

Mnesia

Copyright © 1997-2025 Ericsson AB. All Rights Reserved.
Mnesia 4.16.2
October 21, 2025

Copyright © 1997-2025 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

October 21, 2025

1 Mnesia User's Guide

The Mnesia application is a distributed Database Management System (DBMS), appropriate for telecommunications applications and other Erlang applications, which require continuous operation and exhibit soft real-time properties.

1.1 Introduction

The Mnesia application provides a heavy duty real-time distributed database.

1.1.1 Scope

This User's Guide describes how to build Mnesia database applications, and how to integrate and use the Mnesia database management system with OTP. Programming constructs are described, and numerous programming examples are included to illustrate the use of Mnesia.

This User's Guide is organized as follows:

- *Mnesia* provides an introduction to Mnesia.
- *Getting Started* introduces Mnesia with an example database. Examples are included how to start an Erlang session, specify a Mnesia database directory, initialize a database schema, start Mnesia, and create tables. Initial prototyping of record definitions is also discussed.
- *Build a Mnesia Database* more formally describes the steps introduced in the previous section, namely the Mnesia functions that define a database schema, start Mnesia, and create the required tables.
- *Transactions and Other Access Contexts* describes the transactions properties that make Mnesia into a fault tolerant, real-time distributed database management system. This section also describes the concept of locking to ensure consistency in tables, and "dirty operations", or short cuts, which bypass the transaction system to improve speed and reduce overheads.
- *Miscellaneous Mnesia Features* describes features that enable the construction of more complex database applications. These features include indexing, checkpoints, distribution and fault tolerance, disc-less nodes, replication manipulation, local content tables, concurrency, and object-based programming in Mnesia.
- *Mnesia System Information* describes the files contained in the Mnesia database directory, database configuration data, core and table dumps, as well as the important subject of backup, fall-back, and disaster recovery principles.
- *Combine Mnesia with SNMP* is a short section that outlines Mnesia integrated with SNMP.
- *Appendix A: Backup Callback Interface* is a program listing of the default implementation of this facility.
- *Appendix B: Activity Access Callback Interface* is a program outlining one possible implementation of this facility.
- *Appendix C: Fragmented Table Hashing Callback Interface* is a program outlining one possible implementation of this facility.

1.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language, system development principles, and database management systems.

1.2 Mnesia

The management of data in telecommunications system has many aspects, thereof some, but not all, are addressed by traditional commercial Database Management Systems (DBMSs). In particular the high level of fault tolerance that is required in many nonstop systems, combined with requirements on the DBMS to run in the same address space as the application, have led us to implement a new DBMS, called Mnesia.

Mnesia is implemented in, and tightly connected to Erlang. It provides the functionality that is necessary for the implementation of fault tolerant telecommunications systems.

Mnesia is a multiuser distributed DBMS specially made for industrial telecommunications applications written in Erlang, which is also the intended target language. Mnesia tries to address all the data management issues required for typical telecommunications systems. It has a number of features that are not normally found in traditional databases.

In telecommunications applications, there are different needs from the features provided by traditional DBMSs. The applications now implemented in Erlang need a mixture of a broad range of features, which generally are not satisfied by traditional DBMSs. Mnesia is designed with requirements like the following in mind:

- Fast real-time key/value lookup
- Complicated non-real-time queries mainly for operation and maintenance
- Distributed data because of distributed applications
- High fault tolerance
- Dynamic reconfiguration
- Complex objects

Mnesia is designed with the typical data management problems of telecommunications applications in mind. This sets Mnesia apart from most other DBMS. Hence Mnesia combines many concepts found in traditional databases such as transactions and queries with concepts found in data management systems for telecommunications applications, for example:

- Fast real-time operations
- Configurable degree of fault tolerance (by replication)
- The ability to reconfigure the system without stopping or suspending it.

Mnesia is also interesting because of its tight coupling to Erlang, thus almost turning Erlang into a database programming language. This has many benefits, the foremost is that the impedance mismatch between the data format used by the DBMS and the data format used by the programming language, which is used to manipulate the data, completely disappears.

1.2.1 Mnesia Database Management System (DBMS)

Features

Mnesia contains the following features that combine to produce a fault-tolerant, distributed DBMS written in Erlang:

- Database schema can be dynamically reconfigured at runtime.
- Tables can be declared to have properties such as location, replication, and persistence.
- Tables can be moved or replicated to several nodes to improve fault tolerance. The rest of the system can still access the tables to read, write, and delete records.
- Table locations are transparent to the programmer. Programs address table names and the system itself keeps track of table locations.
- Database transactions can be distributed, and many functions can be called within one transaction.
- Several transactions can run concurrently, and their execution is fully synchronized by the DBMS. Mnesia ensures that no two processes manipulate data simultaneously.

- Transactions can be assigned the property of being executed on all nodes in the system, or on none. Transactions can also be bypassed in favor of running "dirty operations", which reduce overheads and run fast.

Details of these features are described in the following sections.

Add-On Application

Query List Comprehension (QLC) can be used with Mnesia to produce specialized functions that enhance the operational ability of Mnesia. QLC has its own documentation as part of the OTP documentation set. The main features of QLC when used with Mnesia are as follows:

- QLC can optimize the query compiler for the Mnesia DBMS, essentially making the DBMS more efficient.
- QLC can be used as a database programming language for Mnesia. It includes a notation called "list comprehensions" and can be used to make complex database queries over a set of tables.

For information about QLC, see the *qlc* manual page in STDLIB.

When to Use Mnesia

Use Mnesia with the following types of applications:

- Applications that need to replicate data.
- Applications that perform complicated searches on data.
- Applications that need to use atomic transactions to update several records simultaneously.
- Applications that use soft real-time characteristics.

Mnesia is not as appropriate with the following types of applications:

- Programs that process plain text or binary data files.
- Applications that merely need a look-up dictionary that can be stored to disc. Those applications use the standard library module `dets`, which is a disc-based version of the module `ets`. For information about `dets`, see the *dets* manual page in STDLIB.
- Applications that need disc logging facilities. Those applications can use the module `disk_log` by preference. For information about `disk_log`, see the *disk_log* manual page in Kernel.
- Hard real-time systems.

1.3 Getting Started

This section introduces `Mnesia` with an example database. This example is referenced in the following sections, where the example is modified to illustrate various program constructs. This section illustrates the following mandatory procedures through examples:

- Starting the Erlang session.
- Specifying the `Mnesia` directory where the database is to be stored.
- Initializing a new database schema with an attribute that specifies on which node, or nodes, that database is to operate.
- Starting `Mnesia`.
- Creating and populating the database tables.

1.3.1 Starting Mnesia for the First Time

This section provides a simplified demonstration of a `Mnesia` system startup. The dialogue from the Erlang shell is as follows:

```
unix>  erl -mnesia dir '"/tmp/funky"'
Erlang (BEAM) emulator version 4.9

Eshell V4.9  (abort with ^G)
1>
1> mnesia:create_schema([node()]).
ok
2> mnesia:start().
ok
3> mnesia:create_table(funky, []).
{atomic,ok}
4> mnesia:info().
---> Processes holding locks <---
---> Processes waiting for locks <---
---> Pending (remote) transactions <---
---> Active (local) transactions <---
---> Uncertain transactions <---
---> Active tables <---
funky          : with 0 records occupying 269 words of mem
schema         : with 2 records occupying 353 words of mem
===> System info in version "1.0", debug level = none <===
opt_disc. Directory "/tmp/funky" is used.
use fall-back at restart = false
running db nodes = [nonode@nohost]
stopped db nodes = []
remote           = []
ram_copies       = [funky]
disc_copies      = [schema]
disc_only_copies = []
[{nonode@nohost,disc_copies}] = [schema]
[{nonode@nohost,ram_copies}] = [funky]
1 transactions committed, 0 aborted, 0 restarted, 1 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
ok
```

In this example, the following actions are performed:

- **Step 1:** The Erlang system is started from the UNIX prompt with a flag `-mnesia dir '"/tmp/funky"'`, which indicates in which directory to store the data.
- **Step 2:** A new empty schema is initialized on the local node by evaluating *mnesia:create_schema([node()])*. The schema contains information about the database in general. This is explained in detail later.
- **Step 3:** The DBMS is started by evaluating *mnesia:start()*.
- **Step 4:** A first table is created, called `funky`, by evaluating the expression `mnesia:create_table(funky, [])`. The table is given default properties.
- **Step 5:** *mnesia:info()* is evaluated to display information on the terminal about the status of the database.

1.3.2 Example

A `Mnesia` database is organized as a set of tables. Each table is populated with instances (Erlang records). A table has also a number of properties, such as location and persistence.

Database

This example shows how to create a database called `Company` and the relationships shown in the following diagram:

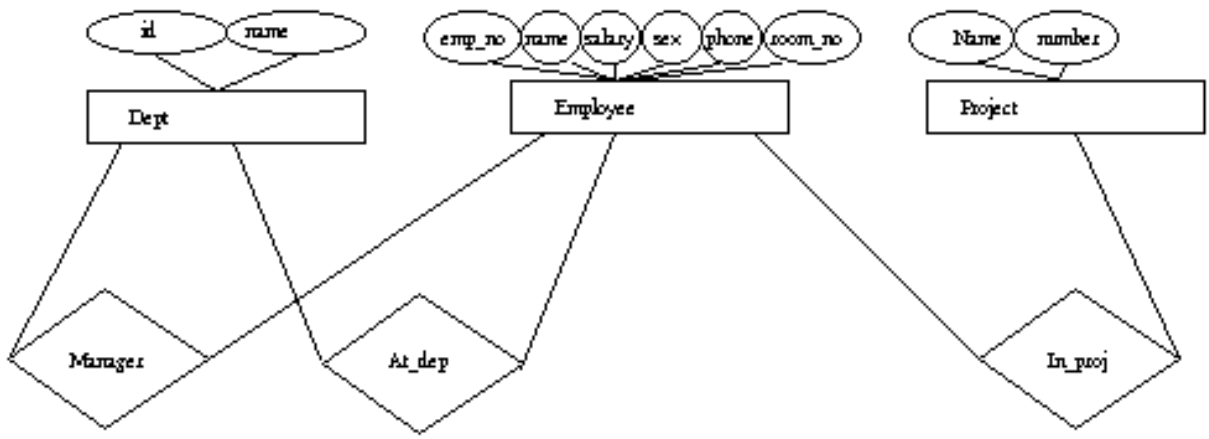


Figure 3.1: Company Entity-Relation Diagram

The database model is as follows:

- There are three entities: department, employee, and project.
- There are three relationships between these entities:
  - A department is managed by an employee, hence the `manager` relationship.
  - An employee works at a department, hence the `at_dep` relationship.
  - Each employee works on a number of projects, hence the `in_proj` relationship.

Defining Structure and Content

First the record definitions are entered into a text file named `company.hrl`. This file defines the following structure for the example database:

```
-record(employee, {emp_no,
                   name,
                   salary,
                   sex,
                   phone,
                   room_no}).

-record(dept, {id,
               name}).

-record(project, {name,
                  number}).


-record(manager, {emp,
                  dept}).

-record(at_dep, {emp,
                 dept_id}).

-record(in_proj, {emp,
                  proj_name}).
```

The structure defines six tables in the database. In `Mnesia`, the function *mnesia:create_table(Name, ArgList)* creates tables. `Name` is the table name.

