Mnemosyne (MNEMOSYNE)

version 1.2

Typeset in LaTeX from SGML source using the DOCBUILDER 3.4 Document System.

Contents

Chapter 1

Mnemosyne User's Guide

1.1 Introduction

Mnemosyne is a query language of Mnesia and provides a simple syntax for complex queries.

> Note:
> *QLC* (Query List Comprehensions) is another solution for queries to Mnesia, Ets and Dets tables which will be the recommended way to perform queries. QLC belongs to Stdlib and is described there.
>
> It is not recommended to use Mnemosyne queries in performance critical applications.

`Mnesia` is a distributed DataBase Management System, utilized for telecommunications applications and other Erlang applications which require continuous operation and soft real-time properties. Mnesia is part of the Open Telecom Platform (OTP), which is a control system platform for building telecommunications applications.

1.1.1 Scope and Purpose

This manual is included as a part of the OTP document set. It describes how to use Mnemosyne queries. Programming constructs are described, and numerous programming examples are included to illustrate the use of Mnemosyne.

1.1.2 Pre-requisites

It is assumed that the reader is familiar with system development principles and database management systems. Readers are also assumed to be familiar with the Erlang programming language in general, and the `Mnesia` application in particular.

1.2 Database Queries

This chapter describes Mnemosyne, the Mnesia database query language, and the syntax, semantics, and rules which apply to Mnesia queries. The following sections are included:

- Mnemosyne - the Mnesia query language
- Evaluating queries
- Mnesia query examples
- Matching
- Generated functions

The following notational conventions are used in this chapter:

- Reserved words and symbols are written like this: `table`.
- Syntactic categories are written like this: <`pattern`>.

1.2.1 Mnemosyne - the Mnesia Query Language

Mnemosyne is the query language and the optimizing query compiler for the Mnesia Database Management System.

General Information about Queries

Database *queries* are used when more complex operations than just a simple key-value lookup are required on a database. A query can find all records in a table that fulfills a given property. For example, think of a table storing the status of subscriber lines in a telephone exchange. A query in such a table can take the format: "Which subscriber *lines* are 'blocked'?".

A query can also find records on the basis of their relationship to other records in the same table, or in other tables. If the table, which stores subscriber lines, is accompanied by a table, which pairs subscriber numbers with a subscriber line identification, we can modify our previous query and ask: "Which subscriber *numbers* are 'blocked' ?". This can be answered by constructing a query, which finds the blocked subscriber line identifications in the subscriber line table, and then finds the associated subscriber number in the table, which pairs subscriber number and subscriber line identifications.

However, the proposed solution may not be the most efficient solution, because it depends on what the tables look like in runtime. In other words, how many records the table contains, and the number of different values stored.

In a situation where there are only a couple of subscriber numbers but a million blocked lines, it would be far more efficient to first find the subscribers and then check if their line is blocked. The query evaluation order depends on how large the tables are compared to each other. The evaluation order also depends on key and other value distribution, and if there are any indices defined (refer to *Mnesia Chapter 5: Indexing* for more information).

The query compiler resolves the evaluation order. We need only express *what* we want to do, and the query compiler and query evaluator will determine the best evaluation order. Therefore, we can express the query in the most readable form.

Queries in Mnesia

Queries in Mnemosyne use first order predicate logic (similar to Prolog), but in an syntax suitable for Erlang. The "query list comprehension" used in Mnemosyne is taken from the functional languages community. The advantage over embedded SQL, is that the constructs integrate smoothly with the Erlang language.

To illustrate the Mnemosyne query language, we will show the Erlang code for the subscriber line and subscriber number tables discussed above. We define two tables `subscriber` and `line`. Their corresponding record declarations in the file `subscriber.hrl` are:

```
-record(subscriber, {snb,
                     cost_limit,
                     li}).

-record(line, {li,
               state}).
```

The query "which subscriber numbers are blocked?" can also be expressed as "which subscribers have lines which are in state 'blocked"'. This query can be coded as follows:

```
query
    [ S.snb ||                  % collect the subscriber number
        S <- table(subscriber), % where S is taken from the subscriber table
        L <- table(line),       % and L is taken from the line table
        L.state = blocked,      % and the state of that line is blocked
        L.li = S.li             % and L and S uses the same li
    ]
end
```

In the example above, the aim is to get an answer from a logical relation. Consider also the following example:

```
query [E.name || E <- table(employee),
                 E.sex = female]
end
```

This means "the Erlang list of all female employees". A formulation closer to the list comprehension is: "the Erlang list of all names of E such that E is in the table `employee` and E's sex is female".

