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*

CICLOPS 2006, Seattle, Washington, USA, August 2006

Motivation

Problem

- In general, tables are **in-memory** data structures.
- Applications that build **very many or very large tables** can quickly **run out of memory**.

Motivation

- **Problem**
  - In general, tables are **in-memory** data structures.
  - Applications that build **very many or very large tables** can quickly **run out of memory**.
- **Solution I**
  - Have a set of primitives that the programmer can use to dynamically abolish some of the tables.
  - Difficult to use and to decide what are the **potentially useless tables** that should be deleted.

Motivation

Problem

- In general, tables are **in-memory** data structures.
- Applications that build **very many or very large tables** can quickly **run out of memory**.

Solution I

- Have a set of primitives that the programmer can use to dynamically abolish some of the tables.
- Difficult to use and to decide what are the **potentially useless tables** that should be deleted.

Solution II

- YapTab's memory management algorithm that **automatically** recovers space from the **least recently used tables** when the system runs out of memory.

Motivation

- **Problem with Solutions I and II**
  - We lose the already found answers for the deleted tables.
  - When a repeated call appears, we need to **recompute** the set of answers from the beginning.

Motivation

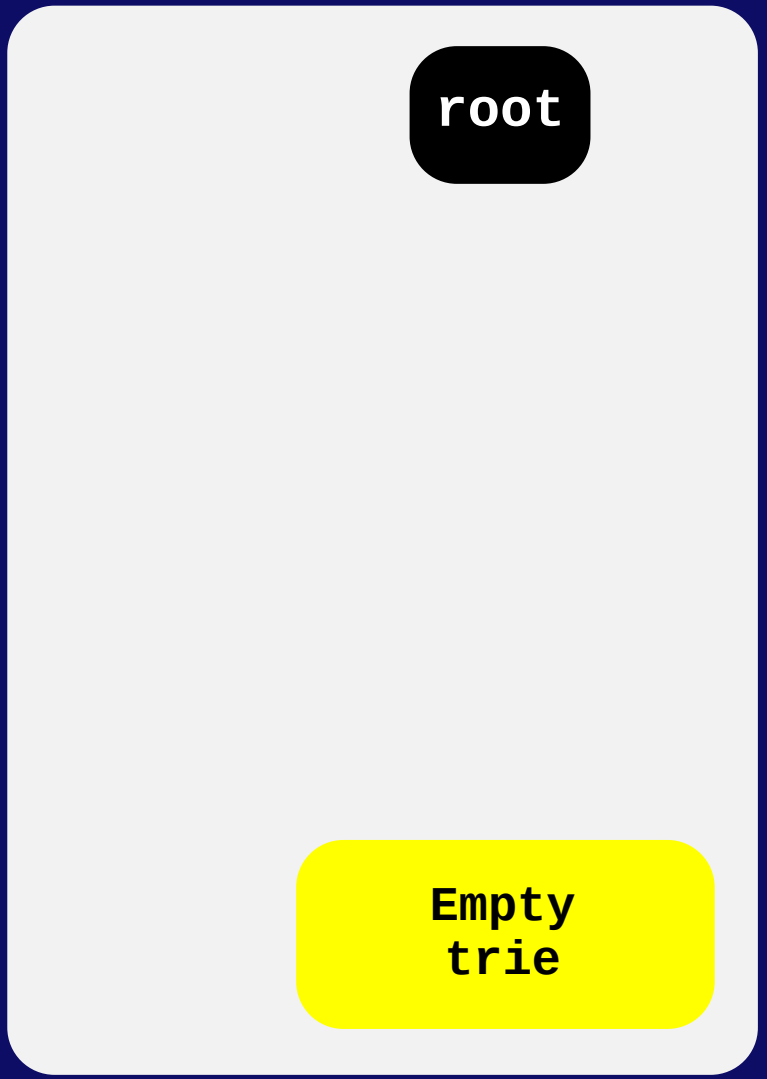
- **Problem with Solutions I and II**
  - We lose the already found answers for the deleted tables.
  - When a repeated call appears, we need to **recompute** the set of answers from the beginning.
- **Our Proposal**
  - Store tables externally using a **relational database management system**.
  - When 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 when the system runs out of memory, but instead of deleting tables, we can use it to decide what tables we should move to database storage.

Table Space

- **Can be accessed to:**
  - Look up if a subgoal is in the table, and if not insert it.
  - Look up if a newly found answer is in the table, and if not insert it.
  - Load answers for repeated subgoals.
- **Implementation requirements:**
  - Fast look-up and insertion methods.
  - Compactness in representation of logic terms.

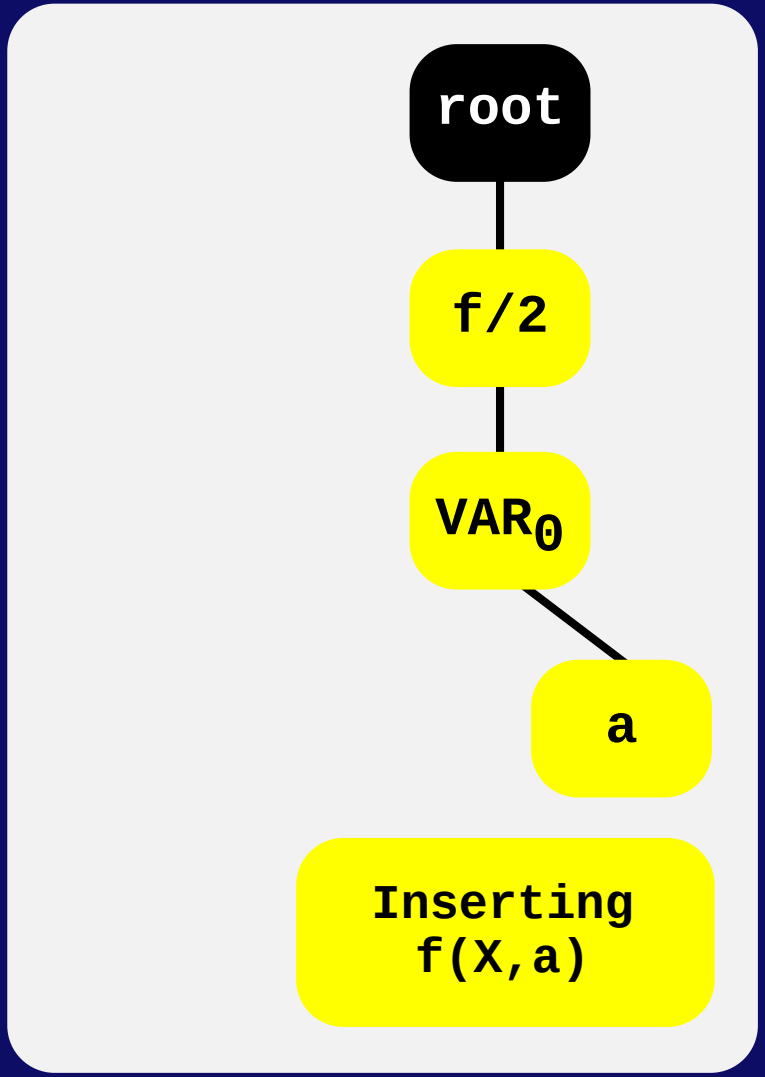
Using Tries to Represent Terms

- Tries are trees in which common prefixes are represented only once.