Note:

The current version of `Mnesia` does not require that the name of the table is the same as the record name, see *Record Names versus Table Names.*.

For example, the table for employees is created with the function `mnesia:create_table(employee, [{attributes, record_info(fields, employee)}])`. The table name `employee` matches the name for records specified in `ArgList`. The expression `record_info(fields, RecordName)` is processed by the Erlang preprocessor and evaluates to a list containing the names of the different fields for a record.

Program

The following shell interaction starts `Mnesia` and initializes the schema for the `Company` database:

```
% erl -mnesia dir '"/ldisc/scratch/Mnesia.Company"'
 Erlang (BEAM) emulator version 4.9

  Eshell V4.9  (abort with ^G)
  1> mnesia:create_schema([node()]).
  ok
  2> mnesia:start().
  ok
```

The following program module creates and populates previously defined tables:

```
-include_lib("stdlib/include/qlc.hrl").
-include("company.hrl").

init() ->
    mnesia:create_table(employee,
                        [{attributes, record_info(fields, employee)}]),
    mnesia:create_table(dept,
                        [{attributes, record_info(fields, dept)}]),
    mnesia:create_table(project,
                        [{attributes, record_info(fields, project)}]),
    mnesia:create_table(manager, [{type, bag},
                                  {attributes, record_info(fields, manager)}]),
    mnesia:create_table(at_dep,
                         [{attributes, record_info(fields, at_dep)}]),
    mnesia:create_table(in_proj, [{type, bag},
                                  {attributes, record_info(fields, in_proj)}]).
```

Program Explained

The following commands and functions are used to initiate the `Company` database:

- `% erl -mnesia dir '"/ldisc/scratch/Mnesia.Company"'`. This is a UNIX command-line entry that starts the Erlang system. The flag `-mnesia dir Dir` specifies the location of the database directory. The system responds and waits for further input with the prompt `1>`.
- *mnesia:create_schema([node()])*. This function has the format `mnesia:create_schema(DiscNodeList)` and initiates a new schema. In this example, a non-distributed system using only one node is created. Schemas are fully explained in *Define a Schema*.
- *mnesia:start()*. This function starts `Mnesia` and is fully explained in *Start Mnesia*.

Continuing the dialogue with the Erlang shell produces the following:

```
3> company:init().
{atomic,ok}
4> mnesia:info().
---> Processes holding locks <---
---> Processes waiting for locks <---
---> Pending (remote) transactions <---
---> Active (local) transactions <---
---> Uncertain transactions <---
---> Active tables <---
in_proj        : with 0 records occuping 269 words of mem
at_dep         : with 0 records occuping 269 words of mem
manager        : with 0 records occuping 269 words of mem
project        : with 0 records occuping 269 words of mem
dept           : with 0 records occuping 269 words of mem
employee       : with 0 records occuping 269 words of mem
schema         : with 7 records occuping 571 words of mem
===> System info in version "1.0", debug level = none <===
opt_disc. Directory "/ldisc/scratch/Mnesia.Company" is used.
use fall-back at restart = false
running db nodes = [nonode@nohost]
stopped db nodes = []
remote           = []
ram_copies       =
    [at_dep,dept,employee,in_proj,manager,project]
disc_copies      = [schema]
disc_only_copies = []
[{nonode@nohost,disc_copies}] = [schema]
[{nonode@nohost,ram_copies}] =
    [employee,dept,project,manager,at_dep,in_proj]
6 transactions committed, 0 aborted, 0 restarted, 6 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
ok
```

A set of tables is created. The function *mnesia:create_table(Name, ArgList)* creates the required database tables. The options available with `ArgList` are explained in *Create New Tables*.

The function `company:init/0` creates the tables. Two tables are of type `bag`. This is the `manager` relation as well the `in_proj` relation. This is interpreted as: an employee can be manager over several departments, and an employee can participate in several projects. However, the `at_dep` relation is `set`, as an employee can only work in one department. In this data model, there are examples of relations that are 1-to-1 (`set`) and 1-to-many (`bag`).

mnesia:info() now indicates that a database has seven local tables, where six are the user-defined tables and one is the schema. Six transactions have been committed, as six successful transactions were run when creating the tables.

To write a function that inserts an employee record into the database, there must be an `at_dep` record and a set of `in_proj` records inserted. Examine the following code used to complete this action:

```
insert_emp(Emp, DeptId, ProjNames) ->
    Ename = Emp#employee.name,
    Fun = fun() ->
                  mnesia:write(Emp),
                  AtDep = #at_dep{emp = Ename, dept_id = DeptId},
                  mnesia:write(AtDep),
                  mk_projs(Ename, ProjNames)
          end,
    mnesia:transaction(Fun).


mk_projs(Ename, [ProjName|Tail]) ->
    mnesia:write(#in_proj{emp = Ename, proj_name = ProjName}),
    mk_projs(Ename, Tail);
mk_projs(_, []) -> ok.
```

- The `insert_emp/3` arguments are as follows:
  - `Emp` is an employee record.
  - `DeptId` is the identity of the department where the employee works.
  - `ProjNames` is a list of the names of the projects where the employee works.

The function `insert_emp/3` creates a Functional Object (Fun). `Fun` is passed as a single argument to the function *mnesia:transaction(Fun)*. This means that `Fun` is run as a transaction with the following properties:

- A `Fun` either succeeds or fails.
- Code that manipulates the same data records can be run concurrently without the different processes interfering with each other.

The function can be used as follows:

```
         Emp  = #employee{emp_no= 104732,
                          name = klacke,
                          salary = 7,
                          sex = male,
                          phone = 98108,
                          room_no = {221, 015}},
       insert_emp(Emp, 'B/SFR', [Erlang, mnesia, otp]).
```

Note:

For information about Funs, see "Fun Expressions" in section `Erlang Reference Manual` in System Documentation..

Initial Database Content

After the insertion of the employee named `klacke`, the database has the following records:

emp_no	name	salary	sex	phone	room_no
104732	klacke	7	male	98108	{221, 015}

Table 3.1: employee Database Record

This `employee` record has the Erlang record/tuple representation `{employee, 104732, klacke, 7, male, 98108, {221, 015}}`.

emp	dept_name
klacke	B/SFR

Table 3.2: at_dep Database Record

This `at_dep` record has the Erlang tuple representation `{at_dep, klacke, 'B/SFR'}`.

emp	proj_name
klacke	Erlang
klacke	otp
klacke	mnesia

Table 3.3: in_proj Database Record

This `in_proj` record has the Erlang tuple representation `{in_proj, klacke, 'Erlang', klacke, 'otp', klacke, 'mnesia'}`.

There is no difference between rows in a table and `Mnesia` records. Both concepts are the same and are used interchangeably throughout this User's Guide.

A `Mnesia` table is populated by `Mnesia` records. For example, the tuple `{boss, klacke, bjarne}` is a record. The second element in this tuple is the key. To identify a table uniquely, both the key and the table name is needed. The term Object Identifier (OID) is sometimes used for the arity two tuple {Tab, Key}. The OID for the record `{boss, klacke, bjarne}` is the arity two tuple `{boss, klacke}`. The first element of the tuple is the type of the record and the second element is the key. An OID can lead to zero, one, or more records depending on whether the table type is `set` or `bag`.

The record `{boss, klacke, bjarne}` can also be inserted. This record contains an implicit reference to another employee that does not yet exist in the database. `Mnesia` does not enforce this.

Adding Records and Relationships to Database

After adding more records to the `Company` database, the result can be the following records:

`employees`:

```
        {employee, 104465, "Johnson Torbjorn",   1, male,  99184, {242,038}}.
        {employee, 107912, "Carlsson Tuula",     2, female,94556, {242,056}}.
        {employee, 114872, "Dacker Bjarne",      3, male,  99415, {221,035}}.
        {employee, 104531, "Nilsson Hans",       3, male,  99495, {222,026}}.
        {employee, 104659, "Tornkvist Torbjorn", 2, male,  99514, {222,022}}.
        {employee, 104732, "Wikstrom Claes",     2, male,  99586, {221,015}}.
        {employee, 117716, "Fedoriw Anna",       1, female,99143, {221,031}}.
        {employee, 115018, "Mattsson Hakan",     3, male,  99251, {203,348}}.
```

`dept`:

```
        {dept, 'B/SF',  "Open Telecom Platform"}.
        {dept, 'B/SFP', "OTP - Product Development"}.
        {dept, 'B/SFR', "Computer Science Laboratory"}.
```

`projects`:

```
%% projects
{project, erlang, 1}.
{project, otp, 2}.
{project, beam, 3}.
{project, mnesia, 5}.
{project, wolf, 6}.
{project, documentation, 7}.
{project, www, 8}.
```

These three tables, `employees`, `dept`, and `projects`, are made up of real records. The following database content is stored in the tables and is built on relationships. These tables are `manager`, `at_dep`, and `in_proj`.

`manager`:

```
{manager, 104465, 'B/SF'}.
{manager, 104465, 'B/SFP'}.
{manager, 114872, 'B/SFR'}.
```

`at_dep`:

```
{at_dep, 104465, 'B/SF'}.
{at_dep, 107912, 'B/SF'}.
{at_dep, 114872, 'B/SFR'}.
{at_dep, 104531, 'B/SFR'}.
{at_dep, 104659, 'B/SFR'}.
{at_dep, 104732, 'B/SFR'}.
{at_dep, 117716, 'B/SFP'}.
{at_dep, 115018, 'B/SFP'}.
```

`in_proj`:

```
{in_proj, 104465, otp}.
{in_proj, 107912, otp}.
{in_proj, 114872, otp}.
{in_proj, 104531, otp}.
{in_proj, 104531, mnesia}.
{in_proj, 104545, wolf}.
{in_proj, 104659, otp}.
{in_proj, 104659, wolf}.
{in_proj, 104732, otp}.
{in_proj, 104732, mnesia}.
{in_proj, 104732, erlang}.
{in_proj, 117716, otp}.
{in_proj, 117716, documentation}.
{in_proj, 115018, otp}.
{in_proj, 115018, mnesia}.
```

The room number is an attribute of the employee record. This is a structured attribute that consists of a tuple. The first element of the tuple identifies a corridor, and the second element identifies the room in that corridor. An alternative is to represent this as a record `-record(room, {corr, no}).` instead of an anonymous tuple representation.

The `Company` database is now initialized and contains data.

Writing Queries

Retrieving data from DBMS is usually to be done with the functions *mnesia:read/3* or *mnesia:read/1*. The following function raises the salary:

```
raise(Eno, Raise) ->
    F = fun() ->
                [E] = mnesia:read(employee, Eno, write),
                Salary = E#employee.salary + Raise,
                New = E#employee{salary = Salary},
                mnesia:write(New)
        end,
    mnesia:transaction(F).
```

Since it is desired to update the record using the function *mnesia:write/1* after the salary has been increased, a write lock (third argument to `read`) is acquired when the record from the table is read.

To read the values from the table directly is not always possible. It can be needed to search one or more tables to get the wanted data, and this is done by writing database queries. Queries are always more expensive operations than direct lookups done with `mnesia:read`. Therefore, avoid queries in performance-critical code.

