Persistent Tables

David S. Warren

Persistent Tables on Disk

Table of Contents

Summary

This package supports the generation and maintenance of persistent tables stored in data files on disk (in a choice of formats.) Persistent tables store tuples that are computed answers of subgoals, just as internal XSB tables do. Persistent tables allow tables to be shared among concurrent processes or between related processes over time. XSB programmers can declare a predicate to be persistently tabled, and the system will then, when a subgoal for the predicate is called, look to see if the corresponding table exists on disk, and, if it does, read the tuples that are answers for the subgoal on demand from the persistent table. If the persistent table for the subgoal does not exist, the XSB subgoal will be called and the tuples that are returned as answers will be stored on disk, and then returned to the call. Persistent tables cannot be recursively self-dependent, unlike internal XSB tables. Normally the tables are subsumptive tables and abstracted from the original call. They act like (internal) subsumptive tables with call abstraction.

A persistent table can serve to communicate between two XSB processes: a process that requests the evaluation of a subgoal and a sub-process that evaluates that subgoal. This is done by declaring a persistently tabled predicate to have its subgoals be evaluated by a subprocess. In this case, when a persistent table for a subgoal needs to be created, a subprocess will be spawned to compute and save the subgoal answers in the persistent table. The calling process will wait for the table to be computed and filled and, when the table is completed, will continue by reading and returning the tuples from the generated persistent table to the initial calling subgoal.

Persistent tables and internal tables (i.e., normal XSB tables) are independent: a predicate may be persistently tabled but not (internally) tabled, tabled but not persistently tabled, neither or both. In many cases one will want to (internally) table a persistently tabled predicate, but not always.

Persistent tables provide a declarative mechanism for accessing data files. A data file can be used to define a persistent table. I.e., a data file can be used to define a persistent table for a desired goal. The data file format must conform to the format declared for the persistent table for its goal. When this is done, simply invoking the goal will access the persistent table, i.e., the data from the data file. One may, or may not, want to (internally) table this goal. If it is (internally) tabled then this will act similarly to a load_dyn of the original data.

In a similar way, persistent tables can serve as a communications mechanism among processes defined by different programming languages, with the tables accessed and/or generated by the various processes.

This module supports using persistent tables in a "view generation framework." This is done by:

1. Defining a module that contains persistent tabled predicates that correspond to the desired (stored) views.
2. Using `pt_need/1` declarations (see below) to declare table dependencies to support concurrent table evaluation.
3. Running a view-generation process (`pt_fill/1/2`) to compute the desired views by calling XSB processes. The view-generation process will "pre"-compute the requred tables in a bottom-up order, using multiple concurrent processes as appropriate (and

declared.) Since no XSB persistently tabled predicate will be called until after all the persistent tables that it depends on have been computed, all XSB predicates will run using those precomputed persistent tables, without blocking and without having to re-compute any of them.

persistent_tables

The `persistent_tables` subsystem maintains persistent tables in directories and files in a subdirectory of the directory containing the source code for a module that defines persistently tabled predicates. The subdirectory is named `xsb_persistent_tables/`. Only predicates defined in a (non-usermod) module can be persistently tabled. For each module with declared persistent tables, there is a subdirectory (whose name is the module name) of `xsb_persistent_tables/` that contains the contents of its tables. In such a subdirectory there is a file, named `PT_Directory.P`, that contains information on all existent persistent tables (stored or proposed.) The subdirectory also contains all the files that store the contents of persistent tables for the given module.

Currently the way a predicate is declared to be persistently tabled is somewhat verbose and redundant. This is because, at this time, there is no XSB compiler (or preprocessor) support for persistent tables, and therefore the user must define explicitly all the predicates necessary for the implementation. In the future, if this facility proves to be useful, and used, we will extend the compiler (or add a preprocessor) to simplify the necessary declarations.

