  if (!n_vars) {               // n_vars is the number of free variables
    bind_subgoal_metadata(n_vars)                 // prepare meta-record
    commit &= exec_prep_stmt(insert_stmt)
  }
  if (commit) {
    mysql_commit(DBTAB_SCHEMA)
    mark_as_stored(sg_fr)                  // update subgoal frame state
    free_answer_trie(SgFr_answers(sg_fr))
  } else {
    mysql_rollback(DBTAB_SCHEMA)
  }
}
```

**Fig. 5.** Pseudo-code for `dbtab_export()`

field of this record. Finally, the COMMIT of the transaction occurs if and only if all INSERT statements are executed correctly; otherwise, a ROLLBACK operation is performed.

To clarify ideas, recall the example of Fig. 1. During the execution of the ':- table f/2.' directive, table SESSION$k$_F2 is created in the DBTAB database and the INSERT statement, meant to handle all insertions into this table, is generated and sent to the database, which will return a handle for it upon successful parsing. The handle is placed inside a prepared statement data structure pointed by the `TabEnt_insert_stmt` field of the table_entry data structure (see Fig. 6 for details). The structure's field `stmt_buff` and `params` sub-structure are initialized, with `params.bind` being set to point at a newly created array of MYSQL_BIND structures.

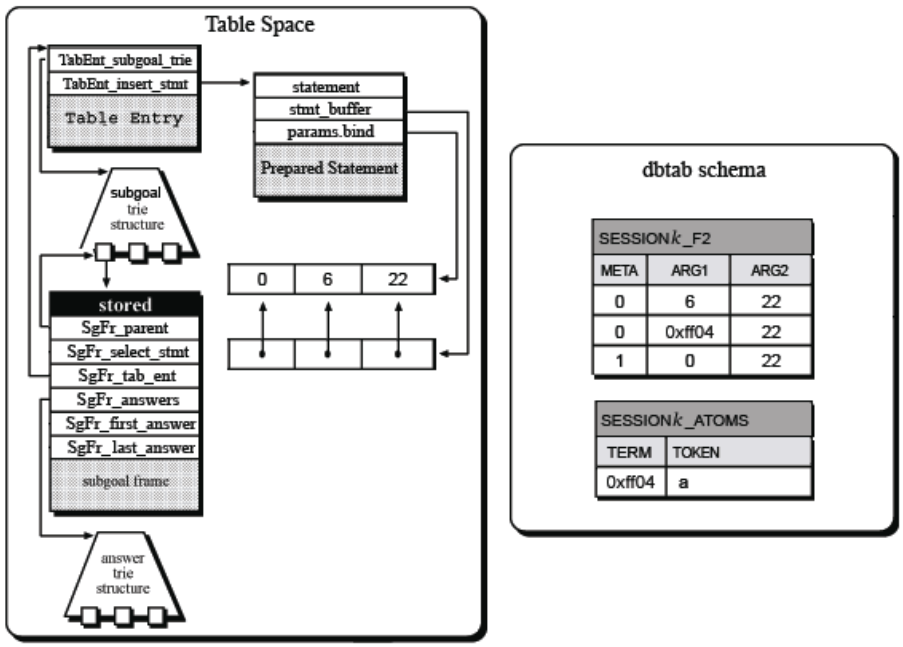


**Fig. 6.** Exporting $f(Y, 1)$: the relational representation

When completion is reached, the subgoal trie is climbed binding the second parameter, ARG2, with the integer term of value 1 (appearing in its internal representation 22). All values for ARG1 are then bound cycling through the leafs of the answer trie. Each branch is climbed up to the root node, that marks the point where the insertion is to be performed. The branch for the integer term of value 0 (internally represented by 6) is stored right away, but the one for atom $a$ (internally represented by 0xff04 in the figure) requires extra work because atoms are also stored into the corresponding SESSION$k$_ATOMS table. At last, the meta-data is inserted. This consists of a record holding the different terms found in the answer trie for the free arguments in the subgoal call along with the other ground arguments. A new variable term replaces $VAR0$, and its

non-tag part is used to hold a bit-mask - the value 0 visible in the last record - signaling that ARG1 column holds no special typed terms[5]. The META field holds the number of free variables for the subgoal, 1 in this case.

### 4.2 Importing Answers

After completion, the first variant call to a stored subgoal call now executes the `dbtab_import()` function, presented at Fig. 7.

