Relational Models for Tabling Logic Programs in a Database

Pedro Costa[1], Ricardo Rocha[2], and Michel Ferreira[1]

[1] DCC-FC & LIACC
University of Porto, Portugal
c0370061@dcc.fc.up.pt michel@dcc.fc.up.pt
[2] DCC-FC & CRACS
University of Porto, Portugal
ricroc@dcc.fc.up.pt

Abstract. Resolution strategies based on tabling are considered to be particularly effective in Logic Programming. Unfortunately, when faced with applications that compute large and/or many answers, memory exhaustion is a potential problem. In such cases, table deletion is the most common approach to recover space. In this work, we propose a different approach, storing tables into 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 RDBMS. Three different relational models for data storage and two recordset retrieval strategies are compared.

1 Introduction

Tabling [1] is an implementation technique where intermediate answers for subgoals are stored and then reused whenever a repeated call appears. Resolution strategies based on tabling [2, 3] proved themselves particularly effective in Logic Programming – the search space is reduced, looping is avoided and the termination properties of Prolog models based on SLD resolution are enhanced.

The performance of tabled evaluation largely depends on the implementation of the table itself – being called very often, fast lookup 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 the *trie* [4]. Tries meet the previously enumerated criteria of compactness and efficiency 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/or very large tables, quickly filling up memory. Swapping the main memory to disk may attenuate the problem but it is not a practical solution, since the memory space is always limited by the underlying architecture. In general, there is no choice but to throw away

some of those tables, ideally, the least likely to be used next [6]. The most 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 has been implemented in YapTab [6], where a memory management strategy based on a *least recently used* algorithm automatically recovers space among the least recently used tables when memory runs out. With this approach the programmer can still force the deletion of particular tables, but may also transfer the decision of which tables are potential candidates for deletion to the memory management algorithm. Note that, in both situations, the loss of answers stored within the deleted tables is unavoidable, eventually leading to re-computation.

In this work, we propose an alternative approach in which tables are sent to secondary memory – a relational database management system (RDBMS) – rather than deleted. Later, when a repeated call occurs, the answers are reloaded from the database thus avoiding re-computation. With this approach, the memory management algorithm can still be used, this time to decide which tables are to be sent to the database when memory runs out. To validate this approach we propose DBTab [7], a relational model for tabled logic program support resulting from the coupling of the YapTab tabling system with the MySQL RDBMS. In this initial implementation the ability of DBTab to represent terms is restricted to atomic strings and 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 in a RDBMS. We then describe how we have extended YapTab to provide engine support for database stored answers. Finally, we present initial results and outline some conclusions.

2 The Table Space

Tabled programs evaluation proceeds 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 new table entry is allocated in the table space – all answers for subgoal $\mathcal{S}$ will be stored under this entry. Variant calls to $\mathcal{S}$ are resolved consuming those previously stored answers. Meanwhile, as new answers are generated, they are inserted into the table and returned to all variant subgoals. When all possible answers are found, $\mathcal{S}$ is said to be *completely evaluated*.

The table space may be accessed in a number of ways: **(i)** to find out 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. With these requirements, performance becomes an important issue so YapTab implements its table space resorting to *tries* [8] – which is regarded as a very efficient way to implement tables [4].

A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term. At the entry point we have the root node. Internal nodes store tokens in terms and leaf nodes specify the end of terms.

Each root-to-leaf path represents a term described by the tokens labelling the nodes traversed. For example, the tokenized form of the term $p(X, q(Y, X), Z)$ is the stream of 6 tokens: $p/3, X, q/2, Y, X, Z$. Two terms with common prefixes will branch off from each other at the first distinguishing token.

The internal nodes of a trie are 4-field data structures. One field stores the node's token, one second field stores a pointer to the node's first child, a third field stores a pointer to the node's parent and a fourth field stores a pointer to the node's next sibling. Each internal node's outgoing transitions may be determined by following the child pointer to the first child node and, from there, continuing through the list of sibling pointers.

To increase performance, YapTab enforces the *substitution factoring* [4] mechanism and implements tables using two levels of tries - one for subgoal calls, the other for computed answers. More specifically, the table space of YapTab 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 predicate's *subgoal trie*;
– each different subgoal call is represented as a unique path in the subgoal trie, starting at the predicate's table entry and ending in a *subgoal frame* data structure, with the subgoal sub-terms being stored within the path's internal nodes;
– the *subgoal frame* data structure acts as an entry point to the *answer trie*;
– each different subgoal answer is represented as a unique path in the *answer trie*, starting at a particular leaf node and ending at the subgoal frame. Oppositely to subgoal tries, answer trie paths hold just the substitution terms for unbound variables in the corresponding subgoal call.
– the leaf's child pointer of answers are used to point to the next available answer, a feature that enables answer recovery in insertion order. The subgoal frame has internal pointers that point respectively to the first and last answer on the trie. Whenever a variant subgoal starts consuming answers, it sets a pointer to the first leaf node. To consume the remaining answers, it must follow the leaf's linked chain, setting the pointer as it consumes answers along the way. Answers are loaded by traversing the answer trie nodes bottom-up.