Two methods are available for writing database queries:

- `Mnesia` functions
- QLC

Using Mnesia Functions

The following function extracts the names of the female employees stored in the database:

```
mnesia:select(employee, [{#employee{sex = female, name = '$1', _ = '_'},[], ['$1']}]).
```

`select` must always run within an activity, such as a transaction. The following function can be constructed to call from the shell:

```
all_females() ->
    F = fun() ->
  Female = #employee{sex = female, name = '$1', _ = '_'},
  mnesia:select(employee, [{Female, [], ['$1']}])
        end,
    mnesia:transaction(F).
```

The `select` expression matches all entries in table employee with the field `sex` set to `female`.

This function can be called from the shell as follows:

```
        (klacke@gin)1> company:all_females().
        {atomic,  ["Carlsson Tuula", "Fedoriw Anna"]}
```

For a description of `select` and its syntax, see *Pattern Matching*.

Using QLC

This section contains simple introductory examples only. For a full description of the QLC query language, see the *qlc* manual page in `STDLIB`.

Using QLC can be more expensive than using `Mnesia` functions directly but offers a nice syntax.

The following function extracts a list of female employees from the database:

```
        Q = qlc:q([E#employee.name || E <- mnesia:table(employee),
                          E#employee.sex == female]),
        qlc:e(Q),
```

Accessing `Mnesia` tables from a QLC list comprehension must always be done within a transaction. Consider the following function:

```
females() ->
    F = fun() ->
  Q = qlc:q([E#employee.name || E <- mnesia:table(employee),
           E#employee.sex == female]),
  qlc:e(Q)
 end,
    mnesia:transaction(F).
```

This function can be called from the shell as follows:

```
         (klacke@gin)1> company:females().
         {atomic, ["Carlsson Tuula", "Fedoriw Anna"]}
```

In traditional relational database terminology, this operation is called a selection, followed by a projection.

The previous list comprehension expression contains a number of syntactical elements:

- The first `[` bracket is read as "build the list".
- The `||` "such that" and the arrow `<-` is read as "taken from".

Hence, the previous list comprehension demonstrates the formation of the list `E#employee.name` such that `E` is taken from the table of employees, and attribute `sex` of each record is equal to the atom `female`.

The whole list comprehension must be given to the function `qlc:q/1`.

List comprehensions with low-level `Mnesia` functions can be combined in the same transaction. To raise the salary of all female employees, execute the following:

```
raise_females(Amount) ->
    F = fun() ->
                Q = qlc:q([E || E <- mnesia:table(employee),
                                E#employee.sex == female]),
  Fs = qlc:e(Q),
                over_write(Fs, Amount)
        end,
    mnesia:transaction(F).

over_write([E|Tail], Amount) ->
    Salary = E#employee.salary + Amount,
    New = E#employee{salary = Salary},
    mnesia:write(New),
    1 + over_write(Tail, Amount);
over_write([], _) ->
    0.
```

The function `raise_females/1` returns the tuple `{atomic, Number}`, where `Number` is the number of female employees who received a salary increase. If an error occurs, the value `{aborted, Reason}` is returned, and `Mnesia` guarantees that the salary is not raised for any employee.

Example:

```
         33>company:raise_females(33).
         {atomic,2}
```

1.4 Build a Mnesia Database

This section describes the basic steps when designing a `Mnesia` database and the programming constructs that make different solutions available to the programmer. The following topics are included:

- Define a schema

- Data model
- Start `Mnesia`
- Create tables

1.4.1 Define a Schema

The configuration of a `Mnesia` system is described in a schema. The schema is a special table that includes information such as the table names and the storage type of each table (that is, whether a table is to be stored in RAM, on disc, or on both, as well as its location).

Unlike data tables, information in schema tables can only be accessed and modified by using the schema-related functions described in this section.

`Mnesia` has various functions for defining the database schema. Tables can be moved or deleted, and the table layout can be reconfigured.

An important aspect of these functions is that the system can access a table while it is being reconfigured. For example, it is possible to move a table and simultaneously perform write operations to the same table. This feature is essential for applications that require continuous service.

This section describes the functions available for schema management, all which return either of the following tuples:

- `{atomic, ok}` if successful
- `{aborted, Reason}` if unsuccessful

Schema Functions

The schema functions are as follows:

- *mnesia:create_schema(NodeList)* initializes a new, empty schema. This is a mandatory requirement before `Mnesia` can be started. `Mnesia` is a truly distributed DBMS and the schema is a system table that is replicated on all nodes in a `Mnesia` system. This function fails if a schema is already present on any of the nodes in `NodeList`. The function requires `Mnesia` to be stopped on the all `db_nodes` contained in parameter `NodeList`. Applications call this function only once, as it is usually a one-time activity to initialize a new database.
- *mnesia:delete_schema(DiscNodeList)* erases any old schemas on the nodes in `DiscNodeList`. It also removes all old tables together with all data. This function requires `Mnesia` to be stopped on all `db_nodes`.
- *mnesia:delete_table(Tab)* permanently deletes all replicas of table `Tab`.
- *mnesia:clear_table(Tab)* permanently deletes all entries in table `Tab`.
- *mnesia:move_table_copy(Tab, From, To)* moves the copy of table `Tab` from node `From` to node `To`. The table storage type `{type}` is preserved, so if a RAM table is moved from one node to another, it remains a RAM table on the new node. Other transactions can still perform read and write operation to the table while it is being moved.
- *mnesia:add_table_copy(Tab, Node, Type)* creates a replica of table `Tab` at node `Node`. Argument `Type` must be either of the atoms `ram_copies`, `disc_copies`, or `disc_only_copies`. If you add a copy of the system table `schema` to a node, you want the `Mnesia` schema to reside there as well. This action extends the set of nodes that comprise this particular `Mnesia` system.
- *mnesia:del_table_copy(Tab, Node)* deletes the replica of table `Tab` at node `Node`. When the last replica of a table is removed, the table is deleted.

- *mnesia:transform_table(Tab, Fun, NewAttributeList, NewRecordName)* changes the format on all records in table `Tab`. It applies argument `Fun` to all records in the table. `Fun` must be a function that takes a record of the old type, and returns the record of the new type. The table key must not be changed.

  Example:

```
-record(old, {key, val}).
-record(new, {key, val, extra}).

Transformer =
   fun(X) when record(X, old) ->
      #new{key = X#old.key,
           val = X#old.val,
           extra = 42}
   end,
{atomic, ok} = mnesia:transform_table(foo, Transformer,
                                      record_info(fields, new),
                                      new),
```

  Argument `Fun` can also be the atom `ignore`, which indicates that only the metadata about the table is updated. Use of `ignore` is not recommended (as it creates inconsistencies between the metadata and the actual data) but it is included as a possibility for the user do to an own (offline) transform.
- `change_table_copy_type(Tab, Node, ToType)` changes the storage type of a table. For example, a RAM table is changed to a `disc_table` at the node specified as `Node`.

1.4.2 Data Model

The data model employed by `Mnesia` is an extended relational data model. Data is organized as a set of tables and relations between different data records can be modeled as more tables describing the relationships. Each table contains instances of Erlang records. The records are represented as Erlang tuples.

Each Object Identifier (OID) is made up of a table name and a key. For example, if an employee record is represented by the tuple `{employee, 104732, klacke, 7, male, 98108, {221, 015}}`, this record has an OID, which is the tuple `{employee, 104732}`.

Thus, each table is made up of records, where the first element is a record name and the second element of the table is a key, which identifies the particular record in that table. The combination of the table name and a key is an arity two tuple `{Tab, Key}` called the OID. For more information about the relationship beween the record name and the table name, see *Record Names versus Table Names*.

What makes the `Mnesia` data model an extended relational model is the ability to store arbitrary Erlang terms in the attribute fields. One attribute value can, for example, be a whole tree of OIDs leading to other terms in other tables. This type of record is difficult to model in traditional relational DBMSs.

1.4.3 Start Mnesia

Before starting `Mnesia`, the following must be done:

- An empty schema must be initialized on all the participating nodes.
- The Erlang system must be started.
- Nodes with disc database schema must be defined and implemented with the function *mnesia:create_schema(NodeList)*.

When running a distributed system with two or more participating nodes, the function *mnesia:start()* must be executed on each participating node. This would typically be part of the boot script in an embedded environment. In a test environment or an interactive environment, `mnesia:start()` can also be used either from the Erlang shell or another program.

Initialize a Schema and Start Mnesia

Let us use the example database `Company`, described in *Getting Started* to illustrate how to run a database on two separate nodes, called `a@gin` and `b@skeppet`. Each of these nodes must have a `Mnesia` directory and an initialized schema before `Mnesia` can be started. There are two ways to specify the `Mnesia` directory to be used:

- Specify the `Mnesia` directory by providing an application parameter either when starting the Erlang shell or in the application script. Previously, the following example was used to create the directory for the `Company` database:

```
%erl -mnesia dir '"/ldisc/scratch/Mnesia.Company"'
```

- If no command-line flag is entered, the `Mnesia` directory becomes the current working directory on the node where the Erlang shell is started.

To start the `Company` database and get it running on the two specified nodes, enter the following commands:

- On the node `a@gin`:

```
 gin %erl -sname a  -mnesia dir '"/ldisc/scratch/Mnesia.company"'
```

- On the node `b@skeppet`:

```
skeppet %erl -sname b -mnesia dir '"/ldisc/scratch/Mnesia.company"'
```

- On one of the two nodes:

```
(a@gin)1>mnesia:create_schema([a@gin, b@skeppet]).
```

- The function *mnesia:start()* is called on both nodes.
- To initialize the database, execute the following code on one of the two nodes:

```
dist_init() ->
    mnesia:create_table(employee,
                        [{ram_copies, [a@gin, b@skeppet]},
                         {attributes, record_info(fields,
         employee)}]),
    mnesia:create_table(dept,
                        [{ram_copies, [a@gin, b@skeppet]},
                         {attributes, record_info(fields, dept)}]),
    mnesia:create_table(project,
                        [{ram_copies, [a@gin, b@skeppet]},
                         {attributes, record_info(fields, project)}]),
    mnesia:create_table(manager, [{type, bag},
                                  {ram_copies, [a@gin, b@skeppet]},
                                  {attributes, record_info(fields,
          manager)}]),
    mnesia:create_table(at_dep,
                        [{ram_copies, [a@gin, b@skeppet]},
                         {attributes, record_info(fields, at_dep)}]),
    mnesia:create_table(in_proj,
                       [{type, bag},
                        {ram_copies, [a@gin, b@skeppet]},
                        {attributes, record_info(fields, in_proj)}]).
```

As illustrated, the two directories reside on different nodes, because `/ldisc/scratch` (the "local" disc) exists on the two different nodes.

By executing these commands, two Erlang nodes are configured to run the `Company` database, and therefore, initialize the database. This is required only once when setting up. The next time the system is started, *mnesia:start()* is called on both nodes, to initialize the system from disc.

In a system of `Mnesia` nodes, every node is aware of the current location of all tables. In this example, data is replicated on both nodes and functions that manipulate the data in the tables can be executed on either of the two nodes. Code that manipulate `Mnesia` data behaves identically regardless of where the data resides.

The function *mnesia:stop()* stops `Mnesia` on the node where the function is executed. The functions `mnesia:start/0` and `mnesia:stop/0` work on the "local" `Mnesia` system. No functions start or stop a set of nodes.

Startup Procedure

Start `Mnesia` by calling the following function:

```
mnesia:start().
```

This function initiates the DBMS locally.