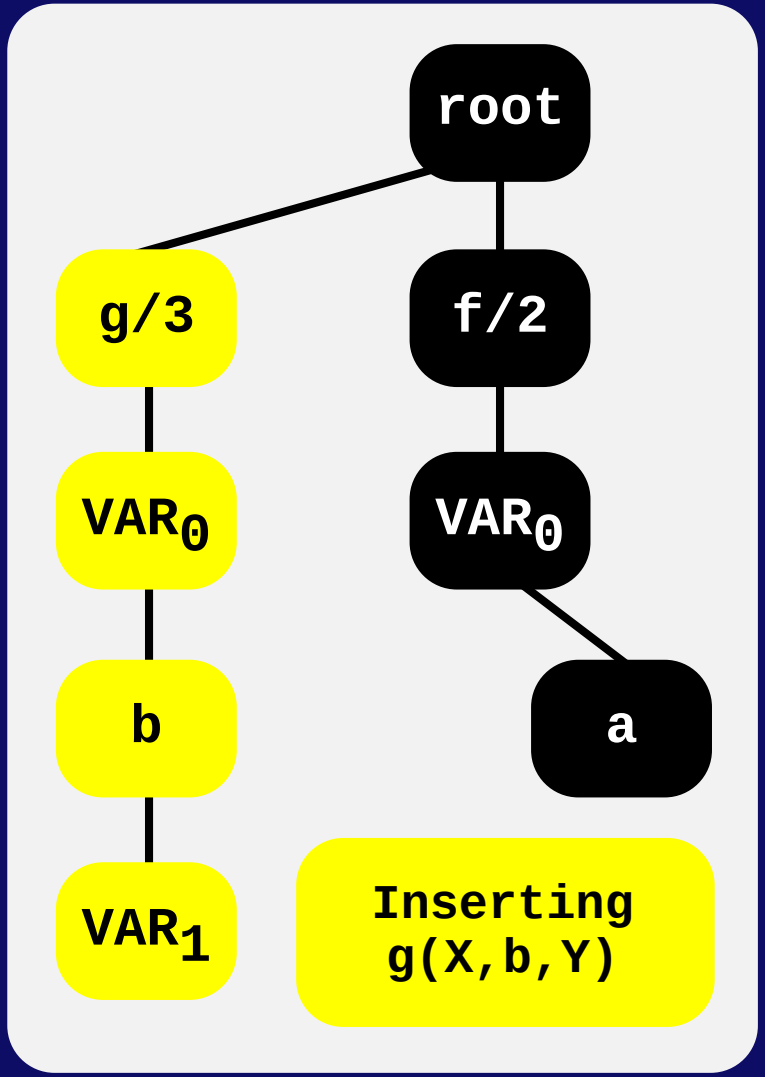
Using Tries to Represent Terms

- Tries are trees in which common prefixes are represented only once.
  - The entry point is called the root node, internal nodes represent symbols in terms and leaf nodes specify completed terms.



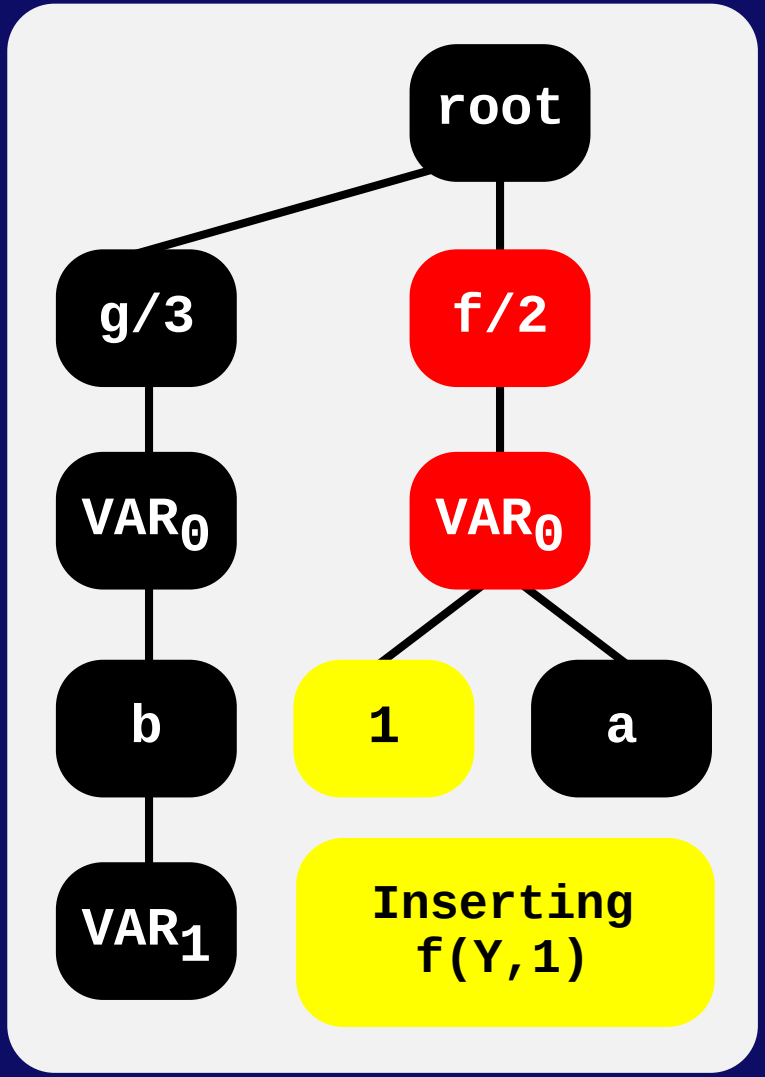
Using Tries to Represent Terms

- Tries are trees in which common prefixes are represented only once.
  - The entry point is called the root node, internal nodes represent symbols in terms and leaf nodes specify completed terms.
  - Each different path through the nodes in the trie corresponds to a term.



Using Tries to Represent Terms

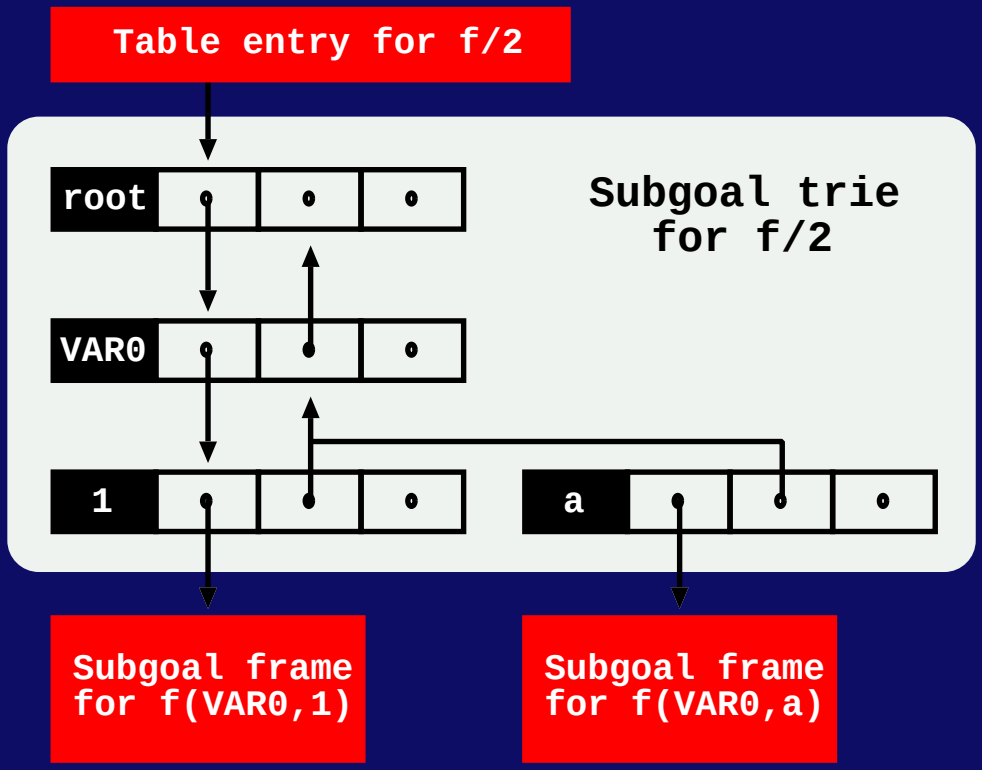
- Tries are trees in which common prefixes are represented only once.
  - The entry point is called the root node, internal nodes represent symbols in terms and leaf nodes specify completed terms.
  - Each different path through the nodes in the trie corresponds to a term.
  - Terms with common prefixes branch off from each other at the first distinguishing symbol.