An important point when using tries to represent terms is the treatment of variables. We follow the formalism proposed by Bachmair *et al.* [9], where each variable in a term is represented as a distinct constant. Formally, this corresponds to a function, $numbervar()$, from the set of variables in a term t to the sequence of constants $VAR0, ..., VARN$, such that $numbervar(X) < numbervar(Y)$ if X is encountered before Y in the left-to-right traversal of t.

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 for the constant a. The subgoal frame is inserted as a leaf, waiting for the answers. Then, the subgoal $f(Y, 1)$ is inserted. It shares one common node with $f(X, a)$ but, having

a different second argument, a new subgoal frame needs to be created. Next, the answers for $f(Y, 1)$ are stored in the answer trie as their values are computed.

Two facts are noteworthy in the example of Fig. 1. First, the example helps us to illustrate how the state of a subgoal changes throughout execution. A subgoal is in the *ready* state when its subgoal frame is added. The state changes to *evaluating* while new answers are being added to the subgoal's table. When all possible answers are stored in the table, the state is set to *complete*. When not complete or evaluating, the subgoal is said to be *incomplete*.

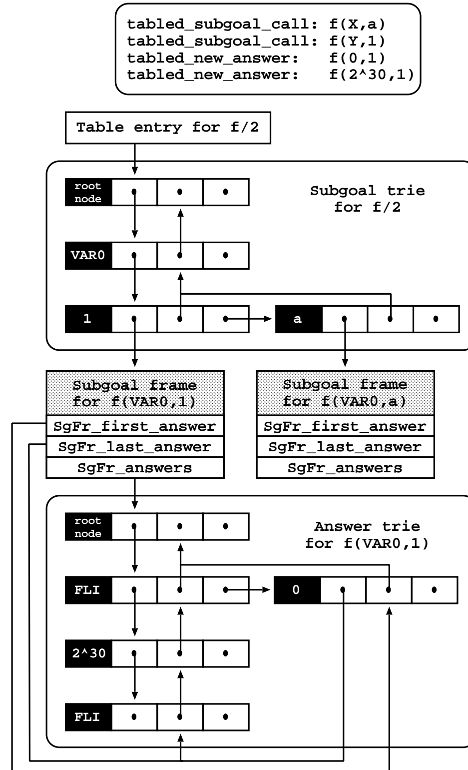


Fig. 1. The table space organization

Second, some terms require special handling from YapTab, such as the 2^{30} integer term, appearing in the picture surrounded by the FLI (*functor long integer*) markers. This is due to the node's design. Internally, YapTab terms are 32 (or 64) bit words, divided in two areas: type and value. Hence, the value part, which is used for data storage purposes, is always smaller than an equally typed C variable. Henceforth, we shall refer to terms whose value fits the proper slot, such as small integer and atom terms, as *short atomic terms* and to all the others, such as floating-point and large integers, as *long atomic terms*. Given a long atomic term, YapTab **(i)** ignores the term's type and forces its (full) value into a single node, or **(ii)** splits the value into as many pieces as needed, assigning a new node to each and every one of these. Either way, the type bits are used to store value data so additional nodes, holding special type markers, are used to delimit the term's value.

3 The DBTab Relational Model

The choice of effective models for table space representation is a hard task to fulfill. The relational model is expected to allow quick storage and retrieval operations over tabled answers in order to minimize the impact on YapTab's performance. Conceptually, DBTab's relational model is straightforward – every tabled subgoal is mapped into a database relation. The relation's name bears the predicate's functor and arity and its attributes match the predicate's arguments. Two distinct database models spawned from this basic concept.

3.1 Datalog Model - Tries as Sets of Subgoals

Given an in-memory tabled subgoal, the complete set of its known answers may be regarded as a set of ground facts that are known to hold in a particular execution context. In other words, known subgoal answers are regarded as Datalog facts. Thus, and henceforth, this schema is referred to as the *Datalog model*.

In this model, each tabled predicate p/n is mapped into a relational table SESSIONk_PN, where k is the current session identifier. Predicate arguments become the ARGi integer fields and the INDEX field is used to store the answer's index in the answer chain of the in-memory subgoal trie. One can further refine this model by choosing one of the two following alternatives.

Inline Values As previously stated, DBTab maps tabled predicates directly into database relations bearing the same name and argument number. Thus, predicates appear as tables, subgoals appear as records within those tables and subgoal sub-terms appear as constants in record fields. Unfortunately, the different nature and size of logical atoms presents a problem. Table fields may be given a primitive type big enough to accommodate all kinds of atoms, but that seems to be too expensive and unpractical.

Instead, each of the predicate's terms is mapped into a set of fields, one for each supported atomic type. Whenever storing a subgoal answer, the term's type is checked to decide which table field must be used to hold the term's value; the remaining ones are set to NULL. This policy also simplifies the reverse decision process – since all of these fields are regarded as a single predicate's argument, the value for the stored term is always the non-NULL one. Due to their size, short atomic terms may be directly stored within the corresponding ARGi integer fields. This small optimization enables the shortening of each field set by one. Figure 2 displays an example of such a relation for the $f/2$ tabled predicate, introduced back in Fig. 1.

SESSION*k*_F2
(pk) INDEX
ARG1
FLT1
LNT1
ARG2
FLT2
LNT2

Fig. 2. The inline values model

Separate Values The inline values model produces highly sparse tables, since a lot of storage space is wasted in records containing mostly NULL-valued fields. To take full advantage of the normalized relational model, the subgoal's answer terms may be stored in several auxiliary tables according to their logical nature. Figure 3 shows the separate values relational model for the previously stated $f/2$ tabled predicate.