```
dbtab_import(SubgoalFrame sg_fr) {
  select_stmt = SgFr_select_stmt(sg_fr)
  if (!PS_STMT(select_stmt)) {
    dbtab_init_view(sg_fr)
    exec_prep_stmt(select_stmt)
  }
  // switch on the number of rows
  if (PS_NROW(select_stmt) == 0) {                          // no answers
    SgFr_first_answer(sg_fr) = NULL
    SgFr_last_answer(sg_fr) = NULL
    SgFr_answers(sg_fr) = NULL
  } else {
    SgFr_first_answer(sg_fr) = PS_TOP_RECORD(select_stmt)
    SgFr_last_answer(sg_fr) = PS_BOTTOM_RECORD(select_stmt)
    if(VIEW_FIELD_SUCCEED == TRUE)                          // yes answer
      SgFr_answers(sg_fr) = NULL
    else                                              // one or more answers
      SgFr_answers(sg_fr) = SgFr_first_answer(sg_fr)
  }
}
```

**Fig. 7.** Pseudo-code for `dbtab_import()`

The first step calls the `dbtab_init_view()` function, creating the SELECT prepared statement that will load all possible answers. All variable terms are returned by default, possibly in conjunction with additional columns for non-atomic values. In case floating-point values are to be returned by ARG$k$, an additional column FLT_ARG$k$ is joined to the data-set (see query 3 at Fig. 4 for such an example). Likewise, if long integers are to be returned by ARG$k$, an additional column LINT_ARG$k$ is joined to the data-set. These columns are placed immediately to the right of ARG$k$ and possibly both of them may appear simultaneously - if such is the case, only one of these columns is set to a non-NULL value. The choice on which one of them to use is made consulting ARG$k$, who's value is replaced by a functor term for the desired type[6].

Also during this process, ground terms are used to set search conditions, within the WHERE clause, to be matched upon data retrieval in order to shorten the fields list, thus reducing the amount of data returned by the server. The statement is finally sent to the database for parsing and, on success, the returned handle is stored inside the prepared statement data structure added to the subgoal frame.

[5] Meaning floating-point numbers or long integers.

[6] YapTab defines internally two special functors for this purpose: `FunctorDouble` and `FunctorLongInt`.

Since the predicate's answer trie will not change once completed, all subsequent calls may fetch their answers from the obtained record-set. The next step is then to reset the subgoal frame `SgFr_first_answer`, `SgFr_last_answer` and `SgFr_answers` internal fields accordingly to the obtained data-set:

**Ground queries** return at most one record. On failure, the pointers are all set to NULL and no record is returned, which means that the answer is *no*. On success, `SgFr_first_answer` and `SgFr_last_answer` point at the only record of the fetched data-set, consisting of a single boolean field named SUCCEED holding a TRUE value, and `SgFr_answers` holds the NULL value, indicating this is a *yes* answer.

**Non-ground queries** may return more than one record. If the reduction of the subgoal holds, the `SgFr_answers` and `SgFr_first_answer` pointers are set respectively to the first record of the data-set, while `SgFr_last_answer` is set to the last.

Figure 8 shows answer collection for $f(Y, 1)$. The meta-data is recovered through the execution of query 4 at Fig. 4. The constant term 1 (internally represented by value 22) is used to set a search condition over ARG2. All values in column ARG1 are then recovered as possible answers for variable term $Y$ through the execution of query 2 at Fig. 4. At last, the subgoal frame pointers `SgFr_answers`, `SgFr_first_answer` and `SgFr_last_answer` are set to the first and last records as explained above.

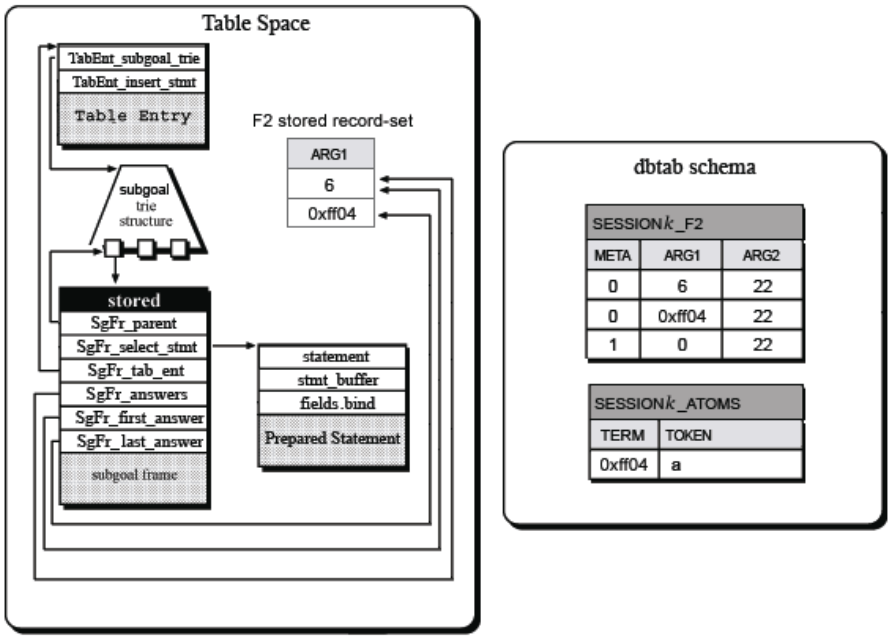


**Fig. 8.** Importing $f(Y, 1)$: the resulting data-set

As control returns from `dbtab_import()`, the `SgFr_answers` value is tested to decide if the query 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 within a *loader choice point*[7]. 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 its `CP_last_answer` field, containing the offset for the last consumed record, is used to calculate the offset for the next answer. If the new offset is a valid one, the `CP_last_answer` is updated accordingly. Otherwise, the choice point is discarded, signaling the positioning at the last answer. Whatever the case, the record is fetched and the variables are rebound according to the fields values. This process continues until all answers are consumed.

## 5 Initial Experimental Results

A batch of tests were performed in a computer with a Pentium®4 2.6GHz processor and 1GB of RAM. The test program, shown in Fig. 9, is a simple path discovery algorithm over a graph.