Using Tries to Organise the Table Space

Subgoal Trie Structure

- Stores the tabled subgoal calls.
- Starts at a table entry and ends with subgoal frames.
- A subgoal frame is the entry point for the subgoal answers.



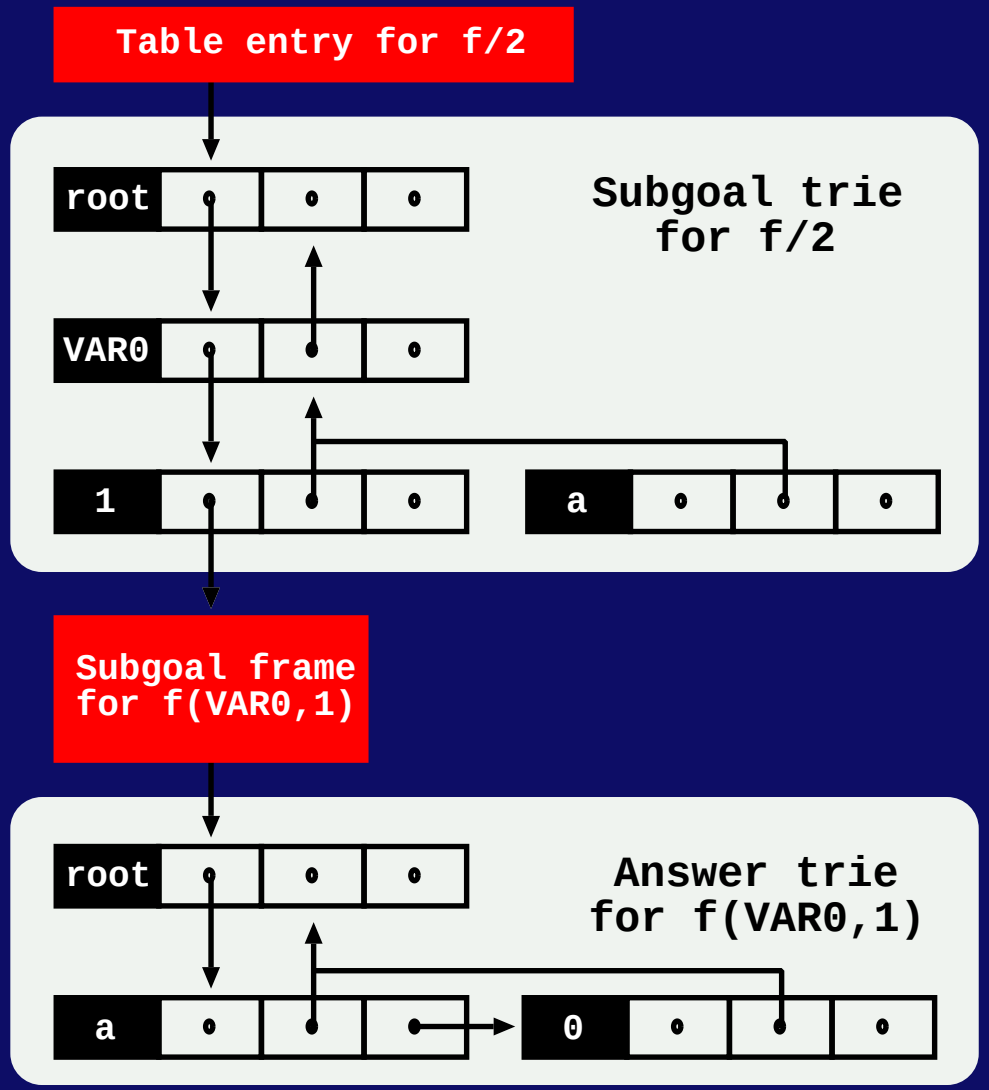
Using Tries to Organise the Table Space

Subgoal Trie Structure

- Stores the tabled subgoal calls.
- Starts at a table entry and ends with subgoal frames.
- A subgoal frame is the entry point for the subgoal answers.