SESSION*k*_F2
(pk) INDEX
ARG1
ARG2

SESSION*k*_FLOATS
(pk) TERM
VALUE

SESSION*k*_LONGINTS
(pk) TERM
VALUE

Fig. 3. The separate values model

Again, short atomic terms are directly stored within the ARGi integer fields, thus reducing the number of required auxiliary tables. Long atomic terms, on

the other hand, are stored in auxiliary tables, such as SESSIONk_LONGINTS and SESSIONk_FLOATS. Each long atomic value appears only once and is identified by a unique key value that replaces the term's value in the proper field ARGi of SESSIONk_F2. The key is in fact a YapTab term, whose type bits tell which auxiliary table holds the actual primitive term's value whereas the value bits hold a sequential integer identifier. Despite the practical similarity, these keys may not be formally defined as foreign keys – notice that the same ARGi field may hold references to both auxiliary tables.

3.2 Hierarchical Model - Tries as Sets of Nodes

Despite its correct relational design, the Datalog model drifts away from the original concept of tries. From the inception, tabling tries where though of as compact data structures. This line of reasoning can also be applied to any proposed data model.

Storing the complete answer set extensionally is a space (and time) consuming task, hence one should draw inspiration from the substitution factoring of tabling tries [4], keeping only answer tries in the database. This not only significantly reduces the amount of data in transit between the logical and the database engines, but also perfectly suits YapTab's memory management algorithm [6], which cleans answer tries from memory but maintains subgoal tries intact.

The challenge is then to figure out a way to mimic the hierarchical structure of a trie in the flat relational database environment. Tries may be regarded as partially ordered sets of nodes ordered by reachability. Hence, mapping relations must be defined in such a way that its tuples contain enough information to preserve the nodes' hierarchy. Such a model is henceforth referred to as the *hierarchical model*.

Arguably, the most simple and compact way to represent a trie is through a database relation like the one presented in Fig. 4, where every node knows only its own order, its parent's order, and its own token. Of all possible numbering methods, we feel that assigning nodes' order while traversing the trie in post-order is the most appropriate, since it will eventually lead to a reduction of the number of data transactions, while maintaining the in-memory insertion order at the same time.

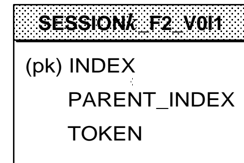


Fig. 4. The hierarchical model

In this context, mapping relations no longer hold entire predicate answer sets, rather each relation stands for a single subgoal call. Hence, one must be able to tell the different subgoals apart. A straightforward way to establish this distinction is to encapsulate the arguments' values as a suffix to the relation's name. The chosen naming convention dictates that each argument is identified by a capital letter, indicating the type of the term, and a number, indicating its order within the subgoal. The possible letters are I and L for standard and long integers, D for doubles, A for (string) atoms and V for (unbound) variables. Figure 4 displays an example of such a relation for the $f/2$ tabled predicate.

4 Extending YapTab's Design

From the beginning, our goal was to introduce DBTab without disrupting the previously existing YapTab framework. With that in mind, we have striven to keep both tabling semantics and top-level predicates syntax intact. Rather, top-level predicates and internal instructions were internally modified to include calls to the developed API functions.

The dumping of in-memory answer tries into the database is triggered by the *least recently used* algorithm when these tables are selected for destruction. Recordset reloading takes place when calls to tabled subgoals occur and their tables have been previously dumped.

Communication between the YapTab engine and the RDBMS is mostly done through the MySQL C API for prepared statements. Two of the major table space data structures, *table entries* and *subgoal frames*, are expanded with new pointers (see Fig. 5) to proprietary `PreparedStatement` wrapping data structures. These structures hold specialized queries that enable the dumping and loading of subgoals to and from the database. All SQL statements are executed in a transactional context.

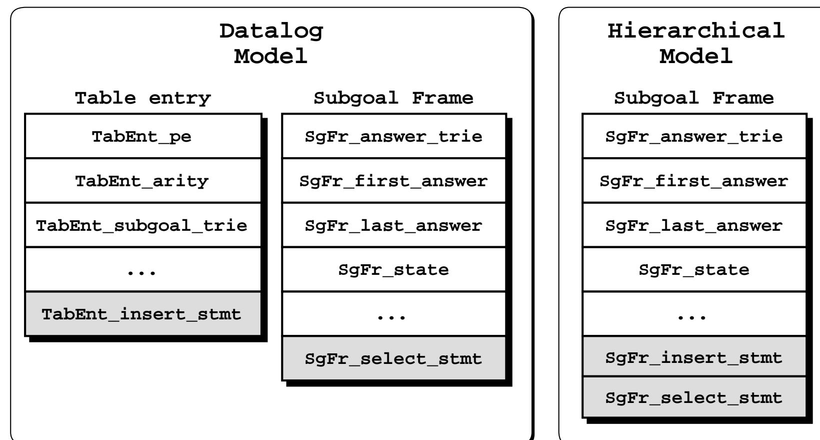


Fig. 5. New table frames' fields

Two new predicates are added to manage database session control. To start a session one 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. The connection handler is usually created resorting to the Myddas package [10] and the session identifier can be either a unbound variable or a positive integer term, meaning respectively that a new session is to be initiated or a previously created one is to be reestablished. The `tabling_kill_session/0` terminates the currently open session.