Some words have a direct correspondence to the elements in the list comprehension notation:

the Erlang list of all	`[ ]`
such that	`||`
is in	`<-`
and	`,`

Table 1.1: Natural Language Translation

Another query component is *rules*, which can be compared to *views* in Relational Databases, where the purpose is to define a "virtual table". This "table" looks like an ordinary table to the user, which means

that queries can be formulated on stored data (as well as views). In the subscriber line example, a rule can give the subscriber number and the corresponding line status from both tables, and there is no need to create a third table.

The rule is a definition of how to calculate the records on demand from a query. In Erlang modules, rules are written with the same syntax as the bodies in the query list comprehensions. They are exported and can be used by other modules.

Our `subscriber` example formulated as a rule would look as follows:

```
blocked_subscribers(S, subscriber) :-
        S <- table(subscriber),
        L <- table(line),
        L.state = blocked,
        L.li = S.li.
```

This rule can be used in a query in the same manner as a table but with the keyword `rule` substituted for `table`.

```
query [ S.snb || S <- rule(blocked_subscribers) ] end
```

Query Syntax

Database queries can be included in an Erlang program, but there must be a directive in the Erlang file which informs the compiler about its behavior towards queries. This directive is:

```
        -include_lib("mnemosyne/include/mnemosyne.hrl").
```

The high-level syntax of the query list comprehension is:

```
query [ <pattern> || <body> ] end
```

The <body> is a comma-separated sequence of:

1. <logical-variable> <- table(<table-name> [, <table-type>])
2. <logical-variable> <- rule(<rule-name>)
3. <logical-variable> <- rule(<rule-name>)
4. <logical-variable> <- rule(<module> : <rule-name>])
5. <logical-variable> <- <erlang-list-expression>
6. <expression> <relop> <expression>
7. <erlang-test-expression>

The <relop> operators are:

- = for unification
- /= for not unification
- < for less than
- > for greater than
- =< for equal to or less than
- >= for equal to or greater than.

A <logical-variable> is written exactly as an Erlang variable. The <table-name>, <table-type>, <rule-name> and <module> are atoms. The <table-name> and <table-type> can also be an Erlang variable. The logical variables are local to a list comprehension and shadows any Erlang variable with the same name.

The <pattern> is an Erlang term without function calls. It may contain (bound) Erlang variables and it usually has one or more <logical-variable>, since these are used to get data out from the query body and into the produced list.

An <expression> is any Erlang expression which may include function calls and <logical-variable>. The variant <erlang-list-expression> is an <expression> which must produce a list where all elements are records of the same type.

The <erlang-test-expression> is an <expression> which has the values `true` or `false`.

Erlang variables are allowed in all variations of <expression> and in <pattern>. They must be bound in the query list comprehension.

Query Semantics

The constructs used in the Mnemosyne query language have the following meanings:

- *Comma*. The comma, used to separate different body elements, is equivalent to "and". Thus, the body can be viewed as a collection of tests and statements which should be true for each solution which is produced when evaluating the query list comprehension. Refer to subscriber query [page 3] as an example of this.
- <logical-variable> <- .... This expression means that the variable is taken from the values in the expression to the right of the arrow. For example, `E <- [#e{a=1}, #e{a=2}]` says that `E` takes the values `#e{a=1}`, or `#e{a=2}`
- <-. These constructs usually generate values. However, if the logical variable is bound it tests that value. If a test fails it means that the query tries another alternative. For example:

  ```
  query [ X.a || X <- [#e{a=1}, #e{a=2}],
                 X <- [#e{a=3}],
                 .... ]
  end
  ```

  The body means that the field 'a' of `X` should be 3 and at the same time either 1 or 2. So the list of solutions will always be empty.

The test <expression> <relop> <expression> and the `true`-or-`false` returning test <erlang-test-expression> simply filters out the solutions. The purpose of the latter test is to provide user defined data tests.

We will next consider the logical variables *associated records* in an expression like `x <- table(a)`. We have already established the following rules and assumptions:

1. the values stored in tables are records
2. all records in a table must be of the same type
3. by default, the record definition has the same name as the table itself
4. The <logical-variable> must have the same record association as the records produced by the right side of the <- constructs.