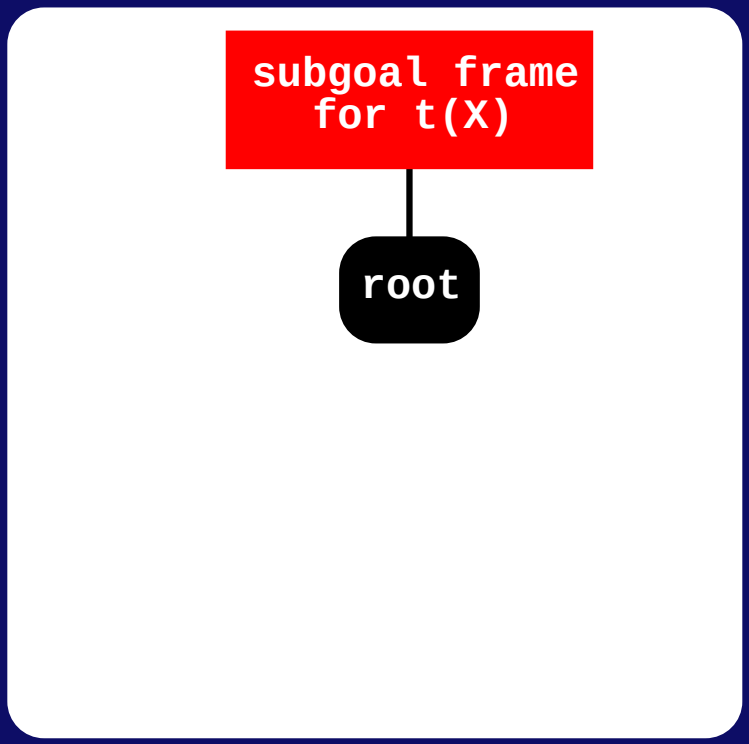
Answer Trie Structure

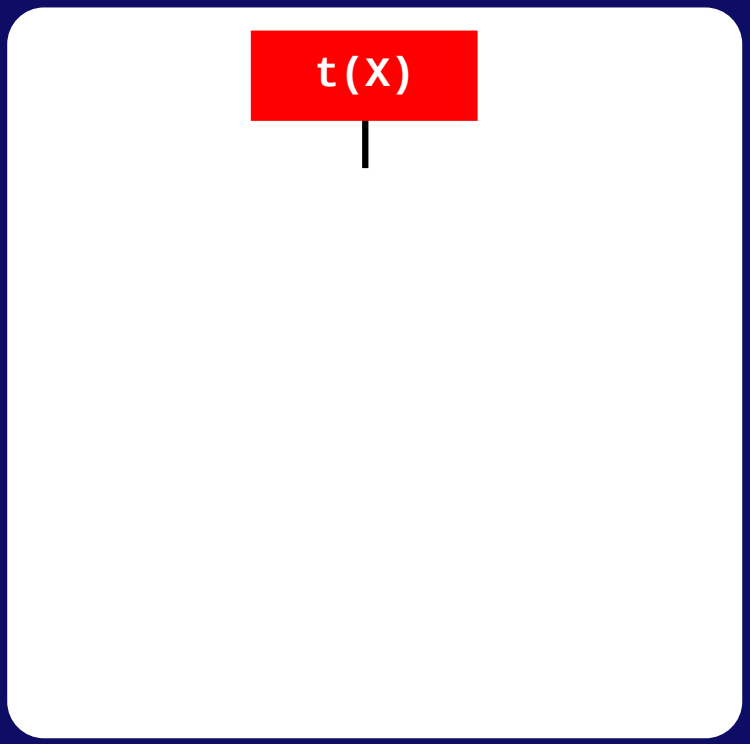
- Stores the subgoal answers.



Using Tries to Organise the Table Space

- **Answer Trie Structure**
  - Leaf nodes are chained in insertion time order.

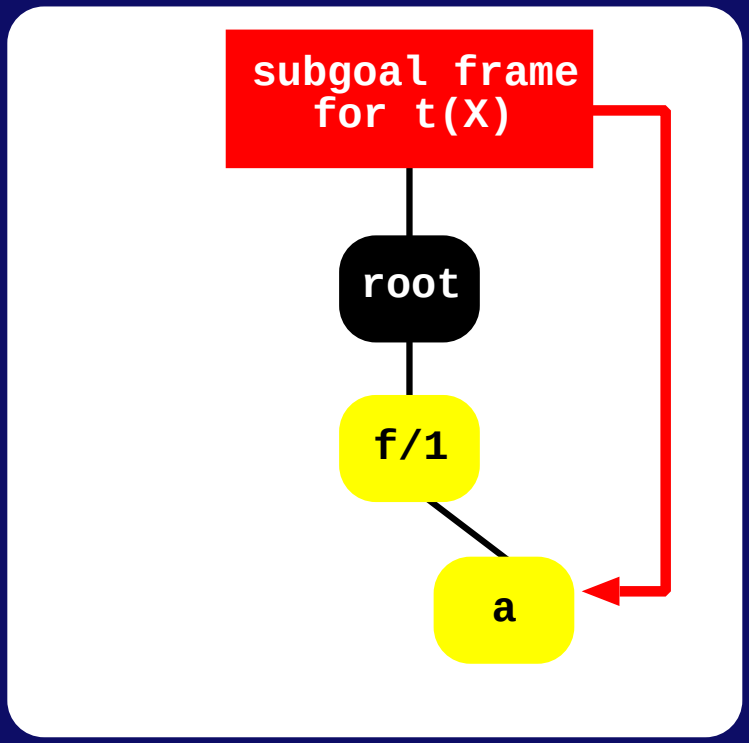


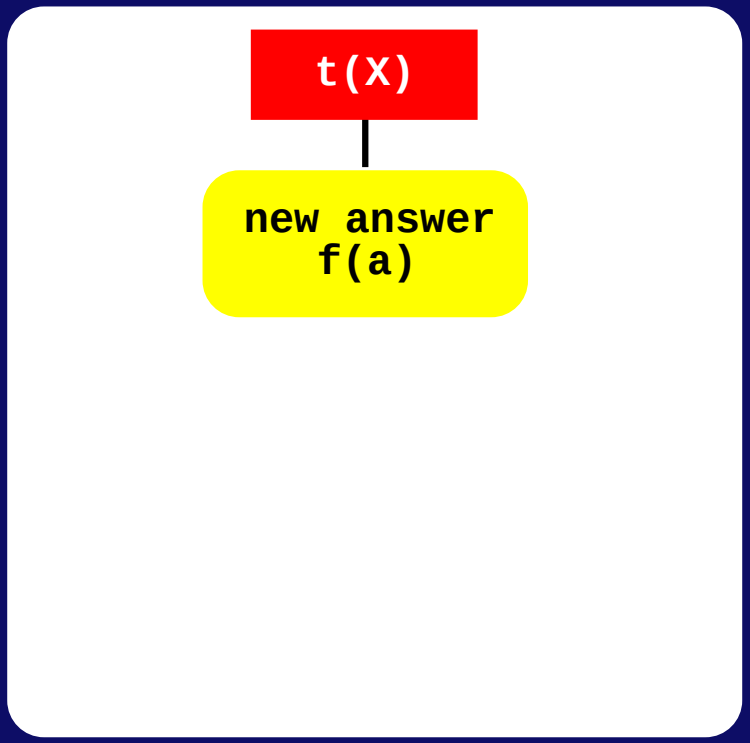


Using Tries to Organise the Table Space

Answer Trie Structure

- Leaf nodes are chained in insertion time order.