YapTab's directive `table/1` is used to set up logic predicates for tabling. The choice of a storage model determines the behaviour of DBTab's expanded version of this built-in predicate. In the hierarchical model, it issues a table

creation statement and exists. In the Datalog model (both alternatives), an additional step creates an INSERT prepared statement. The statement, placed inside the predicate's table entry in the new `TabEnt_insert_stmt` field, allows the introduction of any subgoal answer for the tabled predicate. Two examples of such statements for the inline and separate values models, labeled **(1)** and **(2)** respectively, are shown in Fig. 6.

```
(1) INSERT IGNORE
    INTO SESSIONk_F2(INDEX,ARG1,FLT1,LNT1,ARG2,FLT2,LNT2)
    VALUES (?,?,?,?,?,?,?);
(2) INSERT IGNORE
    INTO SESSIONk_F2(INDEX,ARG1,ARG2)
    VALUES (?,?,?);
(3) INSERT IGNORE
    INTO SESSIONk_F2_V0I1 (INDEX, PARENT_INDEX, TOKEN)
    VALUES (?,?,?);
(4) SELECT DISTINCT ARG1,LNT1
    FROM SESSIONk_F2
    WHERE ARG2=14;
(5) SELECT DISTINCT F2.ARG1 AS ARG1, L.VALUE AS LNT1
    FROM SESSIONk_F2 AS F2
          LEFT JOIN SESSIONk_LONGINTS AS L ON (F2.ARG1=L.TERM)
    WHERE F2.ARG2=14
    ORDER BY F2.INDEX;
(6) SELECT INDEX, TOKEN
    FROM SESSIONk_F2_V0I1
    WHERE PARENT_INDEX=?;
```

Fig. 6. Prepared statements for $f(Y, 1)$

The `abolish_table/1` built-in predicate is used to destroy in-memory tables for tabled predicates. The DBTab expanded version of this predicate frees all prepared statements used to manipulate the associated database tables and drops them if they are no longer being used by any other instance.

Besides the referred predicates, the tabling instruction set must suffer some small modifications. The instruction responsible for the subgoal insertion into the table space is adapted to create a SELECT prepared statement. This statement, kept inside the subgoal frame structure in the new `SgFr_select_stmt` field, is used to load answers for that subgoal. Examples of such statements for the inline values, separate values and hierarchical models, labeled **(4)**, **(5)** and **(6)** respectively, are shown in Fig. 6. The selection query is tailored to reduce the search space – this is accomplished by including all bound terms in comparison expressions within the WHERE sub-clause[3]. Notice that SELECT statement of the Datalog models, examples **(4)** and **(5)**, bear a DISTINCT option. This modifier is added to enforce the uniqueness of retrieved answers.

[3] Here, we are considering a 32 bit word representation where the integer term of value 1 is internally represented as 14.

In the hierarchical model, an additional step creates the INSERT prepared statement. This statement is placed inside the subgoal `SgFr_insert_stmt` new field. Unlike the Datalog models, the hierarchical model's table name identifies unequivocally the subgoal for which its stands, rendering the statement useless to all other subgoals belonging to the same predicate. Example **(3)** of Fig. 6 illustrates the hierarchical INSERT statement.

4.1 Exporting Answers

When the system runs out of memory, the *least recently used* algorithm is called to purge some of tables from the table space. The algorithm divides subgoal frames in two categories, *active* and *inactive*, according to their execution state. Subgoals in *ready* and *incomplete* states are considered *inactive*, while subgoals with *evaluating* state are considered *active*. Subgoals in the *complete* state may be either active or inactive.

Figure 7 illustrates the memory recovery process. Subgoal frames corresponding to inactive subgoals are chained in a double linked list. Two global registers point to the most and least recently used inactive subgoal frames. Starting from the least recently used, the algorithm navigates through the linked subgoal frames until it finds a memory page that fits for recovery. Only tables storing more than one answer may be elected for space recovery (completed nodes with a yes/no answer are ignored). Recall that only the answer trie space is recovered. Rocha [6] suggests that for a large number of applications, these structures consume more that 99% of the table space.