The choice of configuration alters the location and load order of the tables. The alternatives are as follows:

- Tables that are only stored locally are initialized from the local `Mnesia` directory.
- Replicated tables that reside locally as well as somewhere else are either initiated from disc or by copying the entire table from the other node, depending on which of the different replicas are the most recent. `Mnesia` determines which of the tables are the most recent.
- Tables that reside on remote nodes are available to other nodes as soon as they are loaded.

Table initialization is asynchronous. The function call *mnesia:start()* returns the atom `ok` and then starts to initialize the different tables. Depending on the size of the database, this can take some time, and the application programmer must wait for the tables that the application needs before they can be used. This is achieved by using the function *mnesia:wait_for_tables(TabList, Timeout)*, which suspends the caller until all tables specified in `TabList` are properly initiated.

A problem can arise if a replicated table on one node is initiated, but `Mnesia` deduces that another (remote) replica is more recent than the replica existing on the local node, and the initialization procedure does not proceed. In this situation, a call to *mnesia:wait_for_tables/2*, suspends the caller until the remote node has initialized the table from its local disc and the node has copied the table over the network to the local node.

However, this procedure can be time-consuming, the shortcut function *mnesia:force_load_table(Tab)* loads all the tables from disc at a faster rate. The function forces tables to be loaded from disc regardless of the network situation.

Thus, it can be assumed that if an application wants to use tables `a` and `b`, the application must perform some action similar to following before it can use the tables:

```
case mnesia:wait_for_tables([a, b], 20000) of
  {timeout,   RemainingTabs} ->
    panic(RemainingTabs);
  ok ->
    synced
end.
```

Warning:

When tables are forcefully loaded from the local disc, all operations that were performed on the replicated table while the local node was down, and the remote replica was alive, are lost. This can cause the database to become inconsistent.

If the startup procedure fails, the function *mnesia:start()* returns the cryptic tuple `{error,{shutdown, {mnesia_sup,start_link,[normal,[]]}}}`. To get more information about the start failure, use command-line arguments `-boot start_sasl` as argument to the `erl` script.

1.4.4 Create Tables

The function *mnesia:create_table(Name, ArgList)* creates tables. When executing this function, it returns one of the following responses:

- `{atomic, ok}` if the function executes successfully
- `{aborted, Reason}` if the function fails

The function arguments are as follows:

- `Name` is the name of the table. It is usually the same name as the name of the records that constitute the table. For details, see `record_name`.

- `ArgList` is a list of `{Key,Value}` tuples. The following arguments are valid:
  - `{type, Type}`, where `Type` must be either of the atoms `set`, `ordered_set`, or `bag`. Default is `set`.

    Notice that currently `ordered_set` is not supported for `disc_only_copies` tables.

    A table of type `set` or `ordered_set` has either zero or one record per key, whereas a table of type `bag` can have an arbitrary number of records per key. The key for each record is always the first attribute of the record.

    The following example illustrates the difference between type `set` and `bag`:

    ```
    f() ->
       F = fun() ->
             mnesia:write({foo, 1, 2}),
             mnesia:write({foo, 1, 3}),
             mnesia:read({foo, 1})
           end,
       mnesia:transaction(F).
    ```

    This transaction returns the list `[{foo,1,3}]` if table `foo` is of type `set`. However, the list `[{foo,1,2}, {foo,1,3}]` is returned if the table is of type `bag`.

    `Mnesia` tables can never contain duplicates of the same record in the same table. Duplicate records have attributes with the same contents and key.
  - `{disc_copies, NodeList}`, where `NodeList` is a list of the nodes where this table is to reside on disc.

    Write operations to a table replica of type `disc_copies` write data to the disc copy and to the RAM copy of the table.

    It is possible to have a replicated table of type `disc_copies` on one node, and the same table stored as a different type on another node. Default is `[]`. This arrangement is desirable if the following operational characteristics are required:

    - Read operations must be fast and performed in RAM.
    - All write operations must be written to persistent storage.

    A write operation on a `disc_copies` table replica is performed in two steps. First the write operation is appended to a log file, then the actual operation is performed in RAM.
  - `{ram_copies, NodeList}`, where `NodeList` is a list of the nodes where this table is stored in RAM. Default is `[node()]`. If the default value is used to create a table, it is located on the local node only.

    Table replicas of type `ram_copies` can be dumped to disc with the function *mnesia:dump_tables(TabList)*.
  - `{disc_only_copies, NodeList}`. These table replicas are stored on disc only and are therefore slower to access. However, a disc-only replica consumes less memory than a table replica of the other two storage types.
  - `{index, AttributeNameList}`, where `AttributeNameList` is a list of atoms specifying the names of the attributes `Mnesia` is to build and maintain. An index table exists for every element in the list. The first field of a `Mnesia` record is the key and thus need no extra index.

    The first field of a record is the second element of the tuple, which is the representation of the record.
  - `{snmp, SnmpStruct}`. `SnmpStruct` is described in the *SNMP* User's Guide. Basically, if this attribute is present in `ArgList` of *mnesia:create_table/2*, the table is immediately accessible the SNMP.

    It is easy to design applications that use SNMP to manipulate and control the system. `Mnesia` provides a direct mapping between the logical tables that make up an SNMP control application and the physical data that makes up a `Mnesia` table. The default value is `[]`.
  - `{local_content, true}`. When an application needs a table whose contents is to be locally unique on each node, `local_content` tables can be used. The name of the table is known to all `Mnesia` nodes, but its contents is unique for each node. Access to this type of table must be done locally.

- `{attributes, AtomList}` is a list of the attribute names for the records that are supposed to populate the table. Default is the list `[key, val]`. The table must at least have one extra attribute besides the key. When accessing single attributes in a record, it is not recommended to hard code the attribute names as atoms. Use the construct `record_info(fields, record_name)` instead.

  The expression `record_info(fields, record_name)` is processed by the Erlang preprocessor and returns a list of the record field names. With the record definition `-record(foo, {x,y,z}).`, the expression `record_info(fields,foo)` is expanded to the list `[x,y,z]`. It is therefore possible for you to provide the attribute names or to use the `record_info/2` notation.

  It is recommended to use the `record_info/2` notation, as it becomes easier to maintain the program and the program becomes more robust with regards to future record changes.
- `{record_name, Atom}` specifies the common name of all records stored in the table. All records stored in the table must have this name as their first element. `record_name` defaults to the name of the table. For more information, see *Record Names versus Table Names*.

As an example, consider the following record definition:

```
-record(funky, {x, y}).
```

The following call would create a table that is replicated on two nodes, has an extra index on attribute `y`, and is of type `bag`.

```
mnesia:create_table(funky, [{disc_copies, [N1, N2]}, {index,
[y]}, {type, bag}, {attributes, record_info(fields, funky)}]).
```

Whereas a call to the following default code values would return a table with a RAM copy on the local node, no extra indexes, and the attributes defaulted to the list `[key,val]`.

```
mnesia:create_table(stuff, [])
```

1.5 Transactions and Other Access Contexts

This section describes the `Mnesia` transaction system and the transaction properties that make `Mnesia` a fault-tolerant, distributed Database Management System (DBMS).

This section also describes the locking functions, including table locks and sticky locks, as well as alternative functions that bypass the transaction system in favor of improved speed and reduced overhead. These functions are called "dirty operations". The use of nested transactions is also described. The following topics are included:

- Transaction properties, which include atomicity, consistency, isolation, and durability
- Locking
- Dirty operations
- Record names versus table names
- Activity concept and various access contexts
- Nested transactions
- Pattern matching
- Iteration

1.5.1 Transaction Properties

Transactions are important when designing fault-tolerant, distributed systems. A `Mnesia` transaction is a mechanism by which a series of database operations can be executed as one functional block. The functional block that is run as a transaction is called a Functional Object (Fun), and this code can read, write, and delete `Mnesia` records. The Fun is evaluated as a transaction that either commits or terminates. If a transaction succeeds in executing the Fun, it replicates the action on all nodes involved, or terminates if an error occurs.

The following example shows a transaction that raises the salary of certain employee numbers:

```
raise(Eno, Raise) ->
    F = fun() ->
                [E] = mnesia:read(employee, Eno, write),
                Salary = E#employee.salary + Raise,
                New = E#employee{salary = Salary},
                mnesia:write(New)
        end,
    mnesia:transaction(F).
```

The function `raise/2` contains a Fun made up of four code lines. This Fun is called by the statement `mnesia:transaction(F)` and returns a value.

The `Mnesia` transaction system facilitates the construction of reliable, distributed systems by providing the following important properties:

- The transaction handler ensures that a Fun, which is placed inside a transaction, does not interfere with operations embedded in other transactions when it executes a series of operations on tables.
- The transaction handler ensures that either all operations in the transaction are performed successfully on all nodes atomically, or the transaction fails without permanent effect on any node.
- The `Mnesia` transactions have four important properties, called **A**tomicity, **C**onsistency, **I**solation, and **D**urability (ACID). These properties are described in the following sections.

Atomicity

Atomicity means that database changes that are executed by a transaction take effect on all nodes involved, or on none of the nodes. That is, the transaction either succeeds entirely, or it fails entirely.

Atomicity is important when it is needed to write atomically more than one record in the same transaction. The function `raise/2`, shown in the previous example, writes one record only. The function `insert_emp/3`, shown in the program listing in *Getting Started*, writes the record `employee` as well as employee relations, such as `at_dep` and `in_proj`, into the database. If this latter code is run inside a transaction, the transaction handler ensures that the transaction either succeeds completely, or not at all.

`Mnesia` is a distributed DBMS where data can be replicated on several nodes. In many applications, it is important that a series of write operations are performed atomically inside a transaction. The atomicity property ensures that a transaction takes effect on all nodes, or none.

Consistency

The consistency property ensures that a transaction always leaves the DBMS in a consistent state. For example, `Mnesia` ensures that no inconsistencies occur if Erlang, `Mnesia`, or the computer crashes while a write operation is in progress.

Isolation

The isolation property ensures that transactions that execute on different nodes in a network, and access and manipulate the same data records, do not interfere with each other. The isolation property makes it possible to execute the function `raise/2` concurrently. A classical problem in concurrency control theory is the "lost update problem".

The isolation property is in particular useful if the following circumstances occur where an employee (with employee number 123) and two processes (P1 and P2) are concurrently trying to raise the salary for the employee:

- **Step 1:** The initial value of the employees salary is, for example, 5. Process P1 starts to execute, reads the employee record, and adds 2 to the salary.
- **Step 2:** Process P1 is for some reason pre-empted and process P2 has the opportunity to run.
- **Step 3:** Process P2 reads the record, adds 3 to the salary, and finally writes a new employee record with the salary set to 8.
- **Step 4:** Process P1 starts to run again and writes its employee record with salary set to 7, thus effectively overwriting and undoing the work performed by process P2. The update performed by P2 is lost.

A transaction system makes it possible to execute two or more processes concurrently that manipulate the same record. The programmer does not need to check that the updates are synchronous; this is overseen by the transaction handler. All programs accessing the database through the transaction system can be written as if they had sole access to the data.

Durability

The durability property ensures that changes made to the DBMS by a transaction are permanent. Once a transaction is committed, all changes made to the database are durable, that is, they are written safely to disc and do not become corrupted and do not disappear.

> **Note:**
>
> The described durability feature does not entirely apply to situations where `Mnesia` is configured as a "pure" primary memory database.