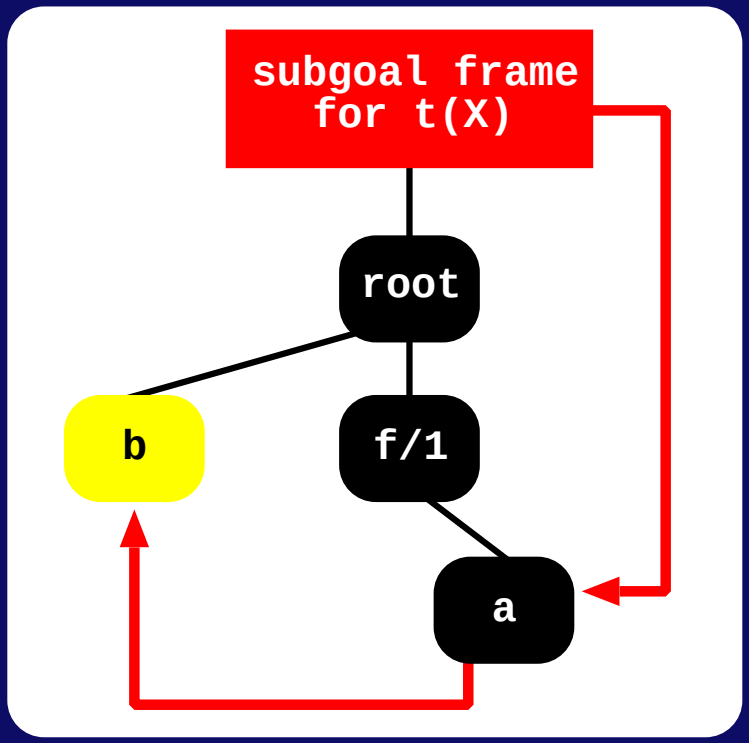


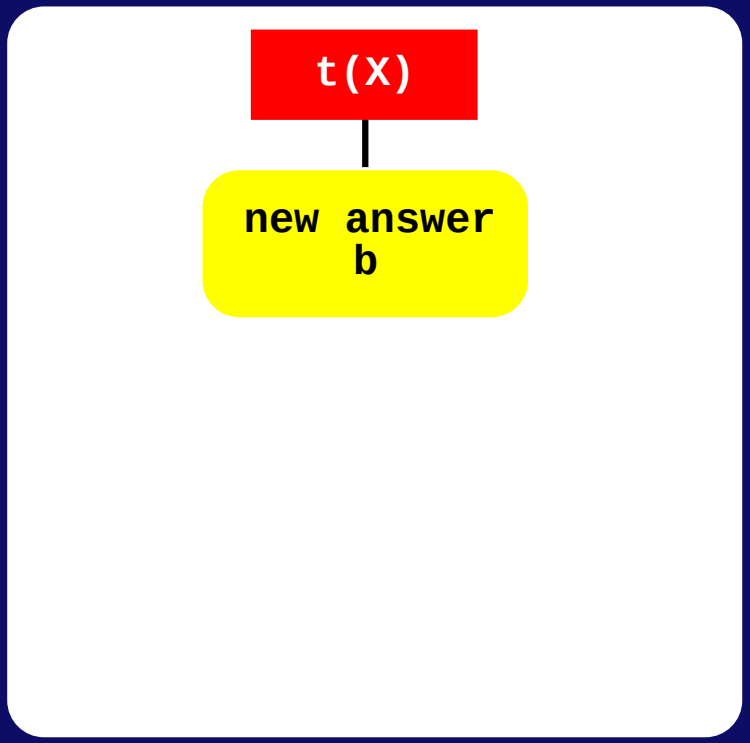


Using Tries to Organise the Table Space

Answer Trie Structure

- Leaf nodes are chained in insertion time order.

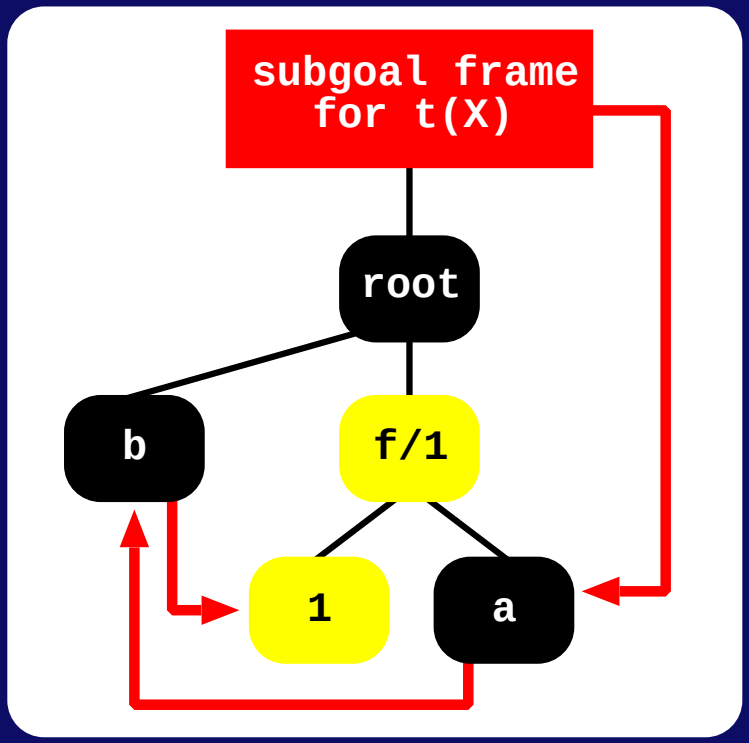


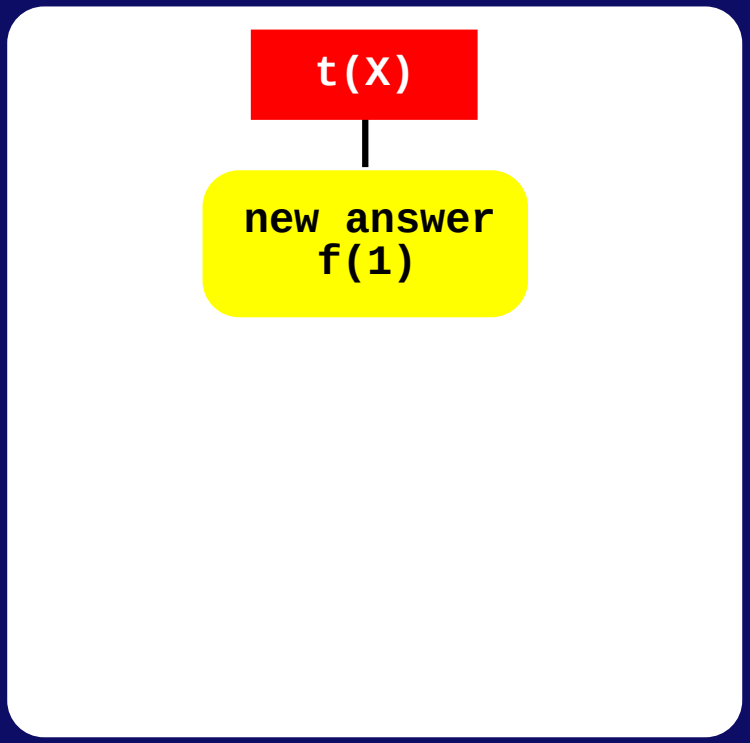


Using Tries to Organise the Table Space

Answer Trie Structure

- Leaf nodes are chained in insertion time order.