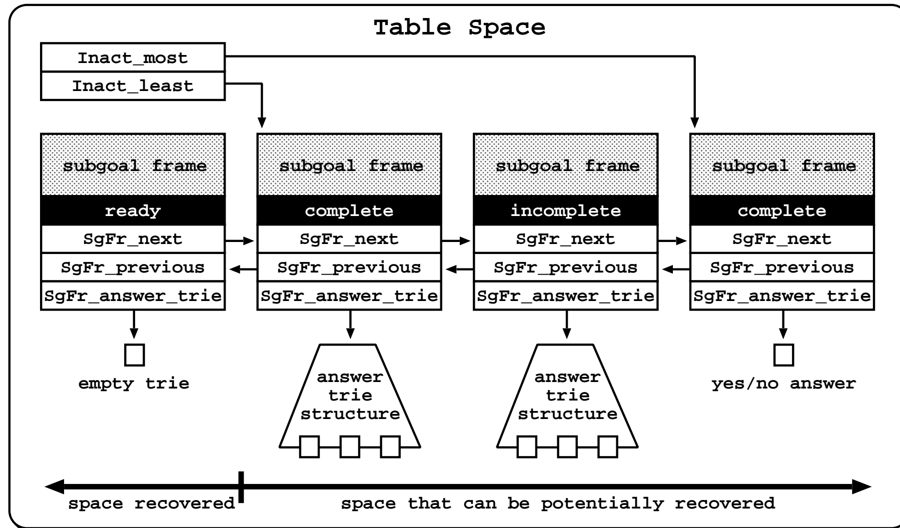


Fig. 7. The least recently used algorithm in action

DBTab cuts in before the actual table elimination. At that point, a specific API function begins a new data transaction. From this point on, implementations differs according to the used database model.

The hierarchical model implementation traverses the trie in post-order, numbering and storing nodes as it goes along. The complete set of records belonging to an answer trie is stored recursively. Starting from a level's last node (first record), the next trie level is stored before proceeding to the previous node (next record). The table records are created as the trie nodes are visited.

The Datalog model implementation first traverses the subgoal trie branch bottom-up (starting from the subgoal frame), 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 and control proceeds cycling through the terms stored within the answer trie nodes. Again, the remaining NULL statement parameters are bound to the current answer terms and the prepared statement is executed, until no more answers remain to be stored. The last step consists in constructing the specific SELECT statement to be used to fetch the answers for the subgoal, whose ground terms are used to refine the search condition within the WHERE sub-clause.

With both models, the transaction is committed and the subgoal frame changes its internal state to *stored*, signalling that its known answers now reside on a database table. Finally, the least recently used algorithm resumes its normal behaviour by removing the answer trie from memory.

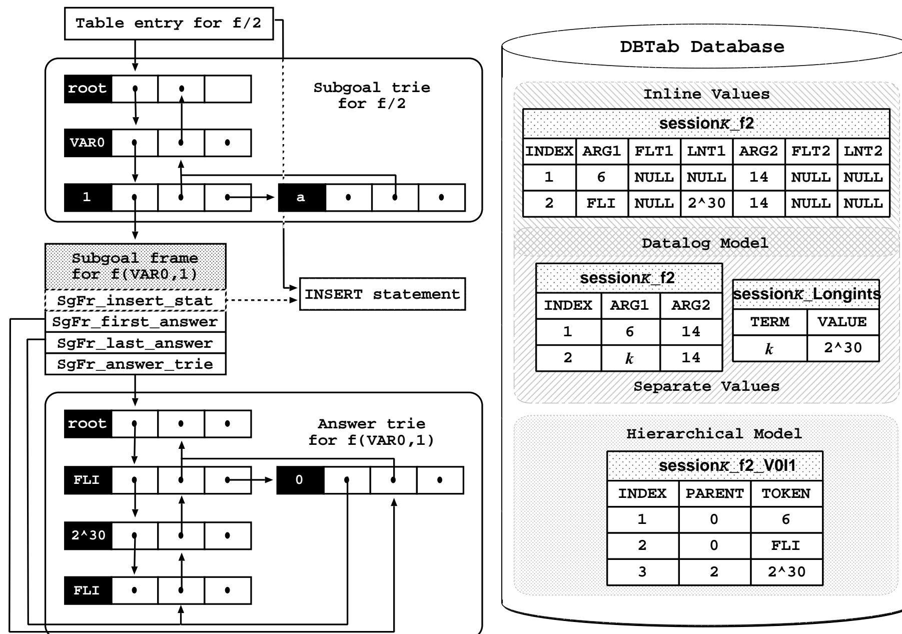


Fig. 8. Exporting $f(Y, 1)$ using both storage models

Figure 8 illustrates the final result of the described process using both storage models. The branch for the integer term of value 0 is stored first, and the

branch for the long integer term 2^{30} is stored next[4]. Notice how, in the separate values model, the ARG1 field of the second record holds the key for the auxiliary table record. Also, notice how the hierarchical model saves one record omitting the second FLI type marker. Recall that, in this model, the INSERT prepared statement is placed at the subgoal frame level.

4.2 Importing Answers

After memory recovery, a repeated call to a pruned subgoal may occur. Normally, YapTab checks the subgoal frame state and, should it be ready, recomputes the entire answer trie before proceeding. Again, DBTab cuts in before this last step. An API function uses the SELECT prepared statement residing in the subgoal frame to retrieve the previously computed answers from the database and uses them to rebuild the answer trie. Figure 9 shows, in the right boxes, the resulting *views* for each storage model. The way in which the returned recordset is interpreted differs from model to model.