1.5.2 Locking

Different transaction managers employ different strategies to satisfy the isolation property. `Mnesia` uses the standard technique of two phase locking. That is, locks are set on records before they are read or written. `Mnesia` uses the following lock types:

- **Read locks**. A read lock is set on one replica of a record before it can be read.
- **Write locks**. Whenever a transaction writes to a record, write locks are first set on all replicas of that particular record.
- **Read table locks**. If a transaction traverses an entire table in search for a record that satisfies some particular property, it is most inefficient to set read locks on the records one by one. It is also memory consuming, as the read locks themselves can take up considerable space if the table is large. Therefore, `Mnesia` can set a read lock on an entire table.
- **Write table locks**. If a transaction writes many records to one table, a write lock can be set on the entire table.
- **Sticky locks**. These are write locks that stay in place at a node after the transaction that initiated the lock has terminated.

`Mnesia` employs a strategy whereby functions, such as *mnesia:read/1* acquire the necessary locks dynamically as the transactions execute. `Mnesia` automatically sets and releases the locks and the programmer does not need to code these operations.

Deadlocks can occur when concurrent processes set and release locks on the same records. `Mnesia` employs a "wait-die" strategy to resolve these situations. If `Mnesia` suspects that a deadlock can occur when a transaction tries to set a lock, the transaction is forced to release all its locks and sleep for a while. The Fun in the transaction is evaluated once more.

It is therefore important that the code inside the Fun given to *`mnesia:transaction/1`* is pure. Some strange results can occur if, for example, messages are sent by the transaction Fun. The following example illustrates this situation:

```
bad_raise(Eno, Raise) ->
    F = fun() ->
                [E] = mnesia:read({employee, Eno}),
                Salary = E#employee.salary + Raise,
                New = E#employee{salary = Salary},
                io:format("Trying to write ... ~n", []),
                mnesia:write(New)
        end,
    mnesia:transaction(F).
```

This transaction can write the text `"Trying to write ... "` 1000 times to the terminal. However, `Mnesia` guarantees that each transaction will eventually run. As a result, `Mnesia` is not only deadlock free, but also livelock free.

The `Mnesia` programmer cannot prioritize one particular transaction to execute before other transactions that are waiting to execute. As a result, the `Mnesia` DBMS transaction system is not suitable for hard real-time applications. However, `Mnesia` contains other features that have real-time properties.

`Mnesia` dynamically sets and releases locks as transactions execute. It is therefore dangerous to execute code with transaction side-effects. In particular, a `receive` statement inside a transaction can lead to a situation where the transaction hangs and never returns, which in turn can cause locks not to release. This situation can bring the whole system to a standstill, as other transactions that execute in other processes, or on other nodes, are forced to wait for the defective transaction.

If a transaction terminates abnormally, `Mnesia` automatically releases the locks held by the transaction.

Up to now, examples of a number of functions that can be used inside a transaction have been shown. The following list shows the **simplest** `Mnesia` functions that work with transactions. Notice that these functions must be embedded in a transaction. If no enclosing transaction (or other enclosing `Mnesia` activity) exists, they all fail.

- *mnesia:transaction(Fun) -> {aborted, Reason} |{atomic, Value}* executes one transaction with the functional object `Fun` as the single parameter.
- *mnesia:read({Tab, Key}) -> transaction abort | RecordList* reads all records with `Key` as key from table `Tab`. This function has the same semantics regardless of the location of `Table`. If the table is of type `bag`, `read({Tab, Key})` can return an arbitrarily long list. If the table is of type `set`, the list is either of length one or `[]`.
- *mnesia:wread({Tab, Key}) -> transaction abort | RecordList* behaves the same way as the previously listed function `read/1`, except that it acquires a write lock instead of a read lock. To execute a transaction that reads a record, modifies the record, and then writes the record, it is slightly more efficient to set the write lock immediately. When a *mnesia:read/1* is issued, followed by a *mnesia:write/1* the first read lock must be upgraded to a write lock when the write operation is executed.
- *mnesia:write(Record) -> transaction abort | ok* writes a record into the database. Argument `Record` is an instance of a record. The function returns `ok`, or terminates the transaction if an error occurs.
- *mnesia:delete({Tab, Key}) -> transaction abort | ok* deletes all records with the given key.
- *mnesia:delete_object(Record) -> transaction abort | ok* deletes records with the OID `Record`. Use this function to delete only some records in a table of type `bag`.

Sticky Locks

As previously stated, the locking strategy used by `Mnesia` is to lock one record when reading a record, and lock all replicas of a record when writing a record. However, some applications use `Mnesia` mainly for its fault-tolerant qualities. These applications can be configured with one node doing all the heavy work, and a standby node that is ready to take over if the main node fails. Such applications can benefit from using sticky locks instead of the normal locking scheme.

A sticky lock is a lock that stays in place at a node, after the transaction that first acquired the lock has terminated. To illustrate this, assume that the following transaction is executed:

```
F = fun() ->
      mnesia:write(#foo{a = kalle})
    end,
mnesia:transaction(F).
```

The `foo` table is replicated on the two nodes `N1` and `N2`.

Normal locking requires the following:

- One network RPC (two messages) to acquire the write lock
- Three network messages to execute the two-phase commit protocol

If sticky locks are used, the code must first be changed as follows:

```
F = fun() ->
      mnesia:s_write(#foo{a = kalle})
    end,
mnesia:transaction(F).
```

This code uses the function *s_write/1* instead of the function *write/1* The function `s_write/1` sets a sticky lock instead of a normal lock. If the table is not replicated, sticky locks have no special effect. If the table is replicated, and a sticky lock is set on node `N1`, this lock then sticks to node `N1`. The next time you try to set a sticky lock on the same record at node `N1`, `Mnesia` detects that the lock is already set and do no network operation to acquire the lock.

It is more efficient to set a local lock than it is to set a networked lock. Sticky locks can therefore benefit an application that uses a replicated table and perform most of the work on only one of the nodes.

If a record is stuck at node `N1` and you try to set a sticky lock for the record on node `N2`, the record must be unstuck. This operation is expensive and reduces performance. The unsticking is done automatically if you issue `s_write/1` requests at `N2`.

Table Locks

`Mnesia` supports read and write locks on whole tables as a complement to the normal locks on single records. As previously stated, `Mnesia` sets and releases locks automatically, and the programmer does not need to code these operations. However, transactions that read and write many records in a specific table execute more efficiently if the transaction is started by setting a table lock on this table. This blocks other concurrent transactions from the table. The following two functions are used to set explicit table locks for read and write operations:

- *mnesia:read_lock_table(Tab)* sets a read lock on table `Tab`.
- *mnesia:write_lock_table(Tab)* sets a write lock on table `Tab`.

Alternative syntax for acquisition of table locks is as follows:

```
mnesia:lock({table, Tab}, read)
mnesia:lock({table, Tab}, write)
```

The matching operations in `Mnesia` can either lock the entire table or only a single record (when the key is bound in the pattern).

Global Locks

Write locks are normally acquired on all nodes where a replica of the table resides (and is active). Read locks are acquired on one node (the local one if a local replica exists).

The function *mnesia:lock/2* is intended to support table locks (as mentioned previously) but also for situations when locks need to be acquired regardless of how tables have been replicated:

```
mnesia:lock({global, GlobalKey, Nodes}, LockKind)

LockKind ::= read | write | ...
```

The lock is acquired on `LockItem` on all nodes in the node list.

1.5.3 Dirty Operations

In many applications, the overhead of processing a transaction can result in a loss of performance. Dirty operation are short cuts that bypass much of the processing and increase the speed of the transaction.

Dirty operation are often useful, for example, in a datagram routing application where `Mnesia` stores the routing table, and it is time consuming to start a whole transaction every time a packet is received. `Mnesia` has therefore functions that manipulate tables without using transactions. This alternative to processing is known as a dirty operation. However, notice the trade-off in avoiding the overhead of transaction processing:

- The atomicity and the isolation properties of `Mnesia` are lost.
- The isolation property is compromised, because other Erlang processes, which use transaction to manipulate the data, do not get the benefit of isolation if dirty operations simultaneously are used to read and write records from the same table.

The major advantage of dirty operations is that they execute much faster than equivalent operations that are processed as functional objects within a transaction.

Dirty operations are written to disc if they are performed on a table of type `disc_copies` or type `disc_only_copies`. `Mnesia` also ensures that all replicas of a table are updated if a dirty write operation is performed on a table.

A dirty operation ensures a certain level of consistency. For example, dirty operations cannot return garbled records. Hence, each individual read or write operation is performed in an atomic manner.

All dirty functions execute a call to `exit({aborted, Reason})` on failure. Even if the following functions are executed inside a transaction no locks are acquired. The following functions are available:

- *mnesia:dirty_read({Tab, Key})* reads one or more records from `Mnesia`.
- *mnesia:dirty_write(Record)* writes the record `Record`.
- *mnesia:dirty_delete({Tab, Key})* deletes one or more records with key `Key`.
- *mnesia:dirty_delete_object(Record)* is the dirty operation alternative to the function *delete_object/1*.
- *mnesia:dirty_first(Tab)* returns the "first" key in table `Tab`.

  Records in `set` or `bag` tables are not sorted. However, there is a record order that is unknown to the user. This means that a table can be traversed by this function with the function *mnesia:dirty_next/2*.

  If there are no records in the table, this function returns the atom `'$end_of_table'`. It is not recommended to use this atom as the key for any user records.
- *mnesia:dirty_next(Tab, Key)* returns the "next" key in table `Tab`. This function makes it possible to traverse a table and perform some operation on all records in the table. When the end of the table is reached, the special key `'$end_of_table'` is returned. Otherwise, the function returns a key that can be used to read the actual record.

  The behavior is undefined if any process performs a write operation on the table while traversing the table with the function *dirty_next/2* This is because `write` operations on a `Mnesia` table can lead to internal reorganizations of the table itself. This is an implementation detail, but remember that the dirty functions are low-level functions.
- *mnesia:dirty_last(Tab)* works exactly like *mnesia:dirty_first/1* but returns the last object in Erlang term order for the table type `ordered_set`. For all other table types, `mnesia:dirty_first/1` and `mnesia:dirty_last/1` are synonyms.
- *mnesia:dirty_prev(Tab, Key)* works exactly like `mnesia:dirty_next/2` but returns the previous object in Erlang term order for the table type `ordered_set`. For all other table types, `mnesia:dirty_next/2` and `mnesia:dirty_prev/2` are synonyms.
- *mnesia:dirty_slot(Tab, Slot)* returns the list of records that are associated with `Slot` in a table. It can be used to traverse a table in a manner similar to the function `dirty_next/2`. A table has a number of slots

that range from zero to some unknown upper bound. The function `dirty_slot/2` returns the special atom `'$end_of_table'` when the end of the table is reached.

The behavior of this function is undefined if the table is written on while being traversed. The function *mnesia:read_lock_table(Tab)* can be used to ensure that no transaction-protected writes are performed during the iteration.

- *mnesia:dirty_update_counter({Tab,Key}, Val)*. Counters are positive integers with a value greater than or equal to zero. Updating a counter adds `Val` and the counter where `Val` is a positive or negative integer.

  `Mnesia` has no special counter records. However, records of the form `{TabName, Key, Integer}` can be used as counters, and can be persistent.