```
:- consult('graph.pl').
:- tabling_init_session(S).
:- table path/2.

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

**Fig. 9.** The test program

For comparison purposes, three main series of tests were performed both in YapTab and DBTAB environments. For each one of these series, the external file that holds the `edge/2` facts was generated with a different number of edges, ranging roughly from 50 to 150, corresponding to 5000 to 50000 possible combinations among nodes. In each sub-series, three types of nodes were considered: integer, floating-point and atom terms. The query '`?- path(X,Y).`' was executed 10 times for each setup and the mean of measured times, in milliseconds, is presented in Table 1. The table shows two columns for YapTab, measuring the generation and recovery times when using tries to represent the table space, and three columns to DBTAB, measuring the times to export and import the respective number of answers and the time to recover answers when navigating the stored data-set after importing it.

As expected, most of DBTAB's execution time is spent in data transactions, mainly during insertion of tuples. Storage of non-integer terms takes approximately three times more than their integer counter-part, due to the extra insertion on auxiliary tables. An implementation of this step using stored procedures might accelerate things a little bit, since it takes only one message to be sent

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

| Answers | Terms | YapTab | | DBTAB | | |
|---|---|---|---|---|---|---|
| | | Generation | Recovery | Export | Import | Recovery |
| 5000 | integers | 23 | 1 | 387 | 41 | 2 |
| | atoms | 21 | 2 | 1148 | 37 | 3 |
| | floats | 22 | 2 | 1404 | 54 | 3 |
| 10000 | integers | 58 | 2 | 780 | 60 | 3 |
| | atoms | 66 | 2 | 2285 | 63 | 4 |
| | floats | 64 | 3 | 2816 | 94 | 5 |
| 50000 | integers | 413 | 5 | 3682 | 240 | 15 |
| | atoms | 422 | 6 | 11356 | 252 | 12 |
| | floats | 386 | 20 | 14147 | 408 | 34 |