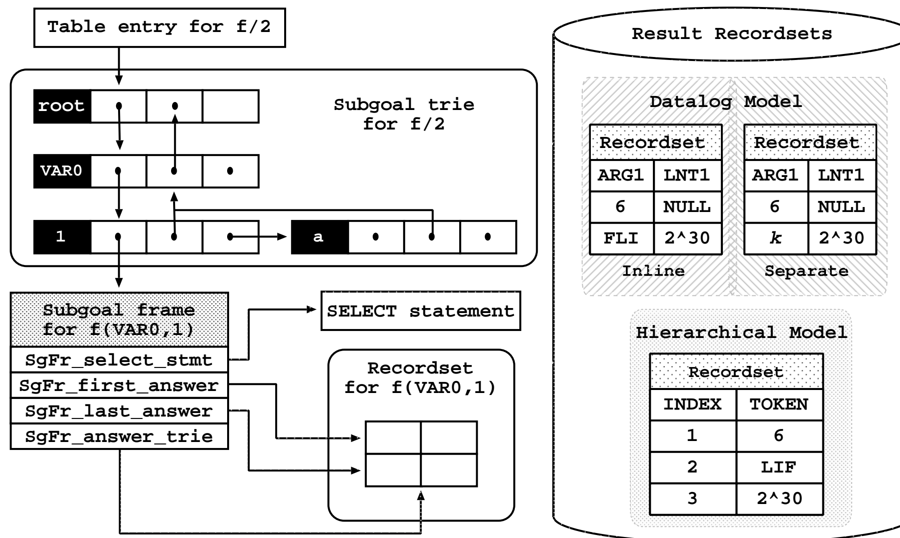


Fig. 9. Importing $f(Y, 1)$ using both storage models

The hierarchical model reloading operation is straightforward. The recordset is loaded one level at a time from the database. All records within a certain level are fetched in insertion order simply by ordering the selection by the index field. The complete set of records belonging to an answer trie is fetched recursively. Starting from a level's last node (first record), the next trie level is fetched before proceeding to the previous node (next record). The trie nodes are created as the records are consumed. A special feature in this process involves the setting and

[4] Again, we are considering a 32 bit word representation where the integer terms of value 0 and 1 are internally represented respectively as 6 and 14.

reading of a type flag whenever an opening long atomic term delimiter is found. The flag is set to the proper primitive type so that, once the entire term is reloaded from the database, the closing delimiter is inserted into the trie. This allows a small efficiency gain, since it reduces tables' and views' sizes.

The Datalog models focus on the ARG*i* fields, where no NULL values can be found. Additional columns, placed immediately to the right of the ARG*i* fields, are regarded as possible placeholders of answer terms only when the first field conveys long atomic type markers. In such a case, the non-NULL additional field value is used to create the specific YapTab term. The separate values alternative uses the auxiliary table's key value as marker, while the inline values alternative uses the special type markers directly.

Complete answer-sets present themselves as a special case of the import operation. Since they will no longer change, one wonders if trie reconstruction is the best approach for most of the cases. In both Datalog alternatives, two possible strategies may be used to supply the YapTab engine with the answers fetched by the SELECT statement.

Rebuilding the Answer Trie The retrieved recordset is used to rebuild the answer trie and is then discarded. Records are sequentially traversed and each one of them provides the sub-terms of a complex term – the answer. This term is passed to the ancillary table look-up/insertion function that places the new answer in the respective subgoal table. By the end of the process, the entire answer trie resides in the table space and the recordset can then be released from memory. This approach requires small changes to YapTab and mostly makes use of its already implemented API.

Browsing the Record-Set In this strategy, the retrieved recordset is kept in memory. Since the answer tries will not change once completed, all subsequent subgoal calls may fetch their answers browsing the recordset through offset arithmetics. Figure 9 illustrates how the ancillary YapTab constructs are used to implement this 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 recordset. When consuming answers, the first record's offset along with the subgoal frame address are stored in a *loader choice point*[5]. 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 recordset, an invalid offset is reached, the loader choice point is discarded and execution fails, thus terminating the on-going evaluation. It is expected that this approach may introduce a small gain in performance and memory usage because **(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.

[5] 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.

5 Initial Experimental Results

Three main series of tests were performed in YapTab, both with and without the DBTab extensions. This meant to establish the grounds for a correct comparison of performances. The environment for our experiments was an Intel Pentium®4 2.6GHz processor with 2 GBytes of main memory and running the Linux kernel-2.6.18. DBTab was deployed on a MySQL 5.0 RDBMS running an InnoDB engine. Both engines were running on the same machine.

```
% connection handle creation stuff
:- consult('graph.pl').
:- tabling_init_session(Conn,Sid).

% top query goal
go(N) :- statistics(walltime, [Start,_]), benchmark(N),
         statistics(walltime, [End,_]), Time is End-Start,
         writeln(['WallTime is ',Time]).
benchmark(1):- path(A,Z), fail.
benchmark(1).
benchmark(2):- write_path(A,Z), fail.
benchmark(2).

% path(A,Z) and write_path(A,Z) succeed if there is a path between A and Z
:- table path/2.
path(A,Z):-path(A,Y), edge(Y,Z).
path(A,Z):-edge(A,Z).

:- table write_path/2.
write_path(A,Z):- write_path(A,Y), edge(Y,Z), writeln(['(',A,',',Z,')']).
write_path(A,Z):- edge(X,Y), writeln(['(',A,',',Z,')']).
```

Fig. 10. Test program

A path discovery program, presented in Fig. 10, was used to measure both YapTab and DBTab performances in terms of answer time. The program's main goal is to determine all existing paths for the graph presented in Fig. 11, starting from any node in the graph.