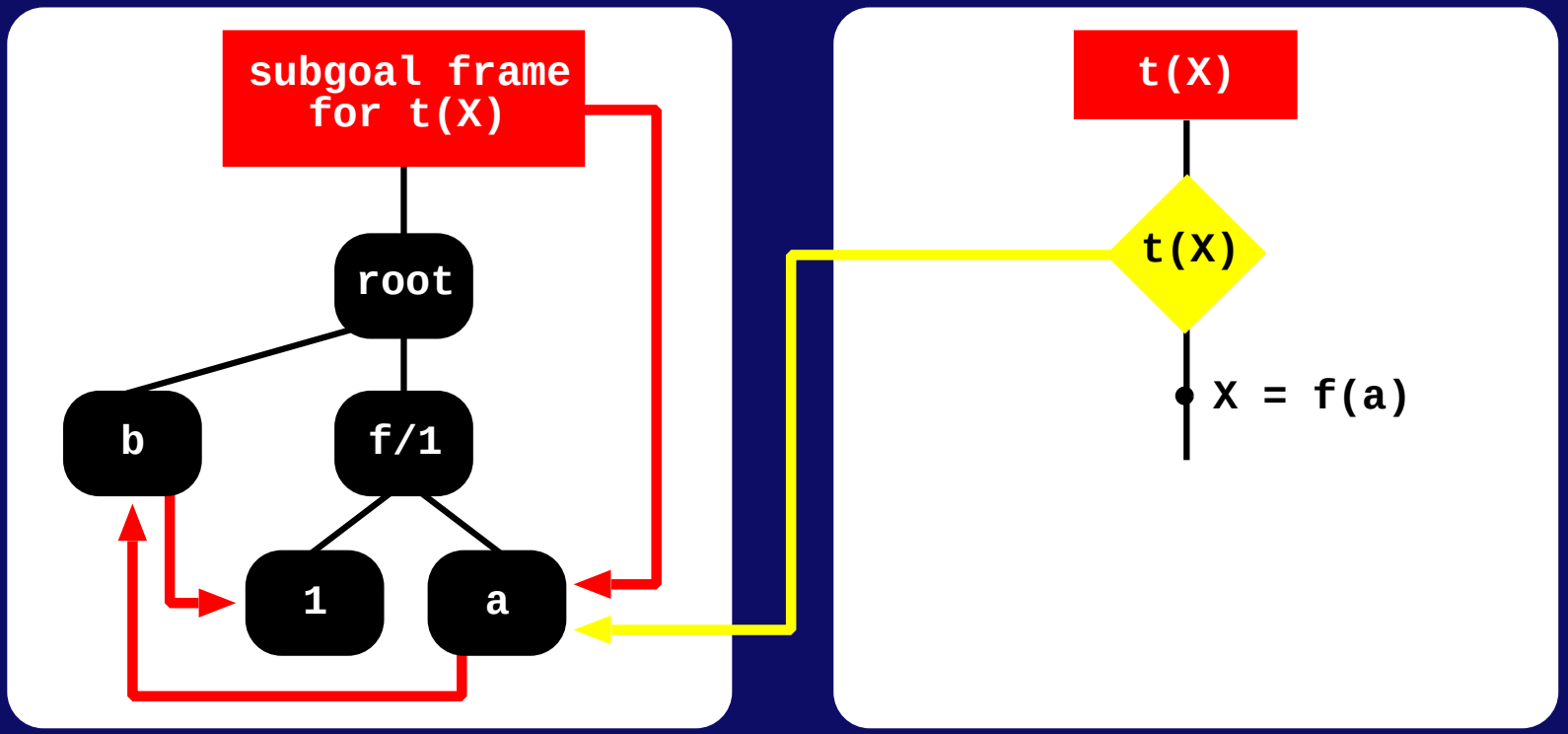




Using Tries to Organise the Table Space

Answer Trie Structure

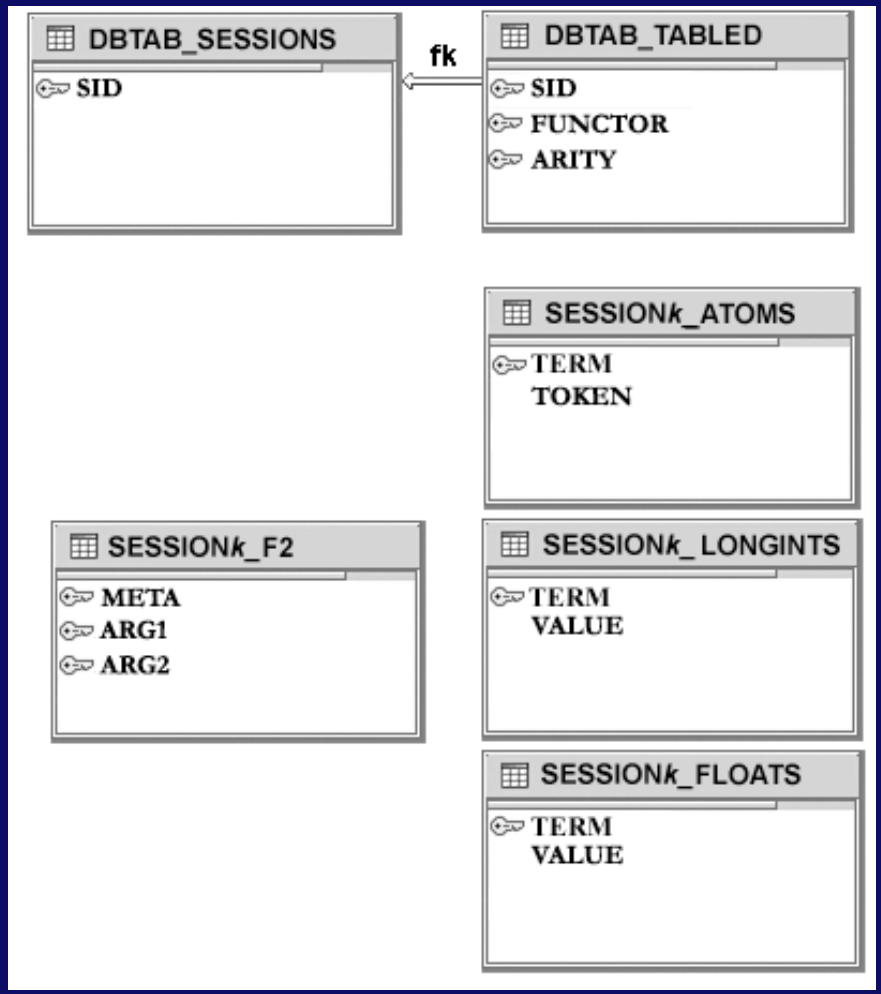
- Leaf nodes are chained in insertion time order.
- Repeated calls keep a reference to the leaf node of the last consumed answer, hence can consume more answers by following the chain.



The DBTAB Relational Storage Model

- **System Tables**
  - DBTAB_SESSIONS: active sessions.
  - DBTAB_TABLED: tabled predicates per session.
- **Predicate Tables (k is the session id)**
  - SESSION*k*_PN: answers for p/n.
- **Auxiliary Tables (k is the session id)**
  - SESSION*k*_ATOMS
  - SESSION*k*_FLOATS
  - SESSION*k*_LONGINTS

The initial implementation of DBTAB only handles integers, atoms and floating-point numbers.



The DBTAB Relational Storage Model

Atomic Terms

- Integer terms within the non-mask part of a term.
- Atom terms (pointers to the internal symbol addressing space).
- Their values are directly stored within the corresponding **ARG*i*** record fields.

Atoms

- The string values for atoms are also stored in the **TOKEN** field of the corresponding **SESSION*k*_ATOMS** table.
- This table is only used to rebuild the previous internal symbol addressing space if reestablishing a session.

The DBTAB Relational Storage Model

Non-Atomic Terms

- Long integer terms (integers larger then the non-mask part of a term).
- Floating-point 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.
- These sequential values are masked as YapTab application terms in order to simplify the loading algorithm.