In the example `X <- table(a)`, the associated record of *x* is *a* because table *a* stores records with the name *a*. Since release 3.4 of Mnesia it has been possible to separate record name and its table type. If the type of the table is different from its name, this can be specified in Mnemosyne using `X <- table(Name, Type)` where Name is the Name of the table and Type is the record name.

Similar to tables, rules produce or test records. The return type for a rule is by default the name of the rule. Rules can be declared to return other types. This makes it possible to construct a rule for some special cases with a name like `blocked_subscriber` which still produces or tests subscriber records.

In Erlang we must always tell the compiler which record definition it should use by putting the record name after a hash mark. In general, this is not needed in Mnemosyne since, in most cases, the query compiler can deduce the associated record. That is the reason `S.li` is acceptable instead of the full `S#subscriber.li`. It will not cause an error if the longer version was written, but if we do write the record name it must be the same record name as the one the query compiler deduces. Sometimes the compiler is unable to find the associated record. When this happens, an error message is issued. It is also preferred to write out the type of the associated record for performance reasons. If the associated record is part of a complex constraint, the constraint may be compiled to a function if the type of the associated record is known (explicitly or deducable) at Erlang compile time.

Note:
A function used in a query list comprehension must *never* directly or indirectly:

1. have side effects;
2. access the database neither by a query nor by Mnesia functions;
3. spawn processes, or;
4. send or receive messages.

Rules

A rule is composed of *clauses* and each clause has the structure:

```
<head> :- <body>
```

- The clauses are separated by semicolon, and the rule is terminated by a dot.
- The `<head>` looks like an Erlang function with one or two arguments, where the first argument is a variable and the second, optional, argument an atom. If there is a second argument, it must be present in all clauses and have the same value.
- The `<body>` has the same syntax as the `<body>`. in query list comprehensions [page 4]
- The argument variable of a rule clause has an associated record.
- The default associated record is the name of the rule. This can be changed by declaring the associated record type in the head of the clause:

  ```
  <rule-name> (<return-var>, <record-name>)
  ```

  The syntax used in previous mnemosyne versions, by declaring the the associated recordtype with an `argtype` declaration still works but is depreciated.

> **Note:**
> The <logical-variable> mentioned in the <head> must also occur in the <body>.

Review the rule example [page 4].

```
blocked_subscribers(S, subscriber) :-
        S <- table(subscriber),
        L <- table(line),
        L.state = blocked,
        L.li = S.li.
```

It produces a list of `subscriber` records. Rules with a single argument return records of the same type as the name of the rule. For example, the following rule produces records of type `blocked`

```
-record (blocked, {snb, li}).
blocked (X) :-
    S <- table (subscriber),
    L <- table(line),
    L.state = blocked,
    L.li = S.li,
    X = #blocked{snb=S#subscriber.snb, li=S#subscriber.li}.
```

1.2.2 Evaluating Queries

The previous sections described how to define queries. This section describes how to evaluate queries.

The principle is simple: query list comprehensions, compile and optimize the query and return a *handle*. This handle is then passed on for execution:

```
    Handle =
        query
           [ S.snb || S <- table(subscriber),
                      S.li = none]
        end,

    AllAnswers =
        mnesia:transaction(
             fun() ->
                      mnemosyne:eval(Handle)
             end)
```

There are three ways of evaluating queries. The `mnemosyne:eval/1` is the simplest of the three. It takes a handle and returns all solutions. Sometimes we only need to view a few solutions, examine them and possibly get more. Think of an airline routing database: you do not want to know all possible connections between two cities, but usually enough information is given after observing one or two.

Use the *cursor* with a query evaluation to produce a few solutions only. With a handle we create a cursor by calling `mnemosyne:cursor/1`. With the cursor we can repeatedly call `mnemosyne:next_answers` to get more solutions. When an empty list is returned there are no more possible solutions. Delete the cursor with `mnemosyne:delete_cursor/1`.

```
Handle =
    query
       [ S.snb || S <- table(subscriber),
                  S.li = none]
    end,

AFewAnswers =
    mnesia:transaction(
         fun() ->
                 Cursor = mnemosyne:cursor(Handle),
                 % now get at least two but not
                 % more than five solutions:
                 L = mnemosyne:next_answers(Cursor,2,5),
                 mnemosyne:delete_cursor(Cursor),
                 L
         end)
```