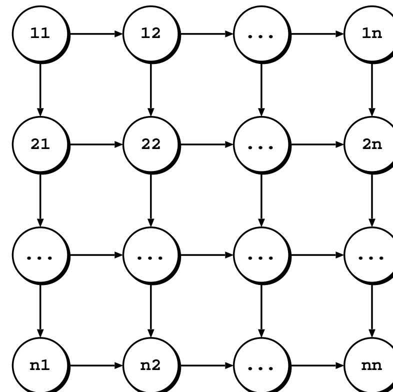


Fig. 11. Test graph

The *go*/1 predicate is the top query goal. It determines the benchmark predicates' performance by calculating the difference between the program's up-time before and after the benchmark execution. Benchmark predicates are *failure driven loops*, i.e., predicates defined by two clauses – a first one executes a call to the tabled goal followed by a fail-

ure instruction and a second one ensures the successful completion of the top query goal.

Two tabled predicates were used to determine all existing paths in the loaded graph. Predicate *path*/2, henceforth called *benchmark #1*, helped us to establish minimum performance standards, since its answer-set is relatively easy both to obtain and handle through the use of standard tabling. However, every day problems often require heavy calculations or some sort of input/output operation. With this in mind, predicate *write_path*/2, henceforth called *benchmark #2*, was introduced. It basically results from the addition of a call to an output predicate, *writeln*/1, at the very end of both clauses of *path*/2. Our aim was to determine if DBTab could improve the overall performance of the main program when such heavy operations were performed. Note that from the start, it was our firm conviction that, for programs mostly based on *pure* tabled computations, DBTab would not bring any gain than improvement in program termination due to the overhead induced by the YapTab/MySQL communication.

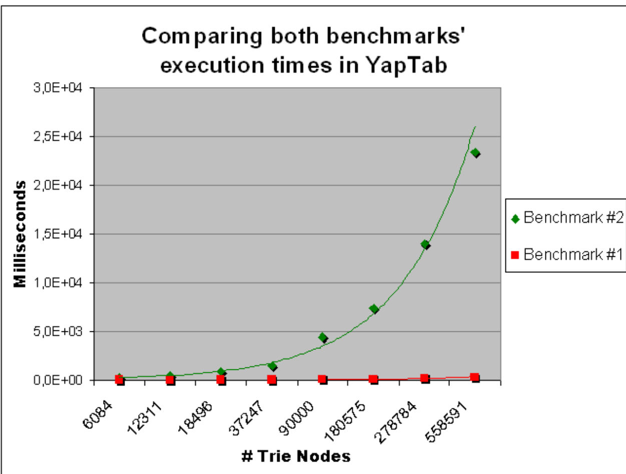
For a relatively accurate measure of execution times, the node's type changed twice and the graph's size changed twenty-five times. Both benchmark programs were executed twenty-five times for each combination of type and size. The average of every combination's run-time was measured in milliseconds.

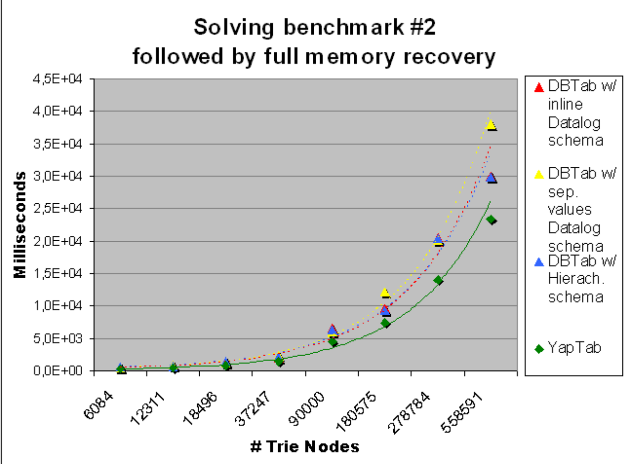
Early tests uncovered a major bottleneck during answer storage. Sending an answer a time to the database significantly damaged performance. As a result, DBTab now incorporates a MySQL feature that enables clustered insert statements, sending answer-tries to the database in clusters of 25 tuples.

Figure 12 shows three charts that visually summarize the results. The first chart shows that input/output operations introduce a significant decrease in performance. When comparing both benchmark's performance in YapTab, one observes that in average *benchmark #2* executes 87 times slower than the *benchmark #1*. The chart shows that the gap widens as the number of nodes in the answer trie grows.

The second chart in Fig. 12 shows first call execution times for *benchmark #2*. In order to measure the impact of DBTab on YapTab's performance, a full table dump is forced after each answer-set is complete. As expected, some overhead is induced into YapTab's normal performance. For the inline values model, the median storage overhead is 0.3 times the amount of time required to execute *benchmark #2*, within an interval ranging from 0.2 to 0.6 times. For the separate values alternative, the median overhead grows to 0.6 times, within an interval ranging from 0.3 to 0.7 times. For the hierarchical model, the median induced overhead is of 0.5 times the computation time of *benchmark #2*, within an interval ranging from 0.2 to 0.7 times. Additionally, label types also have impact on the execution times. In average, answer set generation using long integer labels is 1.6 times slower than when integer labels are used. No doubt, this is due to the larger number of nodes in the answer trie, which result in a larger number of transacted answers.