  Transaction-protected updates of counter records are not possible.

  There are two significant differences when using this function instead of reading the record, performing the arithmetic, and writing the record:

  - It is much more efficient.
  - The funcion *dirty_update_counter/2* is performed as an atomic operation although it is not protected by a transaction. Therfore no table update is lost if two processes simultaneously execute the function `dirty_update_counter/2`.
- *mnesia:dirty_match_object(Pat)* is the dirty equivalent of *mnesia:match_object/1*.
- *mnesia:dirty_select(Tab, Pat)* is the dirty equivalent of *mnesia:select/2*.
- *mnesia:dirty_index_match_object(Pat, Pos)* is the dirty equivalent of *mnesia:index_match_object/2*.
- *mnesia:dirty_index_read(Tab, SecondaryKey, Pos)* is the dirty equivalent of *mnesia:index_read/3*.
- *mnesia:dirty_all_keys(Tab)* is the dirty equivalent of *mnesia:all_keys/1*.

1.5.4 Record Names versus Table Names

In `Mnesia`, all records in a table must have the same name. All the records must be instances of the same record type. The record name, however, does not necessarily have to be the same as the table name, although this is the case in most of the examples in this User's Guide. If a table is created without property `record_name`, the following code ensures that all records in the tables have the same name as the table:

```
mnesia:create_table(subscriber, [])
```

However, if the table is created with an explicit record name as argument, as shown in the following example, subscriber records can be stored in both of the tables regardless of the table names:

```
TabDef = [{record_name, subscriber}],
mnesia:create_table(my_subscriber, TabDef),
mnesia:create_table(your_subscriber, TabDef).
```

To access such tables, simplified access functions (as described earlier) cannot be used. For example, writing a subscriber record into a table requires the function *mnesia:write/3* instead of the simplified functions *mnesia:write/1* and *mnesia:s_write/1*:

```
mnesia:write(subscriber, #subscriber{}, write)
mnesia:write(my_subscriber, #subscriber{}, sticky_write)
mnesia:write(your_subscriber, #subscriber{}, write)
```

The following simple code illustrates the relationship between the simplified access functions used in most of the examples and their more flexible counterparts:

```
mnesia:dirty_write(Record) ->
  Tab = element(1, Record),
  mnesia:dirty_write(Tab, Record).

mnesia:dirty_delete({Tab, Key}) ->
  mnesia:dirty_delete(Tab, Key).

mnesia:dirty_delete_object(Record) ->
  Tab = element(1, Record),
  mnesia:dirty_delete_object(Tab, Record)

mnesia:dirty_update_counter({Tab, Key}, Incr) ->
  mnesia:dirty_update_counter(Tab, Key, Incr).

mnesia:dirty_read({Tab, Key}) ->
  Tab = element(1, Record),
  mnesia:dirty_read(Tab, Key).

mnesia:dirty_match_object(Pattern) ->
  Tab = element(1, Pattern),
  mnesia:dirty_match_object(Tab, Pattern).

mnesia:dirty_index_match_object(Pattern, Attr)
  Tab = element(1, Pattern),
  mnesia:dirty_index_match_object(Tab, Pattern, Attr).

mnesia:write(Record) ->
  Tab = element(1, Record),
  mnesia:write(Tab, Record, write).

mnesia:s_write(Record) ->
  Tab = element(1, Record),
  mnesia:write(Tab, Record, sticky_write).

mnesia:delete({Tab, Key}) ->
  mnesia:delete(Tab, Key, write).

mnesia:s_delete({Tab, Key}) ->
  mnesia:delete(Tab, Key, sticky_write).

mnesia:delete_object(Record) ->
  Tab = element(1, Record),
  mnesia:delete_object(Tab, Record, write).

mnesia:s_delete_object(Record) ->
  Tab = element(1, Record),
  mnesia:delete_object(Tab, Record, sticky_write).

mnesia:read({Tab, Key}) ->
  mnesia:read(Tab, Key, read).

mnesia:wread({Tab, Key}) ->
  mnesia:read(Tab, Key, write).

mnesia:match_object(Pattern) ->
  Tab = element(1, Pattern),
  mnesia:match_object(Tab, Pattern, read).

mnesia:index_match_object(Pattern, Attr) ->
  Tab = element(1, Pattern),
  mnesia:index_match_object(Tab, Pattern, Attr, read).
```

1.5.5 Activity Concept and Various Access Contexts

As previously described, a Functional Object (Fun) performing table access operations, as listed here, can be passed on as arguments to the function *mnesia:transaction/1,2,3*:

- *mnesia:write/3 (write/1, s_write/1)*
- *mnesia:delete/3* (*mnesia:delete/1*, *mnesia:s_delete/1*)
- *mnesia:delete_object/3* (*mnesia:delete_object/1*, *mnesia:s_delete_object/1*)
- *mnesia:read/3* (*mnesia:read/1*, *mnesia:wread/1*)
- *mnesia:match_object/2* (*mnesia:match_object/1*)
- *mnesia:select/3* (*mnesia:select/2*)
- *mnesia:foldl/3* (`mnesia:foldl/4`, *mnesia:foldr/3*, `mnesia:foldr/4`)
- *mnesia:all_keys/1*
- *mnesia:index_match_object/4* (*mnesia:index_match_object/2*)
- *mnesia:index_read/3*
- *mnesia:lock/2* (*mnesia:read_lock_table/1*, *mnesia:write_lock_table/1*)
- *mnesia:table_info/2*

These functions are performed in a transaction context involving mechanisms, such as locking, logging, replication, checkpoints, subscriptions, and commit protocols. However, the same function can also be evaluated in other activity contexts.

The following activity access contexts are currently supported:

- `transaction`
- `sync_transaction`
- `async_dirty`
- `sync_dirty`
- `ets`

By passing the same "fun" as argument to the function *mnesia:sync_transaction(Fun [, Args])* it is performed in synced transaction context. Synced transactions wait until all active replicas has committed the transaction (to disc) before returning from the `mnesia:sync_transaction` call. Using `sync_transaction` is useful in the following cases:

- When an application executes on several nodes and wants to be sure that the update is performed on the remote nodes before a remote process is spawned or a message is sent to a remote process.
- When a combining transaction writes with "dirty_reads", that is, the functions `dirty_match_object`, `dirty_read`, `dirty_index_read`, `dirty_select`, and so on.
- When an application performs frequent or voluminous updates that can overload `Mnesia` on other nodes.

By passing the same "fun" as argument to the function *mnesia:async_dirty(Fun [, Args])*, it is performed in dirty context. The function calls are mapped to the corresponding dirty functions. This still involves logging, replication, and subscriptions but no locking, local transaction storage, or commit protocols are involved. Checkpoint retainers are updated but updated "dirty". Thus, they are updated asynchronously. The functions wait for the operation to be performed on one node but not the others. If the table resides locally, no waiting occurs.

By passing the same "fun" as an argument to the function *mnesia:sync_dirty(Fun [, Args])*, it is performed in almost the same context as the function *mnesia:async_dirty/1,2*. The difference is that the operations are performed synchronously. The caller waits for the updates to be performed on all active replicas. Using `mnesia:sync_dirty/1,2` is useful in the following cases:

- When an application executes on several nodes and wants to be sure that the update is performed on the remote nodes before a remote process is spawned or a message is sent to a remote process.

- When an application performs frequent or voluminous updates that can overload `Mnesia` on the nodes.

To check if your code is executed within a transaction, use the function *mnesia:is_transaction/0*. It returns `true` when called inside a transaction context, otherwise `false`.

`Mnesia` tables with storage type `RAM_copies` and `disc_copies` are implemented internally as `ets` tables. Applications can access the these tables directly. This is only recommended if all options have been weighed and the possible outcomes are understood. By passing the earlier mentioned "fun" to the function *mnesia:ets(Fun [, Args])*, it is performed but in a raw context. The operations are performed directly on the local `ets` tables, assuming that the local storage type is `RAM_copies` and that the table is not replicated on other nodes.

Subscriptions are not triggered and no checkpoints are updated, but this operation is blindingly fast. Disc resident tables are not to be updated with the `ets` function, as the disc is not updated.

The Fun can also be passed as an argument to the function *mnesia:activity/2,3,4*, which enables use of customized activity access callback modules. It can either be obtained directly by stating the module name as argument, or implicitly by use of configuration parameter `access_module`. A customized callback module can be used for several purposes, such as providing triggers, integrity constraints, runtime statistics, or virtual tables.

The callback module does not have to access real `Mnesia` tables, it is free to do whatever it wants as long as the callback interface is fulfilled.

Appendix B, Activity Access Callback Interface provides the source code, `mnesia_frag.erl`, for one alternative implementation. The context-sensitive function *mnesia:table_info/2* can be used to provide virtual information about a table. One use of this is to perform `QLC` queries within an activity context with a customized callback module. By providing table information about table indexes and other `QLC` requirements, `QLC` can be used as a generic query language to access virtual tables.

QLC queries can be performed in all these activity contexts (`transaction`, `sync_transaction`, `async_dirty`, `sync_dirty`, and `ets`). The `ets` activity only works if the table has no indexes.

Note:

The function `mnesia:dirty_*` always executes with `async_dirty` semantics regardless of which activity access contexts that are started. It can even start contexts without any enclosing activity access context.

1.5.6 Nested Transactions

Transactions can be nested in an arbitrary fashion. A child transaction must run in the same process as its parent. When a child transaction terminates, the caller of the child transaction gets return value `{aborted, Reason}` and any work performed by the child is erased. If a child transaction commits, the records written by the child are propagated to the parent.

No locks are released when child transactions terminate. Locks created by a sequence of nested transactions are kept until the topmost transaction terminates. Furthermore, any update performed by a nested transaction is only propagated in such a manner so that the parent of the nested transaction sees the updates. No final commitment is done until the top-level transaction terminates. So, although a nested transaction returns `{atomic, Val}`, if the enclosing parent transaction terminates, the entire nested operation terminates.

The ability to have nested transaction with identical semantics as top-level transaction makes it easier to write library functions that manipulate `Mnesia` tables.

Consider a function that adds a subscriber to a telephony system:

```
add_subscriber(S) ->
    mnesia:transaction(fun() ->
        case mnesia:read( ..........
```

This function needs to be called as a transaction. Assume that you wish to write a function that both calls the function `add_subscriber/1` and is in itself protected by the context of a transaction. By calling `add_subscriber/1` from within another transaction, a nested transaction is created.

Also, different activity access contexts can be mixed while nesting. However, the dirty ones (`async_dirty`, `sync_dirty`, and `ets`) inherit the transaction semantics if they are called inside a transaction and thus grab locks and use two or three phase commit.