A query evaluation can be time consuming, but can be broken up by using the cursor with `setup_query/1` and `init_query/1`:

```
Handle =
    query
       [ S.snb || S <- table(subscriber),
                  S.li = none]
    end,

QuerySetup = mnemosyne:setup_query(Handle),

AFewAnswers =
    mnesia:transaction(
         fun() ->
                Cursor = mnemosyne:init_query(QuerySetup),
                mnemosyne:next_answers(Cursor, 5, 5)
         end),

  % Here we may call more init_query-next_answers constructions
  % with the same Handle. Note that the query is evaluated from
  % "scratch" because of the call to mnemosyne:init_query/1.

  mnemosyne:delete_query(QuerySetup)
```

Because of table updates, a query which is compiled and optimized may be incorrect when the handle returns. This can be rectified with the function `mnemosyne:reoptimize/1` which takes a handle, re-optimizes the query and returns a new handle.

1.2.3 Query Examples

This section describes an example which illustrates the use of Mnemosyne. The example given is of a simplified local exchange, with AXE-10 exchange as a model. The purpose of this section is to show different constructs in a telecom environment. It should not be taken as a proposed data model for a modern telecom system.

Our telephone example includes the following components, relationships, and events:

- The exchange has a number of *subscribers*.
- Each subscriber has a *subscriber number*, which is abbreviated *snb*.
- Each physical line enters the exchange through a *line interface card*. Lines are abbreviated *li*.
- The *li* has an associated *status* which indicates if the line is blocked, or available.
- One single table stores the accumulated cost for each subscriber.

Program Definitions

We identify three tables:

- `subscriber` with subscriber numbers `snb`, line interface number `li`, and a maximum cost `cost_limit` which must not be exceeded.
- `line` with line interface number `li`, and its `state`.
- `account`, a table which stores the cost of calls. It has an `snb` field, and the accumulated cost in `cost`.

The corresponding record definitions are stored in a file named `subscriber.hrl`, which has the following record definitions:

```
-record(subscriber, {snb,
                     cost_limit,
                     li}).

-record(line, {li,
               state}).

-record(account, {snb,
                  cost}).
```

The program file is titled `subscriber.erl`. It declares the module name `subscriber`, calls the record definition in `subscriber.hrl`, and Mnesia query support `mnemosyne.hrl`.

```
-module(subscriber).
-compile(export_all).

-include("subscriber.hrl").
-include_lib("mnemosyne/include/mnemosyne.hrl").
```

We then create the required tables and load data by entering table definitions into a file named `subscriber.tables`, which has the following content:

```
{tables,
 [{subscriber, [{attributes, [snb,cost_limit,li]}]},
  {line, [{attributes, [li, state]}]},
  {account, [{attributes, [snb, cost]}]}
  ]
}.

%% Subscribers
{subscriber, 1230,   0,   none}.
```

```
{subscriber, 1231,   0,   none}.
{subscriber, 1232,   0,   none}.
{subscriber, 1233,   0,   none}.
{subscriber, 1234, 100, {li,1}}.
{subscriber, 1235, 200, {li,3}}.
{subscriber, 1236, 150, {li,2}}.
{subscriber, 1237,   0,   none}.
{subscriber, 1238,   0,   none}.
{subscriber, 1239,   0,   none}.

%% Lines
{line, {li,0}, blocked}.
{line, {li,1},  normal}.
{line, {li,2},  normal}.
{line, {li,3}, blocked}.
{line, {li,4}, blocked}.
{line, {li,5}, blocked}.
{line, {li,6}, blocked}.
{line, {li,7}, blocked}.



%% Accounts
{account, 1234, 0}.
{account, 1235, 0}.
{account, 1236, 0}.
{account, 1237, 0}.
```

Program Output

In our program, this file is called with the statement:

```
mnesia:load_textfile("subscriber.tables")
```

To retrieve a list of all free subscriber numbers we call the following function in a transaction:

```
free_subscriber_numbers() ->
    mnemosyne:eval(
      query [ S.snb || S <- table(subscriber),
                       S.li = none]
      end
     ).
```

The rule `too_high_cost/0` locates and returns all subscribers with an accumulated cost that exceeds their limit:

```
limit_exceeded(S, subscriber) :-
    S <- table(subscriber),
    A <- table(account),
    A.snb = S.snb,
    A.cost > S.cost_limit.
```

We could find all subscriber numbers of subscribers who have exceeded their cost limit as follows:

```
Q = query
[ S.snb || S <- rule(limit_exceeded) ]
end
```

1.2.4 Matching

Mnesia provides the programmer with a method of matching objects against a pattern. This is the Mnesia matching function:

`mnesia:match_object(Pattern) ->transaction abort | ObjList.`

This function matches `Pattern` for objects. A `Pattern` is a tuple with the name (identity) of the table as the first element. The table collates all data retrieved.