The following packaging and import statements, and predicate definition, are needed once in any module that uses persistent tables:

```
:- packaging:bootstrap_package('persistent_tables','persistent_tables').
:- import table_persistent/5, pt_call/1 from persistent_tables.
:- export ensure_<Module>_loaded/0.
ensure_<Module>_loaded.
```

The specific ensure_<Module>_loaded/0 predicate (where <Module> is the actual name of the module) is called by the system when it is required that the module be loaded.

A persistent table for predicate `Pred/K` is declared and defined as follows:

```
:- export Pred/K, Pred_ptdef/K.
:- table_persistent(PredSkel,ModeList,TableInfo,ProcessSpec,DemandGoal).
PredSkel :- pt_call(PredSkel).
Pred_ptdef(....) :- ... definition of Pred/K ....
```

`PredSkel` indicates a most-general goal for the predicate `Pred/K`.

As can be seen, the user must define an auxiliary predicate of the same arity as the persistently tabled predicate, whose name is the original predicate name with `"_ptdef"` appended. This predicate is defined using the clauses intended to define `Pred/K`. `Pred/K` itself is defined by the single clause that calls the persistent-tabling meta-predicate `pt_call/1`. This meta-predicate will generate subgoals for `Pred_undef/K` and call them as is required.

The arguments of the `table_persistent/5` declaration are as follows:

- `PredSkel` : is the goal whose instances are to be persistently tabled. Its arguments must be distinct variables.
- `ModeList` : a list of mode-lists (or a single mode-list.) A mode-list is a list of constants, `+`, `t`, `-`, and `-+` with a length equal to the arity of `Goal`. The mode indicates constraints on the state of its corresponding argument in a subgoal call. A `"-"` mode indicates that the corresponding position of a call to this goal may be bound or free and is to be abstracted when filling the persistent table; a `"+"` mode indicates that the corresponding position must be bound and is not abstracted, and so a separate persistent

table will be kept for each call bound to any specific constant in this argument position; a `"t"` mode indicates that this argument must be bound to a `"timestamp"` value. I.e., it must be bound to an integer obtained from the persistent tabling system that indicates the snapshot of this table to use. (See `add_new_table/2` for details on using timestamps.) A `"-+"` mode indicates that the corresponding argument may be bound or free, but on first call, it will be abstracted and a separate table will be constructed for each value that this argument may take on. So it is similar to a `"-"` mode in that it is abstracted, but differs in that it generates multiple tables, one for each distinct value this argument takes on. This can be used to split data into separate files to be processed concurrently.

There may be multiple such mode-lists and the first one that a particular call of `Goal` matches will be used to determine the table to be generated and persistently stored. A call does *not* match a mode-list if the call has a variable in a position that is a `"+"` in that mode-list. If a call does not match any mode-list, an error is thrown. Clearly if any mode list contains a `t` mode, all must contain one in the same position. (Note: I have not as yet found much need for multiple mode lists.)

- `TableInfo` : a term that describes the type and format of the persistent tables for this predicate. It currently has only the following possibilities:
  - `canonical` : indicates that the persistent table will be stored in a file as lists of field values in XSB canonical form. These files support answers that contain variables. (Except, answers to goals with modes of `-+` must be ground.)
  - `delimited(OPTS)` : indicates that the persistent table will be stored in a file as delimited fields, where OPTS is a list of options specifying the separator (and other properties) as described as options for the predicate `read_dsv/3` defined in the XSB lib module `proc_files`. Goal answers stored in these files must be ground.
- `ProcessSpec` : a term that describes how the table is to be computed. It can be one of the following forms:
  - `xsb` : indicating that the persistent table will be filled by calling the goal in the current xsb process.
  - `spawn_xsb` : indicating that the persistent table will be filled by spawning an xsb process to evaluate the goal and fill the table.
- `DemandGoal` : a goal that will be called just before the main persistently tabled goal is called to compute and fill a persistent table. The main use of this goal is to invoke `pt_need/1` commands (see below) to indicate to the system that the persistent tables that this goal depends on are needed. This allows tables that will be needed by this computation to be computed by other processes. This is the way that parallel computation of a complex query is supported.