Example:

```
add_subscriber(S) ->
    mnesia:transaction(fun() ->
       %% Transaction context
       mnesia:read({some_tab, some_data}),
       mnesia:sync_dirty(fun() ->
           %% Still in a transaction context.
           case mnesia:read( ..) ..end), end).
add_subscriber2(S) ->
    mnesia:sync_dirty(fun() ->
       %% In dirty context
       mnesia:read({some_tab, some_data}),
       mnesia:transaction(fun() ->
           %% In a transaction context.
           case mnesia:read( ..) ..end), end).
```

1.5.7 Pattern Matching

When the function *mnesia:read/3* cannot be used, `Mnesia` provides the programmer with several functions for matching records against a pattern. The most useful ones are the following:

```
mnesia:select(Tab, MatchSpecification, LockKind) ->
    transaction abort | [ObjectList]
mnesia:select(Tab, MatchSpecification, NObjects, Lock) ->
    transaction abort | {[Object],Continuation} | '$end_of_table'
mnesia:select(Cont) ->
    transaction abort | {[Object],Continuation} | '$end_of_table'
mnesia:match_object(Tab, Pattern, LockKind) ->
    transaction abort | RecordList
```

These functions match a `Pattern` against all records in table `Tab`. In a *mnesia:select* call, `Pattern` is a part of `MatchSpecification` described in the following. It is not necessarily performed as an exhaustive search of the entire table. By using indexes and bound values in the key of the pattern, the actual work done by the function can be condensed into a few hash lookups. Using `ordered_set` tables can reduce the search space if the keys are partially bound.

The pattern provided to the functions must be a valid record, and the first element of the provided tuple must be the `record_name` of the table. The special element `'_'` matches any data structure in Erlang (also known as an Erlang term). The special elements `'$<number>'` behave as Erlang variables, that is, they match anything, bind the first occurrence, and match the coming occurrences of that variable against the bound value.

Use function *mnesia:table_info(Tab, wild_pattern)* to obtain a basic pattern, which matches all records in a table, or use the default value in record creation. Do not make the pattern hard-coded, as this makes the code more vulnerable to future changes of the record definition.

Example:

```
Wildpattern = mnesia:table_info(employee, wild_pattern),
%% Or use
Wildpattern = #employee{_ = '_'},
```

For the employee table, the wild pattern looks as follows:

```
{employee, '_', '_', '_', '_', '_',' _'}.
```

To constrain the match, it is needed to replace some of the `'_'` elements. The code for matching out all female employees looks as follows:

```
Pat = #employee{sex = female, _ = '_'},
F = fun() -> mnesia:match_object(Pat) end,
Females = mnesia:transaction(F).
```

The match function can also be used to check the equality of different attributes. For example, to find all employees with an employee number equal to their room number:

```
Pat = #employee{emp_no = '$1', room_no = '$1', _ = '_'},
F = fun() -> mnesia:match_object(Pat) end,
Odd = mnesia:transaction(F).
```

The function *mnesia:match_object/3* lacks some important features that *mnesia:select/3* have. For example, `mnesia:match_object/3` can only return the matching records, and it cannot express constraints other than equality. To find the names of the male employees on the second floor:

```
MatchHead = #employee{name='$1', sex=male, room_no={'$2', '_'}, _='_'},
Guard = [{'>=', '$2', 220},{'<', '$2', 230}],
Result = '$1',
mnesia:select(employee,[{MatchHead, Guard, [Result]}])
```

The function `select` can be used to add more constraints and create output that cannot be done with `mnesia:match_object/3`.

The second argument to `select` is a `MatchSpecification`. A `MatchSpecification` is a list of `MatchFunctions`, where each `MatchFunction` consists of a tuple containing `{MatchHead, MatchCondition, MatchBody}`:

- `MatchHead` is the same pattern as used in `mnesia:match_object/3` described earlier.
- `MatchCondition` is a list of extra constraints applied to each record.
- `MatchBody` constructs the return values.

For details about the match specifications, see "Match Specifications in Erlang" in *ERTS* User's Guide. For more information, see the *ets* and *dets* manual pages in `STDLIB`.

The functions *select/4* and *select/1* are used to get a limited number of results, where `Continuation` gets the next chunk of results. `Mnesia` uses `NObjects` as a recommendation only. Thus, more or less results than specified with `NObjects` can be returned in the result list, even the empty list can be returned even if there are more results to collect.

Warning:

There is a severe performance penalty in using `mnesia:select/[1|2|3|4]` after any modifying operation is done on that table in the same transaction. That is, avoid using *mnesia:write/1* or *mnesia:delete/1* before `mnesia:select` in the same transaction.

If the key attribute is bound in a pattern, the match operation is efficient. However, if the key attribute in a pattern is given as `'_'` or `'$1'`, the whole `employee` table must be searched for records that match. Hence if the table is large, this can become a time-consuming operation, but it can be remedied with indexes (see *Indexing*) if the function *mnesia:match_object* is used.

QLC queries can also be used to search `Mnesia` tables. By using the function *mnesia:table/[1|2]* as the generator inside a QLC query, you let the query operate on a `Mnesia` table. `Mnesia`-specific options to `mnesia:table/2` are `{lock, Lock}`, `{n_objects,Integer}`, and `{traverse, SelMethod}`:

- `lock` specifies whether `Mnesia` is to acquire a read or write lock on the table.
- `n_objects` specifies how many results are to be returned in each chunk to QLC.
- `traverse` specifies which function `Mnesia` is to use to traverse the table. Default `select` is used, but by using `{traverse, {select, MatchSpecification}}` as an option to *mnesia:table/2* the user can specify its own view of the table.

If no options are specified, a read lock is acquired, 100 results are returned in each chunk, and `select` is used to traverse the table, that is:

```
mnesia:table(Tab) ->
    mnesia:table(Tab, [{n_objects,100},{lock, read}, {traverse, select}]).
```

The function *mnesia:all_keys(Tab)* returns all keys in a table.

1.5.8 Iteration

`Mnesia` provides the following functions that iterate over all the records in a table:

```
mnesia:foldl(Fun, Acc0, Tab) -> NewAcc | transaction abort
mnesia:foldr(Fun, Acc0, Tab) -> NewAcc | transaction abort
mnesia:foldl(Fun, Acc0, Tab, LockType) -> NewAcc | transaction abort
mnesia:foldr(Fun, Acc0, Tab, LockType) -> NewAcc | transaction abort
```

These functions iterate over the `Mnesia` table `Tab` and apply the function `Fun` to each record. `Fun` takes two arguments, the first is a record from the table, and the second is the accumulator. `Fun` returns a new accumulator.

The first time `Fun` is applied, `Acc0` is the second argument. The next time `Fun` is called, the return value from the previous call is used as the second argument. The term the last call to `Fun` returns is the return value of the function *mnesia:foldl/3* or *mnesia:foldr/3*.

The difference between these functions is the order the table is accessed for `ordered_set` tables. For other table types the functions are equivalent.

`LockType` specifies what type of lock that is to be acquired for the iteration, default is `read`. If records are written or deleted during the iteration, a write lock is to be acquired.

These functions can be used to find records in a table when it is impossible to write constraints for the function *mnesia:match_object/3*, or when you want to perform some action on certain records.

For example, finding all the employees who have a salary less than 10 can look as follows:

```
find_low_salaries() ->
  Constraint =
       fun(Emp, Acc) when Emp#employee.salary < 10 ->
              [Emp | Acc];
          (_, Acc) ->
              Acc
       end,
  Find = fun() -> mnesia:foldl(Constraint, [], employee) end,
  mnesia:transaction(Find).
```

To raise the salary to 10 for everyone with a salary less than 10 and return the sum of all raises:

```
increase_low_salaries() ->
   Increase =
       fun(Emp, Acc) when Emp#employee.salary < 10 ->
              OldS = Emp#employee.salary,
              ok = mnesia:write(Emp#employee{salary = 10}),
              Acc + 10 - OldS;
          (_, Acc) ->
              Acc
       end,
  IncLow = fun() -> mnesia:foldl(Increase, 0, employee, write) end,
  mnesia:transaction(IncLow).
```

Many nice things can be done with the iterator functions but take some caution about performance and memory use for large tables.

Call these iteration functions on nodes that contain a replica of the table. Each call to the function `Fun` access the table and if the table resides on another node it generates much unnecessary network traffic.

`Mnesia` also provides some functions that make it possible for the user to iterate over the table. The order of the iteration is unspecified if the table is not of type `ordered_set`:

```
mnesia:first(Tab) ->  Key | transaction abort
mnesia:last(Tab)  ->  Key | transaction abort
mnesia:next(Tab,Key)  ->  Key | transaction abort
mnesia:prev(Tab,Key)  ->  Key | transaction abort
mnesia:snmp_get_next_index(Tab,Index) -> {ok, NextIndex} | endOfTable
```

The order of `first`/`last` and `next`/`prev` is only valid for `ordered_set` tables, they are synonyms for other tables. When the end of the table is reached, the special key `'$end_of_table'` is returned.

If records are written and deleted during the traversal, use the function *mnesia:foldl/3* or *mnesia:foldr/3* with a `write` lock. Or the function *mnesia:write_lock_table/1* when using `first` and `next`.

Writing or deleting in transaction context creates a local copy of each modified record. Thus, modifying each record in a large table uses much memory. `Mnesia` compensates for every written or deleted record during the iteration in a transaction context, which can reduce the performance. If possible, avoid writing or deleting records in the same transaction before iterating over the table.

In dirty context, that is, `sync_dirty` or `async_dirty`, the modified records are not stored in a local copy; instead, each record is updated separately. This generates much network traffic if the table has a replica on another node and has all the other drawbacks that dirty operations have. Especially for commands *mnesia:first/1* and *mnesia:next/2*, the same drawbacks as described previously for *mnesia:dirty_first/1* and *mnesia:dirty_next/2* applies, that is, no writing to the table is to be done during iteration.

1.6 Miscellaneous Mnesia Features

The previous sections describe how to get started with `Mnesia` and how to build a `Mnesia` database. This section describes the more advanced features available when building a distributed, fault-tolerant `Mnesia` database. The following topics are included:

- Indexing
- Distribution and fault tolerance
- Table fragmentation
- Local content tables
- Disc-less nodes
- More about schema management
- `Mnesia` event handling

- Debugging `Mnesia` applications
- Concurrent processes in `Mnesia`
- Prototyping
- Object-based programming with `Mnesia`

1.6.1 Indexing

Data retrieval and matching can be performed efficiently if the key for the record is known. Conversely, if the key is unknown, all records in a table must be searched. The larger the table, the more time consuming it becomes. To remedy this problem, `Mnesia` indexing capabilities are used to improve data retrieval and matching of records.

The following two functions manipulate indexes on existing tables:

- *mnesia:add_table_index(Tab, AttributeName) -> {aborted, R} |{atomic, ok}*
- *mnesia:del_table_index(Tab, AttributeName) -> {aborted, R} |{atomic, ok}*

These functions create or delete a table index on a field defined by `AttributeName`. To illustrate this, add an index to the table definition `(employee, {emp_no, name, salary, sex, phone, room_no})`, which is the example table from the `Company` database. The function that adds an index on element `salary` can be expressed as `mnesia:add_table_index(employee, salary)`.

The indexing capabilities of `Mnesia` are used with the following three functions, which retrieve and match records based on index entries in the database:

- *mnesia:index_read(Tab, SecondaryKey, AttributeName) -> transaction abort | RecordList* avoids an exhaustive search of the entire table, by looking up `SecondaryKey` in the index to find the primary keys.
- *mnesia:index_match_object(Pattern, AttributeName) -> transaction abort | RecordList* avoids an exhaustive search of the entire table, by looking up the secondary key in the index to find the primary keys. The secondary key is found in field `AttributeName` of `Pattern`. The secondary key must be bound.
- *mnesia:match_object(Pattern) -> transaction abort | RecordList* uses indexes to avoid exhaustive search of the entire table. Unlike the previous functions, this function can use any index as long as the secondary key is bound.