In comparison to a list comprehension query, `mnesia:match_object` is a low level function. The following two functions both return the same objects; however, the second example uses matching.

```
f1() ->
    Q = query
         [E || E <- table(employee),
          E.sex = female]
    end,
    F = fun() -> mnemosyne:eval(Q) end,
    mnesia:transaction(F).
```

and

```
f2() ->
    WildPat = mnesia:table_info(employee, wild_pattern),
    Pat = WildPat#employee{sex = female},
    F = fun() -> mnesia:match_object(Pat) end,
    mnesia:transaction(F).
```

The pattern supplied to the `mnesia:match_object/1` function must be a valid record, and the first element of the provided tuple must be a valid table name. The special element `'_'` matches all the records.

There are advantages in using the Mnemosyne query syntax instead of the `mnesia:match_object/1` function:

- The pattern is computed in compile time by the Mnemosyne compiler instead of doing it in run time in the `f2/0` function.
- Mnemosyne provides more sophisticated evaluation optimizations based on indices and on statistics from and about the table.
  Whereas, the optimizations that `mnesia:match_object/1` function provides are limited in both scope and number. The `mnesia:match_object` function is also performed during run time, which in turn reduces performance.
- The Mnemosyne query syntax is quite compact and makes it easier to express complex queries.

It is also possible to use the match function if we want to check the equality of different attributes. Assume we have the following record definition:

```
-record(foo, {a, b, c}).
```

The pattern {foo, '$1', '$1', '_'} then extracts all objects of type foo where the first two attributes have the same value.

If the key attribute is bound in a pattern, the match operation is very efficient. The pattern {foo, 123, '_', elvis} can be used to extract all objects with key 123, and the last attribute set to the atom elvis. This is the same as extracting all the elvis objects from the result of mnesia:read({foo, 123}), but more efficient.

If the key attribute in a pattern is given as '_', or '$1', the whole foo table must be searched for objects that match. If the table is large, this may be a time consuming operation. This can be remedied with indices (refer to *Mnesia Chapter 5: Indexing* for more information).

This chapter closes with an example of information extraction from a Company database:

- all employees who have a salary higher than X.
- all employees who work in the Dep department.

The first example demonstrates the query execution with list comprehension notation. The second example illustrates a query coded with a matching function.

The list comprehension based implementation looks as follows:

```
get_emps(Salary, Dep) ->
    Q = query
          [E || E <- table(employee),
                At <- table(at_dep),
                E.salary > Salary,
                E.emp_no = At.emp,
                At.dept_id = Dep]
        end,
    F = fun() -> mnemosyne:eval(Q) end,
    mnesia:transaction(F).
```

The data model for the Company database introduced in the Mnesia documentation was designed to facilitate the posting of queries like the one shown above.

To implement the same query by directly searching the database is more complex. The following function does precisely this:

```
get_emps2(Salary, Dep) ->
    Epat = mnesia:table_info(employee, wild_pattern),
    Apat = mnesia:table_info(at_dep, wild_pattern),
    F = fun() ->
                All = mnesia:match_object(Epat),
                High = filter(All, Salary),
                Alldeps = mnesia:match_object(Apat),
                filter_deps(High, Alldeps, Dep)
        end,
    mnesia:transaction(F).


filter([E|Tail], Salary) ->
    if
```

```
        E#employee.salary > Salary ->
            [E | filter(Tail, Salary)];
        true ->
            filter(Tail, Salary)
    end;
filter([], _) ->
    [].

filter_deps([E|Tail], Deps, Dep) ->
    case search_deps(E#employee.name, Deps, Dep) of
        true ->
            [E | filter_deps(Tail, Deps, Dep)];
        false ->
            filter_deps(Tail, Deps, Dep)
    end;
filter_deps([], _,_) ->
    [].


search_deps(Name, [D|Tail], Dep) ->
    if
        D#at_dep.emp == Name,
        D#at_dep.dept_id == Dep -> true;
        true -> search_deps(Name, Tail, Dep)
    end;
search_deps(Name, Tail, Dep) ->
    false.
```

The function `mnesia:match_object/1` will automatically make use of indices if any exist. However, no heuristics are performed in order to select the best index, if more than one exists.