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

and most of the processing is done server-side. Non-atomic terms (floats) also present an interesting problem at fetching time. The use of LEFT JOIN clauses in the retrieval select statement (as seen in Fig. 4) becomes a heavy weight when dealing with large data-sets. Some query optimization is required to simplify the process and decrease the time required to import answers.

Two interesting facts emerge from the table. First, the navigation times for tries and data-sets are relatively similar, with stored data-sets requiring, on average, the double of time to be completely scanned. The second observed fact regards the time required to recompute answer tries for atomic terms (integers and atoms). When the answer trie becomes very large (the 50000 tuples rows), its computation requires more time, almost the double, than the fetching (import plus recovery) of its relational representation. DBTAB may thus become an interesting approach when the complexity of recalculating the answer trie largely exceeds the amount of time required to fetch the entire answer data-set.

An important side-effect of DBTAB is the attained gain in memory consumption. Recall that trie nodes are represented with 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: three nodes are used to store them in the trie.

## 6 Conclusions and Further Work

In this work, we have introduced the DBTAB model: a relational database model to represent and store tables externally in tabled logic programs. We discussed how to represent tables externally in database storage; how to handle atomic terms such as integers, atoms and floating-point numbers; and how we have

extended the YapTab tabling system to provide engine support for exporting and importing answers to and from the database.

DBTAB was designed to be used as an alternative approach to the problem of recovering space when the tabling system runs out of memory. The common control implemented in most tabling systems is to have a set of tabling primitives that the programmer can use to dynamically delete some of the tables. By storing tables externally instead of deleting them, DBTAB avoids re-computation when subsequent calls to those tables appear. Another important aspect of DBTAB is the gain in memory consumption when representing answers for floating-point and long integer terms. Our preliminaries results showed 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 data-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 introduce some other enhancements to improve the quality of the developed model. The expansion of the actual DBTAB model to cover all possibilities for tabling presented by YapTab is the first goal to achieve in a near future. First implementation tests shown that pairs, lists and application terms can be recorded and recovered through the use of a recursive algorithm and record trees. Each of these types of term is represented by a sequential number, which serves as the tree root, that is to be stored as described before at the tabled predicate relational table. Auxiliary tables have to be built to store all internal terms used by complex terms. These tables must possess a key field that links every node descendant to their direct ancestor. This operation is easy to implement and is expected to execute very quickly. Recovering is slightly more expensive. All child-nodes of the root node have to be selected, each one of them being interpreted as the root of a new sub-tree. The process continues until all leave-nodes are reached. By then, the specific term can be reconstructed by YapTab.

During execution, YapTab processes may have to stop due to several reasons: hosts may crash or have to be turned off, the users may want to interrupt process evaluation, etc. If such a situation arises, table space residing in memory is lost, leading to repeated calculation of the completed answer tries in later program executions. A possible solution to this problem is to search for meta-representation of terms before starting the process of tabling. If such a representation is found, the information contained in it can be used to not only build the corresponding branch in the subgoal tree but also the required prepared statements used to store new found answers and retrieve previously computed ones.

## Acknowledgments

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

## References


1. Michie, D.: Memo Functions and Machine Learning. Nature **218** (1968) 19–22
2. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: International Conference on Logic Programming. Number 225 in LNCS, Springer-Verlag (1986) 84–98
3. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43** (1996) 20–74
4. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag (1987)
5. 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
6. Fredkin, E.: Trie Memory. Communications of the ACM **3** (1962) 490–499
7. 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
8. Widenius, M., Axmark, D.: MySQL Reference Manual: Documentation from the Source. O'Reilly Community Press (2002)
9. 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