  [Note: A future extension may try to automatically generate these goals from the source program, or from a previous execution of the program. The details remain to be designed and implemented...]

The file named `PT_Directory.P` is maintained by the subsystem to keep track of the state of persistent tables for its associated module. It contains facts for two predicates: `table_instance/8` and `table_instance_cnt/2`.

The predicate `table_instance(TId, Goal, Module, Status, GoalArgs, AnsVars, TableInfo, FileName)` contains information on a particular persistent table, as follows:

- `TId` : a unique id for the persistent table. It is unique for the module. It is generated by concatenating the predicate name and a unique number (generated using the fact in `table_instance_cnt/2`.) (The predicate name is actually unnecessary, since the number uniquely identifies the table. The name is included to make it easier for a user to see the goal a table file is associated with.)
- `Goal` : the goal of the persistently tabled predicate that generates this table.
- `Module` : the module of the persistently tabled predicate. (Maybe should eliminate this field, since it is not necessary, the super-directory is now the module, so we need to know it to get here...)
- `Status` : the status of this persistent table. It can be:
  - `generated(DateTimeGen,DateTimeUsed)` : indicating that the table is completed and available for use. `DateTimeGen` is the date and time it was generated. `DateTimeUsed` is the date and time it was most recently used. (Updating this date means writing the PT_Directory.P file more often. Is it worth that overhead to keep this value?)
  - `group_generated(DateTimeGen,DateTimeUsed)` : indicating that this goal generated a group of files, based on a `-+` mode. This fact does not describe a single table but stands for group of tables.
  - `being_generated(Pid,DateTime)` : indicating that the table is in the process of being generated by process with process ID `Pid`, and started generation at `DateTime`.
  - `invalid(DateTime)` : indicating that the generation of this table failed or aborted in some way, at time `DateTime`, and so is not valid. (There is work to do to maintained this field, by catching errors, associating them with the correct Tid, and updating this value to allow propagation of failure.)
  - `needs_generation(DateTime)` : indicating that the table was requested to be generated (at `DateTime`), but no process is currently in the process of generating it. This status is set by `pt_need/1` and is used to support concurrent generaton of persistent tables.
- `GoalArgs` : a list, `Arity` long, of arguments to `Goal`, that generates this persistent table. These are exactly the arguments of `Goal`.
- `GoalVars` : the list of variables in `Goal`. These generate the answer tuples and correspond to the fields in the persistent table.
- `TableInfo` : the table info for this table, as described above for `table_persistent/5`.
- `FileName` : the name of the file containing the table data.

The predicate `table_instance_cnt/2` has one fact that defines two system values: 1) the last number used to name a unique table file, when generating a `TId` for a persistent table. This is incremented every time a new persistent table file is created and used in the name of that file. And 2) a non-negative integer represeting the most-recent version time-stamp used. (See the predicates below for how time-stamps can be used.)

The contents of the persistent tables that are described in `table_instance/8` are stored in files in the same directory as its `PT_Directory.P` file. The files are named "ta-

ble_<TId>.P" (for files containing canonical terms, and .txt for delimited files containing separated values.)

Methodology for Defining View Systems

As mentioned above, persistent tables can be used to construct view systems, i.e., DAGs representing expressions over functions on relations. A relational function is a basic view definition. An expression over such functions is a view system. The leaf relations in the expression are the base relations, and every sub-expression defines a view. A view expression can be evaluated bottom up, given values for every base relation. Independent subexpressions can be evaluated in parallel. Failing computations can be corrected, and only those views depending on a failed computation need to be re-computed.

Sometimes view systems are required to be "incremental". That is, given a completely computed view system, in which the base relations are given and all derived relations have been computed, we are given tuples to add to (and maybe delete from) the given base relations, and we want to compute all the new derived view contents. In many systems such incremental changes to the base relations result in incremental changes to the derived relations, and those new derived relations can be computed in much less time than would be required to recompute all the derived relations starting from scratch with the new (updated) base relations.