As can be seen, the list comprehension provides a more elegant solution.

1.2.5 Matching in Record Fields

There is a difference when matching record fields in a mnemosyne list comprehension and in Erlang in general (for example, a function clause header). The following code returns true for all `employee` where `emp_id` is 312 or 400:

```
test_employee(#employee{emp_id = 312}) -> true;
test_employee(#employee{emp_id = 400}) -> true;
test_employee(_) -> false.
```

That is, it does not check other fields of the employee record. Compare that with the following mnemosyne query:

```
query [E || E <- table(employee),
            E <- [#employee{emp_id=312},
                  #employee{emp_id=400}]]
```

The query will return all employees from the employee table whos `emp_id` is either 312 or 400 *and have the other fields set to the default values for an employee*. To select all items that have a field set to some values (disregarding the other fields) the constraint can be put in separate function. For example, select all employees whos `emp_id` is either 312 or 400 independently of other fields:

```
query [E || E <- table(employee),
            test_employee(E)]

test_employee(#employee{emp_id = 312}) -> true;
test_employee(#employee{emp_id = 400}) -> true;
test_employee(_) -> false.
```

If there is only one acceptable value for a record field it is more efficient to write it directly in the query. Select employees whos `emp_id` is 312:

```
query [E || E <- table(employee),
            E#employee.emp_id = 312]
```

Mnemosyne Reference Manual

Short Summaries

- Erlang Module **Mnemosyne** [page 16] – A query language support for the DBMS Mnesia

Mnemosyne

The following functions are exported:

- `all_answers(Cursor) -> Answer`
  [page 18] Return all answers from `Cursor` in a list
- `cursor(Handle) -> Cursor`
  [page 18] Create a cursor in preparation for fetching answers
- `cursor(Handle,Nprefetch) -> Cursor`
  [page 18] Create a cursor in preparation for fetching answers
- `delete_cursor(Cursor)`
  [page 18] Stop the query associated with Cursor
- `delete_query(QuerySetup)`
  [page 18] Delete a query setup.
- `eval(Handle) -> Answers`
  [page 18] Get all answers from the `Handle`
- `init_query(QuerySetup) -> Cursor`
  [page 18] Quick query initialization
- `init_query(QuerySetup,Nprefetch) -> Cursor`
  [page 18] Quick query initialization
- `next_answers(Cursor) -> Answers`
  [page 18] Fetche the next answers
- `next_answers(Cursor,Nmin,Nmax) -> Answers`
  [page 18] Fetche the next answers
- `reoptimize(Handle) -> Handle`
  [page 18] Optimize a query
- `setup_query(Handle) -> QuerySetup`
  [page 19] Create a query setup
- `version() -> String`
  [page 19] Mnemosyne module version

mnemosyne

Erlang Module

Queries are used for accessing the data in a Database Management System. The query specifies a relation (possibly complicated) to all of the selected data. This could involve several tables as well as conditions such as '<' (less than), function calls and similar.

Mnesia has two query interfaces which are used together:

- Mnemosyne, which is this module
- *QLC (Query List Comprehensions)*, an Erlang language construct for the queries. This will be the recommended way to perform queries

The exact syntax of query list comprehensions are described in a separate section [page 19] of this document.

The query list comprehensions only define the query and the syntax of the solutions to be returned. The actual evaluation is determined by calling different functions with a handle obtained by the list comprehension. For example:

```
-record(person, {name,age}).
Handle =
     query
         [ P.name || P <- table(person) ]
     end,
L = mnesia:transaction(
     fun() ->
         mnemosyne:eval(Handle)
     end)
```

The example above matches a list of all names in the table "person" with the variable `L`. Note the following points:

- Each database table must have a corresponding record declaration.
- A *query* is declared with

  ```
  query [ <pattern> || <body> ] end
  ```

  where <`pattern`> is an Erlang term without function calls. The notation `P.name` means that `P` is a variable and it has an associated record with a field `name` which we use. The <`body`> is a sequence of conditions separated by commas. In the example, we have `P <- table(person)` which means: "`P` is taken from the table `person`".

  The whole query could therefore be read as follows: "Make the list of all names of `P` such that `P` is taken from the table `person`".