The third chart in Fig. 12 shows subsequent call execution times. From its observation, one can learn that retrieval operations have little cost in terms of





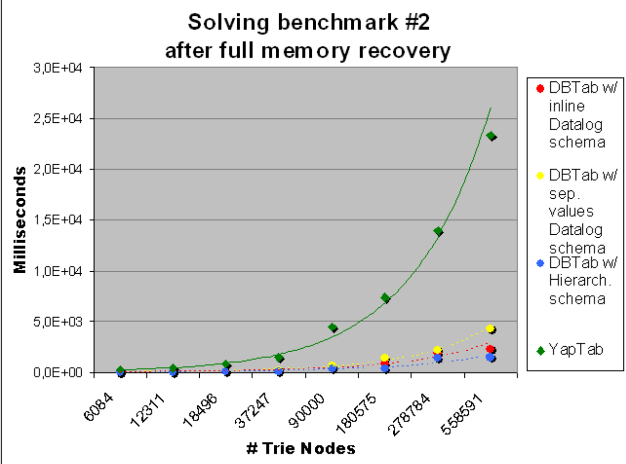


Fig. 12. Test results

performance and indeed introduce some speed-up when compared to full answer-set re-computation. For the inline values model, the minimum, median and maximum retrieval times are 0.1 times the amount of time required to execute *benchmark #2*. The performance of the other alternative is similar, although in the worst-case scenario the median retrieval time may rise to 0.2. For the hierarchical model, the median retrieval time is 0.1 times the computation time of *benchmark #2*, within an interval ranging from (nearly) 0.0 to 0.2 times.

The performances of trie traversal and recordset browsing are compared in Fig. 13. In this figure, it is possible to observe that recordset browsing times grow with the graph size. For the inline values model, it takes in average 3.8 times more the time required to traverse the respective answer trie. For the separate values, it takes only 3.6 times. However, this technique introduces a considerable gain in terms of used memory, as the recordset in-memory size is in average 0.4 of the correspondent answer trie size for both Datalog alternatives.

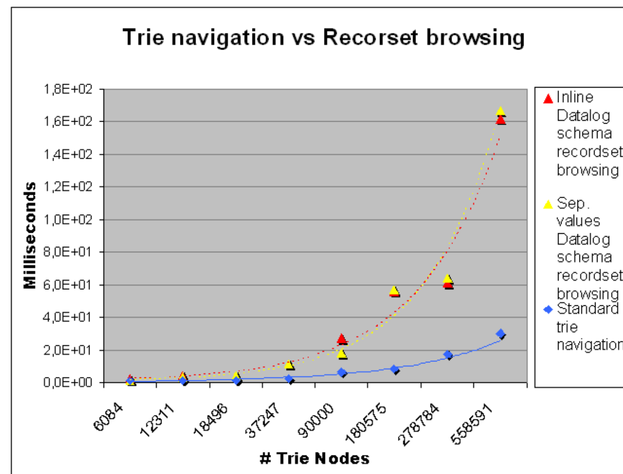


Fig. 13. Trie navigation versus recordset browsing

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.

In all performed tests, data transaction performances revealed a similar pattern. Storage revealed to be an expensive operation. In all cases, this operation's cost exceeded that of recomputing the same answer set. The inline values model was always the fastest, the hierarchical model the second fastest and the separate values Datalog model was always the slowest. However, when the computation

involved side-effected operations, this scenario has radically changed and the cost of storing the answer set became quite acceptable.

Things were somewhat different with the retrieval operation. Of all implementations, the hierarchical model was always the fastest and the separate values Datalog model the slowest. This last implementation shows a significant performance decay, no doubt induced by the use of left join clauses in the retrieval select statement. In average, the separate values Datalog model took at least twice the time to retrieve the answer-sets than its inline counterpart. When the answer set evaluation involved no costly operations, reloading answers from the database was obviously slower. On the other hand, when the side-effected operations were introduced, reloading became a quite attractive possibility.

In what regards to term types, integer terms where obviously the simplest and fastest to handle. The other primitive types, requiring special handling, induced significant overheads to both storage and retrieval operations. Atom terms, not considered in the tests, are known to behave as standard integers.

For small answer-sets, recordset browsing might be enough to locate a small initial subset of answers and decide whether that subset is the adequate one or if it should be ignored; in either case, this allows saving both the time and memory resources required for the complete trie reconstruction. However, as answer-sets size increase, this traversal approach performance decays, reaching up to three times the amount of time required to reconstruct the table. In this last case, the only advantage of recordset browsing is the introduced saving in terms of memory requirements.

Our preliminary 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 recordset 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 cover all possibilities for tabling presented by YapTab and extend DBTab to support lists and application terms.

Acknowledgments

This work has been partially supported by projects STAMPA (PTDC/EIA/67738/2006), JEDI (PTDC/EIA/66924/ 2006) and by funds granted to LIACC through the Programa de Financiamento Plurianual, 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. 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. Costa, P., Rocha, R., Ferreira, M.: DBTAB: a Relational Storage Model for the YapTab Tabling System. In: Colloquium on Implementation of Constraint and LOgic Programming Systems. (2006) 95–109
8. Fredkin, E.: Trie Memory. Communications of the ACM **3** (1962) 490–499
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
10. Soares, T., Ferreira, M., Rocha, R.: The MYDDAS Programmer's Manual. Technical Report DCC-2005-10, Department of Computer Science, University of Porto (2005)