To implement a view system in XSB using persistent tables, each view definition is provided by the definition of a persistently tabled predicate. Then given table instances for the base relations, each view goal can be called to create a persistent table representing the contents of the corresponding derived view.

The following describes, at a high level, a methodology for implementing a given view system in XSB using persistent tables.

1. Define the top-level view relations, just thinking Prolog, in a single XSB module. A top-level relation is the ultimate desired output of a view system, i.e., a relation that is normally not used in the definition of another view. Define supporting relations as seems reasonable. Don't worry about efficiency. Use Prolog intuitions for defining relations. Don't worry about incrementality; just get the semantics defined correctly.
2. Now think about bottom-up evaluation. I.e., we use subsumptive tables, so goals will be called (mostly) open, with variables as arguments. Decide what relations will be stored intermediate views. Restructure if necessary to get reasonable stored views.
3. Now make it so the stored views can be correctly evaluated bottom-up, i.e., with an open call. This will mean that the Prolog intuition of passing bound values downward into called predicates needs to be rethought. For bottom-up evalution, all head variables have to be bound by some call in the body. So some definitions may need new body calls, to provide a binding for variables whose values had been assumed to be passed in by the caller.
4. Declare the stored views as table_persistent, and test on relatively small input data. For each table_persistent, decide initially whether to compute it in the given environment or to spawn a process to evaluate in a new process environment.
5. If you don't need incrementality (i.e., given relatively small additions/deletions to the base relations, compute the new derived relations without recomputing results for old

unchanged data): then tune (maybe adding split-compute-join concurrency, using the `-+` mode, as appropriate.) And you're done.

6. If you *do* need incrementality: In principle, the system ought to be able automatically to transform the program given thus far into an incremental version. (See Annie Liu's research.) But at this point, I don't know how to do this ensuring that the reslting performance is close to optimal. (Maybe Annie does, but...) So we will transform the existing program by hand, and we will give "rules-of-thumb" to help in this process.
7. To begin, we will assume that we are only adding new contents to the existing views. Now, for every stored view predicate P in the existing definition, we make two predicates: old_P and delta_P. The predicate old_P will contain the tuples of the existing ... (to be continued...)

Using Timestamps (or version numbers)

The persistent table package provides some support for integer timestamps for versioning of tables. The programmer can define view predicates with an argument whose value is a version number. The version number must be bound on all calls to persistently tabled goals that contan them. Normally a subgoal of a persistently tabled predicate with a given version number will depend on other subgoals with the same version. This allows the programmer to keep earlier versions of tables for view systems, in order to back out changes or to keep a history of uses of the view system. So normally a new set of base tables will get a new version number, and then all subgoals depending of those base tables will have that same version number.

The `pt_add_table/3` predicate will add base tables and give them a new version number, returning that new version number. This allows the programmer to use that version number in subsequent calls to `pt_fill` to fill the tables with the correct version. Also, when calling the predicate `pt_eval_viewsys/5` the `Time` variable can be used in the subgoals in the `FillList` to invoke the correctly versioned subgoals.

A particularly interestng use of versions is in the implementation of incremental view systems. Recall that in an incremental view system, one has a table that contains the accumulated records named, say, old_records/5, and receives a base table of new records to process named, say, new_records/5. The incremental view system will define an updated record file named, say, all_records/5, which will contain the updated records after processing and includng the new_records. It is natural to use versions here, and make each predicate old_record/5, new_record/5, and old_record/5 have a version argument, say the first argument. Then note that we can define old_records in terms of the previous version of all_records, as follows:

```
old_records(Time,....) :-
     Time > 1,
     PrevTime is Time - 1,
     all_records(PrevTime,...).
```

Note that the version numbers, being always bound on call (and treated according to a `+` mode), will not appear in any stored table. The numbers will appear only in the called subgoals that are stored in the `table_instance/8` predicate in the `PT_Directory.P` file. So using version numbers does not make the persistent tables any larger.