  However, the query list comprehension does not return the answers but a *handle*. This handle is used as an argument for different evaluation functions which do the actual query processing. In the example we used the simplest, `eval/1`, which evaluates the query and returns all the answers.

- Some parts of the query must be evaluated in a Mnesia transaction or by utilizing an alternative Mnesia access context. These functions are marked in the function descriptions below.

After obtaining a handle from a query list comprehension, the query can be evaluated in three different ways:

- A simple all-answer query as in the example shown above. This function is `eval/1`.
- Getting the answers in small or large chunks. The query may be aborted when enough solutions have been obtained. These are called *cursors*. The functions are `cursor/1`, `cursor/2`, `next_answers/1`, `next_answers/3`, `all_answers/1`, `all_answers/3`, and `delete_cursor/1`.
- An even more sophisticated cursor version where the time consuming part of the cursor creation can be done in advance. The functions are `setup_query/1`, `init_query/1`, `init_query/2`, `next_answers/1`, `next_answers/3`, `all_answers/1`, `all_answers/3`, and `delete_query/1`.

Let us reconsider the previous example, this time with cursors. In the following example, we will get just five names *without evaluating all of the answers*:

```
-record(person, {name,age}).
Handle =
     query
         [ P.name || P <- table(person) ]
     end,
L = mnesia:transaction(
     fun() ->
         Cursor = mnemosyne:cursor(Handle),
         As = mnemosyne:next_answers(Cursor, 5, 5),
         mnemosyne:delete_cursor(Cursor),
         As
     end)
```

The third way of evaluating a query is by a further division of the query process. The `cursor/1` function is now split into two. The reason for this is that we can set up the query when there is plenty of time and initialize it when answers are needed quickly. As in the previous example, we will get just five names:

```
-record(person, {name,age}).

Handle =
     query
         [ P.name || P <- table(person) ]
     end,

QuerySetup = mnemosyne:setup_query(Handle),
L = mnesia:transaction(
     fun() ->
         Cursor = mnemosyne:init_query(QuerySetup),
         mnemosyne:next_answers(Cursor, 5, 5)
     end),

% Here we may call more init_query-next_answers constructions
% with the same Handle
mnemosyne:delete_query(QuerySetup)
```

Exports

`all_answers(Cursor) -> Answer`

Returns all remaining answers from the query identified by `Cursor`. It can be applied after `next_answers` to obtain all answers that are left.

Note: This must be evaluated inside a transaction.

`cursor(Handle) -> Cursor`
`cursor(Handle,Nprefetch) -> Cursor`

Sets up a query for evaluation and starts an answer pre-fetch. `Nprefetch` gives the number of answers to pre-fetch and must be greater than 0. The default value is 1. A pre-fetch is the first part of a query evaluation. It is placed in a separate process which may on some occasions speed up the subsequent collection of answers.

Note: This must be evaluated inside a transaction.

`delete_cursor(Cursor)`

Deletes the Cursor and associated query evaluation.

Note: This must be evaluated inside a transaction.

`delete_query(QuerySetup)`

Deletes a query setup.

`eval(Handle) -> Answers`

Starts a query evaluation according to the `Handle` and collects all answers in one operation.

Note: This must be evaluated inside a transaction.

`init_query(QuerySetup) -> Cursor`
`init_query(QuerySetup,Nprefetch) -> Cursor`

Performs the last short step in starting a query from `QuerySetup`. `Nprefetch` defines the number of answers to pre-fetch as in `cursor/2`. The default value is 1.

Note: This must be evaluated inside a transaction.

`next_answers(Cursor) -> Answers`
`next_answers(Cursor,Nmin,Nmax) -> Answers`

Fetches the next answers from the query evaluation identified by `Cursor`. At least `Nmin` and at most `Nmax` answers are collected. If less than `Nmin` answers are returned; for example, 0, there are no more answers. If enough answers are not available, but more are expected, the functions wait for them.

Note: This must be evaluated inside a transaction.

`reoptimize(Handle) -> Handle`

Re-optimizes a query. Queries are always optimized, but the optimization takes into account the dynamic table statistics like size, attribute distribution etc. If a table has changed after obtaining the `Handle` from a query list comprehension, the query execution plan will no longer be appropriate (although semantically correct). This function will rearrange the execution plan according to the current statistics from the database.