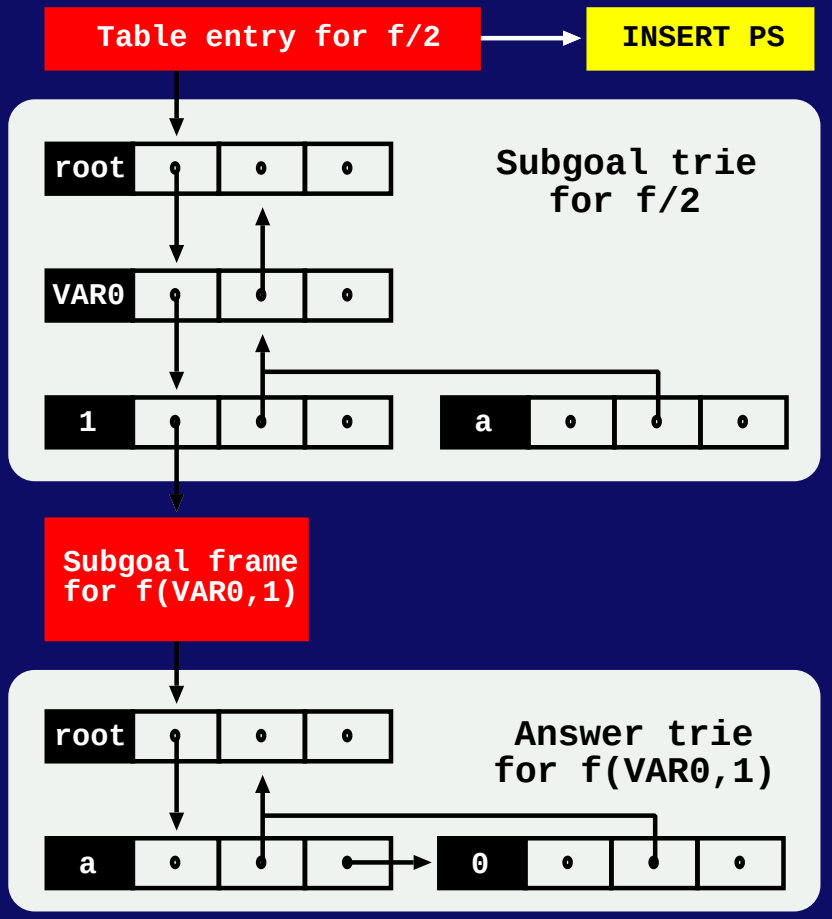
Exporting Answers

- Table entry structure extended with an INSERT prepared statement:

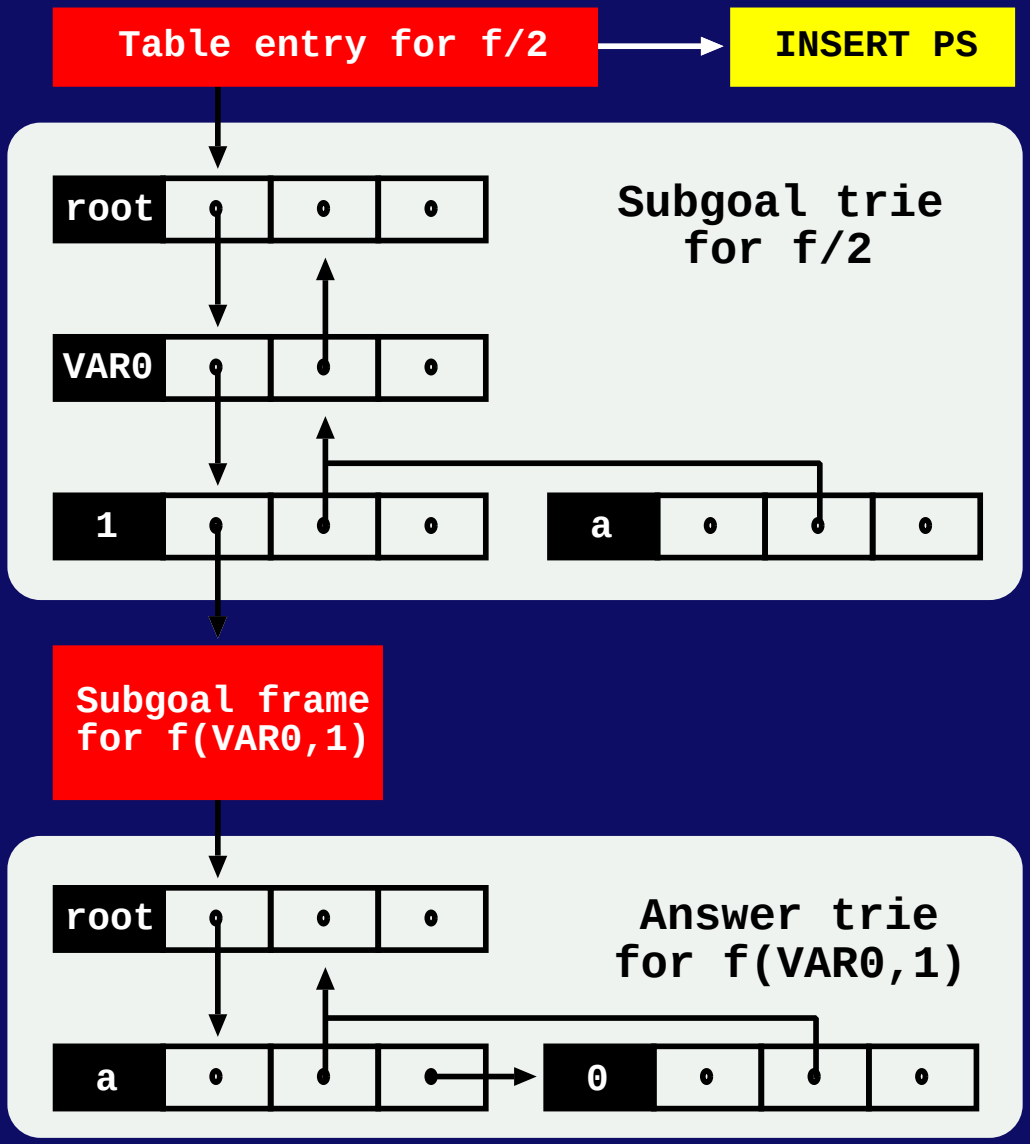
INSERT IGNORE INTO SESSIONk_F2(META,ARG1,ARG2) VALUES (?,?,?);

- **Basic Idea**

1. Bind all terms from subgoal trie branch to prepared statement parameters.
2. Bind all terms from answer trie branch to the remaining prepared statement parameters.
3. Execute prepared statement.
4. Goto 2 until no more answers.
5. Store meta-data for the subgoal trie branch.



Exporting Answers



DBTAB

SESSIONk_F2

META	ARG1	ARG2
0	6	22
0	0xff04	22
1	0	22

non-atomic terms?

number of free variables

SESSIONk_ATOMS

TERM	TOKEN
0xff04	a

Importing Answers

- Subgoal frame structure extended with two SELECT statements:

SELECT ARG1 FROM SESSIONk_F2 WHERE META=1 AND ARG2=22;

SELECT ARG1 FROM SESSIONk_F2 WHERE META=0 AND ARG2=22;

- **Basic Idea**

1. Execute statement 1 to retrieve meta-data info.
2. Execute statement 2 to load answers.
3. If no rows are retrieved for statement 2, set subgoal frame pointers to NULL. Otherwise, set first and last answer pointers to the first and last rows respectively.

Importing Answers

- Subgoal frame structure extended with two SELECT statements:

SELECT ARG1 FROM SESSIONk_F2 WHERE META=1 AND ARG2=22;

SELECT ARG1 FROM SESSIONk_F2 WHERE META=0 AND ARG2=22;