Usage and interface (persistent_tables)

- **Exports:**
  - *Predicates:*

    pt_abolish_subgoals/1, pt_add_table/2, pt_add_table/3, pt_add_tables/2, pt_add_tables/3, pt_call/1, pt_delete_earlier/2, pt_delete_later/2, pt_eval_viewsys/5, pt_fill/1, pt_fill/2, pt_generate_table/10, pt_move_tables/1, pt_need/1, pt_remove_unused_tables/1, pt_reset/1, pt_spawn_call/1, table_persistent/5.
- **Other modules used:**
  - *Application modules:*

    basics, consult, error_handler, gensym, machine, proc_files, pt_grouper, pt_utilities, setof, shell, standard, string, xsb_configuration.

Documentation on exports (persistent_tables)

table_persistent/5: [PREDICATE]

This predicate (used as a directive) declares a predicate to be persistently tabled. The form is table_persistent(+Goal, +Modes, +TableInfo, +ProcessSpec, +DemandGoal), where:

- Goal : is the goal whose instances are to be persistently tabled. Its arguments must be distinct variables. Goal must be defined by the single clause:

  ```
  Goal :- pt_fill(Goal).
  ```

  Clauses to define the tuples of Goal must be associated with another predicate (of the same arity), whose name is obtained from Goal's predicate name by appending _ptdef.
- ModeList : a list of mode-lists (or a single mode-list.) A mode-list is a list of constants, +, t, -, and -+ with a length equal to the arity of Goal. The mode indicates puts constraints on the state of corresponding argument in a subgoal call. A "-" mode indicates that the corresponding position of the goal is to be abstracted for the persistent table; a "+" mode indicates that the corresponding position is not abstracted and a separate persistent table will be kept for each call bound to any specific constant in this argument position; a "t" mode indicates that this argument will have a "timestamp". I.e., it will be bound to an integer obtained from the persistent tabling system that indicates the snapshot of this table to use. (See add_new_table/2 for details on using timestamps.) A mode of "-+" is similar to a "-" mode in that the associated argument is abstracted. The difference is that instead of all the answers being stored in a single table, there are multiple tables, one for each value of this argument for which there are answers.

  There may be multiple such mode-lists and the first one that a particular call of Goal matches will be used to determine the table to be generated and persistently stored. A call does *not* match a mode-list if the call has a variable in a position

that is a `"+"` in that mode-list. If a call does not match any mode-list, an error is thrown. If any mode list contains a `t` mode, all must contain one in the same position.

- `TableInfo` is a term that describes the type and format of the persistent tables for this predicate. It may have the following forms, with the described meanings:
  - `file(canonical)` : indicates that the persistent table will be stored in a file as lists of field values in XSB canonical form.
  - `file(delimited(OPTS))` : indicates that the persistent table will be stored in a file as delimited fields, where OPTS is a list of options specifying the separator (and other properties) as described as options for the predicate `read_dsv/3` in the XSB lib module `proc_files`.
- `ProcessSpec` is a term that describes how the table is to be computed. It can be one of the following forms:
  - `xsb` : indicating that the persistent table will be filled by calling the goal in the current xsb process.
  - `spawn_xsb` : indicating that the persistent table will be filled by spawning an xsb process to evaluate the goal and fill the table.
- `DemandGoal` : a goal that will be called just before the main persistently tabled goal is called to compute and fill a persistent table. The main use of this goal is to invoke `pt_need/1` commands (see below) to indicate to the system that the persistent tables that this goal depends on are indeed needed. This allows tables that will be needed by this computation to be computed by other processes. This is the way that parallel computation of a complex query is supported.

`pt_call/1:` [PREDICATE]

`pt_call(+Goal)` assumes that `Goal` is persistently tabled and calls it. This predicate is normally used only in the definition of the `_ptdef` version of the persistently tabled predicate, as described above.