`setup_query(Handle) -> QuerySetup`

Creates a query setup, that is, performs most of a query evaluation without actually initiating the actual evaluation.

`version() -> String`

Returns the current module version.

List Comprehension

There must be a directive in the Erlang file telling the compiler how to treat queries. This directive is:

```
-include_lib("mnemosyne/include/mnemosyne.hrl").
```

A list comprehension consists of:

```
query [ <pattern> || <body> ] end
```

The <pattern> is a description of the terms that are returned by a query. Details of how to obtain the actual values in the <pattern> is given by the <body>.

The <pattern> is an Erlang term without function calls. It typically has one or more variables from the <body> which are instantiated for each answer produced. Every element in the returned list is composed by instantiating this <pattern> and then adding it to the answers.

The <body> takes a sequence of *goals* separated by ",". The possible goals are:

- <logical-variable> <- table(<table-name> [, <table-type>])
- <logical-variable> <- rule(<rule-name>)
- <logical-variable> <- rule(<module> : <rule-name>)
- <logical-variable> <- <erlang-list-expression>
- <expression> <relop> <expression>
- <erlang-test-expression>

A <logical-variable> is written exactly as an Erlang variable. The <table-name>, <table-type>, <rule-name> and <module> are atoms. The <table-name> and <table-type> may be an Erlang variable which must be bound at runtime. The logical variables are local to a list comprehension and shadows any Erlang variables with the same name.

An <expression> is any Erlang expression which may include function calls and <logical-variable>. The variants <erlang-list-expression> is an <expression> which must produce lists where all elements are records of the same type. The <logical-variable> must have the same associated record. The

<erlang-test-expression> is an <expression> which only has the values true or false.

Erlang variables are allowed in all variants of <expression> and in <pattern>. They must always be bound in the query list comprehension.

logical variables is local to a query list comprehension and have an associated Erlang record. The associated record can in most cases be inferred by the query compiler. Therefore, the normal notation for the field f1 in variable X is just X.f1. The query compiler notifies when it cannot deduce the corresponding record. The explicit form is X#r.f1 as in ordinary Erlang. If the type of the record is not deducable at Erlang compile time, it is more efficient to use the explicit form as a help to the compiler. A variable receiving values from a table will have the record with the same name as the table.

Erlang variables are allowed in <expression> and in some places as described above. They must always be bound in the query list comprehension.

Errors in the description are reported as exceptions in the Erlang standard format as follows:

```
{error, {Line,Module,Msg}}
```

The descriptive English text is returned by calling

```
Module:format_error(Msg)
```

Note:
A function used in a query list comprehension must *never* directly or indirectly:

1. have side effects
2. access the database either by a query or by Mnesia functions
3. spawn processes
4. send or receive messages

Rules (Views)

A *rule* (or *view*) is a declaration of how to combine data from sources as a kind of "subroutine". Assume that we have the following query list comprehension:

```
query
    [ Employee || Employee <- table(employee),
                  Employee.department = sales  ]
end
```

This retrieves a list of all sales employees. This could be formulated in the following rule:

```
sales(E, employee) :-
        E <- table(employee),
        E.salary = sales.
```

The `employee` declaration in the head of the rule forces the rule argument to associate the `employee` record. If we omit the declaration, then the associated record would be the rule name, in this case `sales`. Note that the syntax used in previous versions of Mnemosyne by using an separate `argtype` declaration still works, but the above method is prefered.

The `sales` rule may now be used in a query list comprehension:

```
query
    [ SalesPerson || SalesPerson <- rule(sales) ]
end
```

The SalesPerson is an `employee` record because of the declaration of the rule above. Another example lists the names of all female sales people:

```
query
    [ SalesPerson.name || SalesPerson <- rule(sales),
                          SalesPerson.sex = female   ]
end
```

The rule must have one argument when used. Although the declaration of a rule looks similar to an ordinary function, no function of that name is constructed. Hence the name of the rule can be used for another function. All rules are automatically exported so they could be referred in other modules by the usual notation `module:name`. After the `:-`, there is the usual <`body`> as in the query list comprehension.

Generated Functions

When compiling queries some extra (hidden) functions are automatically generated and exported. Thus, there cannot be other functions with the same name and arity within the module. Three such generated functions exist. They are:

- `MNEMOSYNE QUERY/2`
- `MNEMOSYNE RECFUNDEF/1`
- `MNEMOSYNE RULE/1`

List of Tables

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.