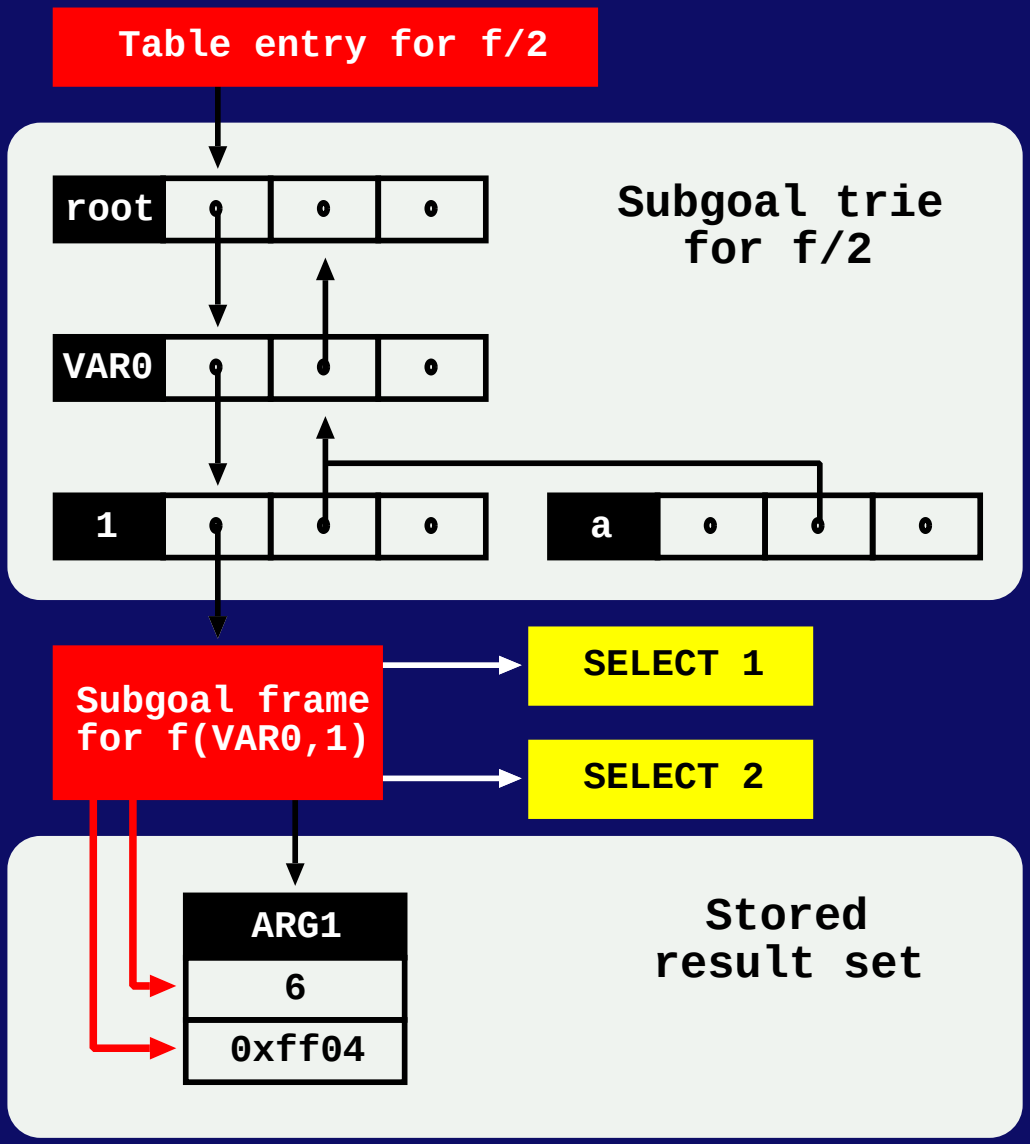
- **Basic Idea**

1. Execute statement 1 to retrieve meta-data info.
2. Execute statement 2 to load answers.
3. If no rows are retrieved for statement 2, set subgoal frame pointers to NULL. Otherwise, set first and last answer pointers to the first and last rows respectively.

- Statement 2 with floating-point values on **ARG1**:

```
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 F2.ARG2 = 22;
```

Importing Answers



DBTAB

SESSIONk_F2

META	ARG1	ARG2
0	6	22
0	0xff04	22
1	0	22

non-atomic terms?

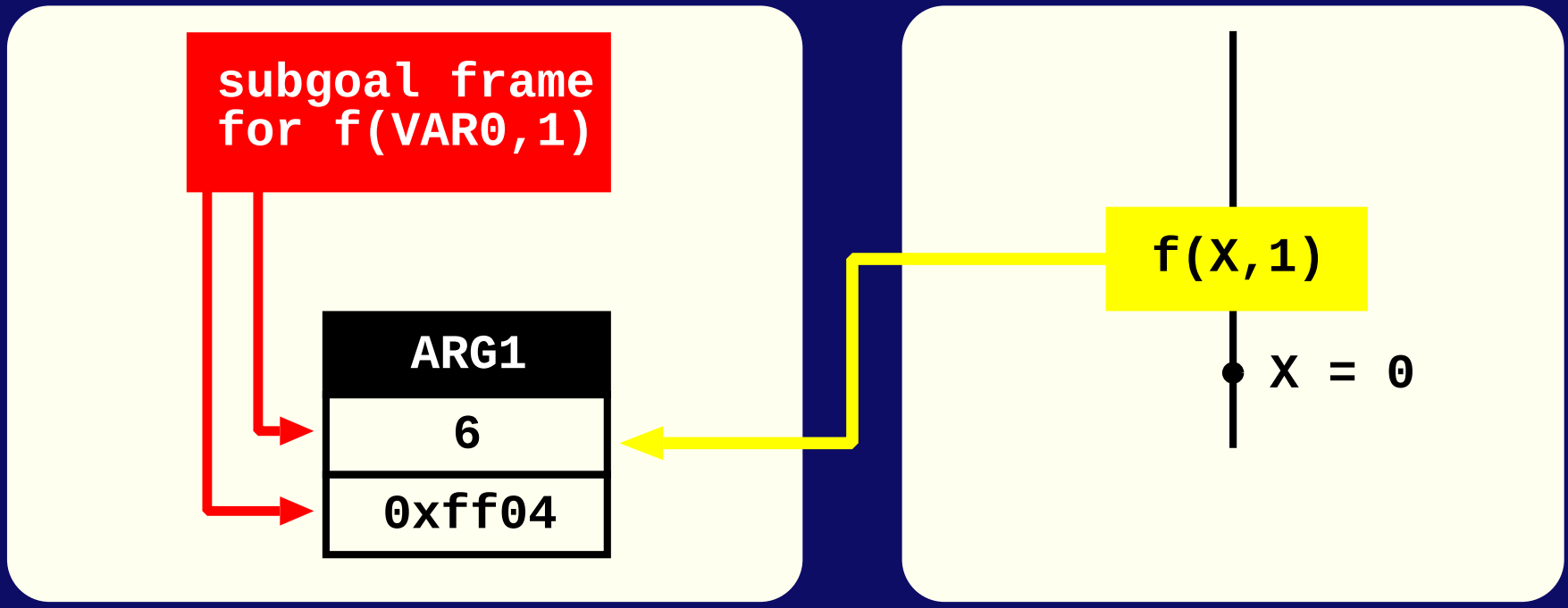
number of free variables

SESSIONk_ATOMS

TERM	TOKEN
0xff04	a

Importing Answers

- Repeated calls now keep a reference to the **offset** of the last consumed answer.



Preliminary Results

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

Running times in milliseconds

Discussion

- Most of the execution time is spent during trie storage.
- Due to insertion on auxiliary tables, non-integer terms take approximately 3 times more than integer terms to export.
- LEFT JOIN statements when importing answers is also a problem.
- Navigation in stored data-sets is about to 2 times slower than navigation in tries. May become interesting when re-computation for a tabled predicate raises over this factor.
- An important side-effect of DBTAB is that stored data-sets require on average, one third of the memory used by tries to represent the same set of answers.

Further Work

- Evaluate the impact of DBTAB on a more representative set of programs.
- Fit within the context of YapTab's memory management algorithm.
- Expand term representation to lists and functors.
- Reconstruct tries from stored answers (?).