If the table for `Goal` exists, it reads the table file and returns its answers. If the table file is being generated, it waits until it is generated and then reads and returns its answers. If the table file doesn't exist and is not in the process of being generated, it generates the table and then returns its results. If the persistent table process declaration indicates `spawn_xsb`, it spawns a process to generate the table and reads and returns those answers when the process is completed. If the process indication is `xsb`, it calls the goal and fills the table if necessary, and returns the answers.

`pt_fill/1:` [PREDICATE]

The predicate `pt_fill(+GoalList)` checks if the persistent table for each persistently tabled `Goal` in `GoalList` exists and creates it if not. It should always succeed (once, unless it throws an error) and the table will then exist. If the desired table is already generated, it immediately succeeds. If the desired table is being generated, it looks to see if there is another table that is marked as `needs_generating` and, if so, invokes the `pt_fill/1` operation for that table. It continues this until it finds that `Goal` is

marked as `generated`, at which time it returns successfully. If no table for `Goal` exists or is being generated, it generates it.

`pt_fill/2:` [PREDICATE]

`pt_fill(+Goal,+NumProcs)` is similar to `pt_fill/1` except that it starts `NumProcs` processes to ensure that the table for `Goal` is generated. Note that filling the table for `Goal` may require filling many other tables. And those table may become marked as `needs_generation`, in which case multiple processes can work concurrently to fill the required tables.

`pt_need/1:` [PREDICATE]

`pt_need(+Goals)` creates table entries in the `PT_Directory.P` file for each persistently tabled `Goal` in the list of goals `Goals`. (`Goals` alternatively may be a single persistently tabled goal.} The new entry is given status `needs_generation`. This predicate is intended to be used in a goal that appears as the 5th argument of a `table_persistent/5` declaration. It is used to indicate other goals that are required for the computation of the goal in the first argument of its `table_persistent/5` declaration. By marking them as "needed", other processes (started by a call to `pt_fill/2`) can begin computing them concurrently. Note that these `Goals` can share variables with the main `Goal` of the declaration, and thus appropriate instances of the subgoals can be generated. For example, if time stamps are used, the needed subgoals should have the same variable as the main goal in the corresponding "time" positions.

Note that a call to `pt_need/1` should appear **only** in the final argument of a `table_persistent/5` declaration. Its correct execution requires a lock to be held and predicates to be loaded, which are ensured when that goal is called, but cannot be correctly ensured by any other call(s) to the `persistent_tables` subsystem.

`pt_eval_viewsys/5:` [PREDICATE]

The predicate `pt_eval_viewsys(+GoalList, +FileList, -Time, +FillList, +NProcs)` adds user files containing base tables to a persistent tabling system and invokes the computing and filling of dependent tables. `GoalList` is a list of subgoals that correspond to the base tables of the view system. `FileList` is the corresponding list of files that contain the data for the base tables. They must be formated as the `table_persistent` declarations of their corresponding subgoals specify. `Time` is a variable that will be set to the timestamp, if the base goals of `GoalList` contain time stamp arguments. `FillList` is a list of persistently tabled subgoals to be filled (using `pt_fill/1/2`.) `NProcs` is an integer indicating the maximum number of processes to use to evaluate the view system. This predicate provides a simple interface to `pt_add_tables/3` and `pt_fill/2`.

`pt_move_tables/1:` [PREDICATE]

`pt_move_tables(+MoveList)` moves persistent tables. `MoveList` is a list of pairs of goals of the form `FromGoal > ToGoal`, where `FromGoal` and `ToGoal` are persistently

tabled goals and their persistent tables have been filled. For each such pair the table file for `ToGoal` is set to the file containing the table for `FromGoal`. The table files must be of the same format. `FromGoal` has its table_instance fact removed. This predicate may be useful for updating new and old tables when implementing incremental view systems.

`pt_reset/1:` [PREDICATE]

`pt_reset(+Module)` processes the `PT_Directory.P` file and deletes all table_instance records for tables that have status `being_generated`. This will cause them to be re-evaluated when necessary. This is appropriate to call if all processes computing these tables have been aborted and were not able to update the directory. It may also be useful if for some reason all processes are waiting for something to be done and no progress is being made.

`pt_remove_unused_tables/1:` [PREDICATE]

This predicate cleans up unused files from the directory that stores persistent tables. `pt_remove_unused_tables(+Module)` looks through the `PT_Directory.P` file for the indicated module and removes all files with names of the form (table_<Tid>.P)(or .txt) for which there is no table id of `<Tid>`. So a user may delete (or abolish) a persistent table by simply editing the `PT_Directory.P` file (when no one is using it!) and deleting its table_instance fact. Then periodically running this predicate will clean up the storage for unnecessary tables.

`pt_abolish_subgoals/1:` [PREDICATE]

This predicate `pt_abolish_subgoals(+GoalList)` abolishes the persistent tables for all goals in `GoalList` by removing the corresponding facts in table_instance. The table files containing the data remain, and can be cleaned up using `pt_remove_unused_tables/1`.

`pt_add_table/2:` [PREDICATE]

`pt_add_table(+Goal,+FileName)` uses the file `FileName` to create a persistent table for `Goal`. `Goal` must be persistently tabled. It creates a new table_instance record in the `PT_Directory.P` file and points it to the given file. The file is not checked for having a format consistent with that declared for the persistently tabled predicate, i.e., that it is correctly formated to represent the desired tuples. The user is responsible for ensuring this.

`pt_add_table/3:` [PREDICATE]

`pt_add_table(+Goal,+FileName,?TimeStamp)` uses the file `FileName` to create a persistent table for `Goal`, which must be persistently tabled. It returns in `TimeStamp` a new (the next) time stamp for this module (obtained from the fact for predicate `table_instance_cnt/2`} in the ET Directory.) It is assumed that `Goal` has a time argument and the returned value will be used in its eventual call.

This predicate creates a new table_instance record in the `PT_Directory.P` file and sets its defining file to be the value of `FileName`. The file is not checked for consistency, that it is correctly formated to represent the desired tuples. The user is responsible for insuring this.

`pt_add_tables/2:` [PREDICATE]

`pt_add_tables(+GoalList,+FileList)` is similar to `pt_add_table/2` but takes a list of goals and a corresponding list of files, and defines the tables of the goals using the files.

`pt_add_tables/3:` [PREDICATE]

`pt_add_tables(+GoalList,+FileList,-Time)` is similar to `pt_add_table/3` but takes a list of goals and a corresponding list of files, and defines the tables of the goals using the files, returning the snapshot time in `Time`.

`pt_spawn_call/1:` [PREDICATE]

This is an internal predicate used by the `persistent_tables` system. It needs to be exported because it is used across process boundaries.

`pt_delete_later/2:` [PREDICATE]

`pt_delete_later(Module,TimeStamp)` delete all tables that have a timestamp larger than `Timestamp`. It keeps the tables of the TimeStamp snapshot. It deletes the corresponding table records from the PT_Directory, and removes the corresponding files that store the tuples.

`pt_delete_earlier/2:` [PREDICATE]

`pt_delete_earlier(Module,TimeStamp)` delete all tables that have a timestamp smaller than `Timestamp`. It keeps the tables of the TimeStamp snapshot. It deletes the corresponding table records from the PT_Directory, and removes the corresponding files that store the tuples.

`pt_generate_table/10:` [PREDICATE]

This is an internal predicate used by the `persistent_tables` system. It needs to be exported because it is used across process boundaries.

Predicate Definition Index

P

T

Operator Definition Index

(Index is empty)

Concept Definition Index

(Index is empty)

Global Index

This is a global index containing pointers to places where concepts, predicates, modes, properties, types, applications, etc., are referred to in the text of the document. Note that due to limitations of the `info` format unfortunately only the first reference will appear in online versions of the document.