These functions are further described and exemplified in *Pattern Matching*.

1.6.2 Distribution and Fault Tolerance

`Mnesia` is a distributed, fault-tolerant DBMS. Tables can be replicated on different Erlang nodes in various ways. The `Mnesia` programmer does not need to state where the different tables reside, only the names of the different tables need to be specified in the program code. This is known as "location transparency" and is an important concept. In particular:

- A program works regardless of the data location. It makes no difference whether the data resides on the local node or on a remote node.

  Notice that the program runs slower if the data is located on a remote node.
- The database can be reconfigured, and tables can be moved between nodes. These operations do not affect the user programs.

It has previously been shown that each table has a number of system attributes, such as `index` and `type`.

Table attributes are specified when the table is created. For example, the following function creates a table with two RAM replicas:

```
mnesia:create_table(foo,
                    [{ram_copies, [N1, N2]},
                     {attributes, record_info(fields, foo)}]).
```

Tables can also have the following properties, where each attribute has a list of Erlang nodes as its value:

- `ram_copies`. The value of the node list is a list of Erlang nodes, and a RAM replica of the table resides on each node in the list.

  Notice that no disc operations are performed when a program executes write operations to these replicas. However, if permanent RAM replicas are required, the following alternatives are available:

  - The function *mnesia:dump_tables/1* can be used to dump RAM table replicas to disc.
  - The table replicas can be backed up, either from RAM, or from disc if dumped there with this function.
- `disc_copies`. The value of the attribute is a list of Erlang nodes, and a replica of the table resides both in RAM and on disc on each node in the list. Write operations addressed to the table address both the RAM and the disc copy of the table.
- `disc_only_copies`. The value of the attribute is a list of Erlang nodes, and a replica of the table resides only as a disc copy on each node in the list. The major disadvantage of this type of table replica is the access speed. The major advantage is that the table does not occupy space in memory.

In addition, table properties can be set and changed. For details, see *Define a Schema*.

There are basically two reasons for using more than one table replica: fault tolerance and speed. Notice that table replication provides a solution to both of these system requirements.

If there are two active table replicas, all information is still available if one replica fails. This can be an important property in many applications. Furthermore, if a table replica exists at two specific nodes, applications that execute at either of these nodes can read data from the table without accessing the network. Network operations are considerably slower and consume more resources than local operations.

It can be advantageous to create table replicas for a distributed application that reads data often, but writes data seldom, to achieve fast read operations on the local node. The major disadvantage with replication is the increased time to write data. If a table has two replicas, every write operation must access both table replicas. Since one of these write operations must be a network operation, it is considerably more expensive to perform a write operation to a replicated table than to a non-replicated table.

1.6.3 Table Fragmentation

Concept

A concept of table fragmentation has been introduced to cope with large tables. The idea is to split a table into several manageable fragments. Each fragment is implemented as a first class `Mnesia` table and can be replicated, have indexes, and so on, as any other table. But the tables cannot have `local_content` or have the `snmp` connection activated.

To be able to access a record in a fragmented table, `Mnesia` must determine to which fragment the actual record belongs. This is done by module `mnesia_frag`, which implements the `mnesia_access` callback behavior. It is recommended to read the documentation about the function *mnesia:activity/4* to see how `mnesia_frag` can be used as a `mnesia_access` callback module.

At each record access, `mnesia_frag` first computes a hash value from the record key. Second, the name of the table fragment is determined from the hash value. Finally the actual table access is performed by the same functions as for non-fragmented tables. When the key is not known beforehand, all fragments are searched for matching records.

Notice that in `ordered_set` tables, the records are ordered per fragment, and the order is undefined in results returned by `select` and `match_object`, as well as `first`, `next`, `prev` and `last`.

The following code illustrates how a `Mnesia` table is converted to be a fragmented table and how more fragments are added later:

```
Eshell V4.7.3.3  (abort with ^G)
(a@sam)1> mnesia:start().
ok
(a@sam)2> mnesia:system_info(running_db_nodes).
[b@sam,c@sam,a@sam]
(a@sam)3> Tab = dictionary.
dictionary
(a@sam)4> mnesia:create_table(Tab, [{ram_copies, [a@sam, b@sam]}]).
{atomic,ok}
(a@sam)5> Write = fun(Keys) -> [mnesia:write({Tab,K,-K}) || K <- Keys], ok end.
#Fun<erl_eval>
(a@sam)6> mnesia:activity(sync_dirty, Write, [lists:seq(1, 256)], mnesia_frag).
ok
(a@sam)7> mnesia:change_table_frag(Tab, {activate, []}).
{atomic,ok}
(a@sam)8> mnesia:table_info(Tab, frag_properties).
[{base_table,dictionary},
 {foreign_key,undefined},
 {n_doubles,0},
 {n_fragments,1},
 {next_n_to_split,1},
 {node_pool,[a@sam,b@sam,c@sam]}]
(a@sam)9> Info = fun(Item) -> mnesia:table_info(Tab, Item) end.
#Fun<erl_eval>
(a@sam)10> Dist = mnesia:activity(sync_dirty, Info, [frag_dist], mnesia_frag).
[{c@sam,0},{a@sam,1},{b@sam,1}]
(a@sam)11> mnesia:change_table_frag(Tab, {add_frag, Dist}).
{atomic,ok}
(a@sam)12> Dist2 = mnesia:activity(sync_dirty, Info, [frag_dist], mnesia_frag).
[{b@sam,1},{c@sam,1},{a@sam,2}]
(a@sam)13> mnesia:change_table_frag(Tab, {add_frag, Dist2}).
{atomic,ok}
(a@sam)14> Dist3 = mnesia:activity(sync_dirty, Info, [frag_dist], mnesia_frag).
[{a@sam,2},{b@sam,2},{c@sam,2}]
(a@sam)15> mnesia:change_table_frag(Tab, {add_frag, Dist3}).
{atomic,ok}
(a@sam)16> Read = fun(Key) -> mnesia:read({Tab, Key}) end.
#Fun<erl_eval>
(a@sam)17> mnesia:activity(transaction, Read, [12], mnesia_frag).
[{dictionary,12,-12}]
(a@sam)18> mnesia:activity(sync_dirty, Info, [frag_size], mnesia_frag).
[{dictionary,64},
 {dictionary_frag2,64},
 {dictionary_frag3,64},
 {dictionary_frag4,64}]
(a@sam)19>
```

Fragmentation Properties

The table property `frag_properties` can be read with the function *mnesia:table_info(Tab, frag_properties)*. The fragmentation properties are a list of tagged tuples with arity 2. By default the list is empty, but when it is non-empty it triggers `Mnesia` to regard the table as fragmented. The fragmentation properties are as follows:

`{n_fragments, Int}`

`n_fragments` regulates how many fragments that the table currently has. This property can explicitly be set at table creation and later be changed with `{add_frag, NodesOrDist}` or `del_frag`. `n_fragments` defaults to `1`.

`{node_pool, List}`

The node pool contains a list of nodes and can explicitly be set at table creation and later be changed with `{add_node, Node}` or `{del_node, Node}`. At table creation `Mnesia` tries to distribute the replicas of

each fragment evenly over all the nodes in the node pool. Hopefully all nodes end up with the same number of replicas. `node_pool` defaults to the return value from the function *mnesia:system_info(db_nodes)*.

`{n_ram_copies, Int}`

Regulates how many `ram_copies` replicas that each fragment is to have. This property can explicitly be set at table creation. Defaults is `0`, but if `n_disc_copies` and `n_disc_only_copies` also are `0`, `n_ram_copies` defaults to `1`.

`{n_disc_copies, Int}`

Regulates how many `disc_copies` replicas that each fragment is to have. This property can explicitly be set at table creation. Default is `0`.

`{n_disc_only_copies, Int}`

Regulates how many `disc_only_copies` replicas that each fragment is to have. This property can explicitly be set at table creation. Defaults is `0`.

`{foreign_key, ForeignKey}`

`ForeignKey` can either be the atom `undefined` or the tuple `{ForeignTab, Attr}`, where `Attr` denotes an attribute that is to be interpreted as a key in another fragmented table named `ForeignTab`. `Mnesia` ensures that the number of fragments in this table and in the foreign table are always the same.

When fragments are added or deleted, `Mnesia` automatically propagates the operation to all fragmented tables that have a foreign key referring to this table. Instead of using the record key to determine which fragment to access, the value of field `Attr` is used. This feature makes it possible to colocate records automatically in different tables to the same node. `foreign_key` defaults to `undefined`. However, if the foreign key is set to something else, it causes the default values of the other fragmentation properties to be the same values as the actual fragmentation properties of the foreign table.

`{hash_module, Atom}`

Enables definition of an alternative hashing scheme. The module must implement the *mnesia_frag_hash* callback behavior. This property can explicitly be set at table creation. Default is `mnesia_frag_hash`.

`{hash_state, Term}`

Enables a table-specific parameterization of a generic hash module. This property can explicitly be set at table creation. Default is `undefined`.

```
Eshell V4.7.3.3  (abort with ^G)
(a@sam)1> mnesia:start().
ok
(a@sam)2> PrimProps = [{n_fragments, 7}, {node_pool, [node()]}].
[{n_fragments,7},{node_pool,[a@sam]}]
(a@sam)3> mnesia:create_table(prim_dict,
                              [{frag_properties, PrimProps},
                               {attributes,[prim_key,prim_val]}]).
{atomic,ok}
(a@sam)4> SecProps = [{foreign_key, {prim_dict, sec_val}}].
[{foreign_key,{prim_dict,sec_val}}]
(a@sam)5> mnesia:create_table(sec_dict,
                              [{frag_properties, SecProps},
(a@sam)5>                      {attributes, [sec_key, sec_val]}]).
{atomic,ok}
(a@sam)6> Write = fun(Rec) -> mnesia:write(Rec) end.
#Fun<erl_eval>
(a@sam)7> PrimKey = 11.
11
(a@sam)8> SecKey = 42.
42
(a@sam)9> mnesia:activity(sync_dirty, Write,
                          [{prim_dict, PrimKey, -11}], mnesia_frag).
ok
(a@sam)10> mnesia:activity(sync_dirty, Write,
                           [{sec_dict, SecKey, PrimKey}], mnesia_frag).
ok
(a@sam)11> mnesia:change_table_frag(prim_dict, {add_frag, [node()]}).
{atomic,ok}
(a@sam)12> SecRead = fun(PrimKey, SecKey) ->
              mnesia:read({sec_dict, PrimKey}, SecKey, read) end.
#Fun<erl_eval>
(a@sam)13> mnesia:activity(transaction, SecRead,
                           [PrimKey, SecKey], mnesia_frag).
[{sec_dict,42,11}]
(a@sam)14> Info = fun(Tab, Item) -> mnesia:table_info(Tab, Item) end.
#Fun<erl_eval>
(a@sam)15> mnesia:activity(sync_dirty, Info,
                           [prim_dict, frag_size], mnesia_frag).
[{prim_dict,0},
 {prim_dict_frag2,0},
 {prim_dict_frag3,0},
 {prim_dict_frag4,1},
 {prim_dict_frag5,0},
 {prim_dict_frag6,0},
 {prim_dict_frag7,0},
 {prim_dict_frag8,0}]
(a@sam)16> mnesia:activity(sync_dirty, Info,
                           [sec_dict, frag_size], mnesia_frag).
[{sec_dict,0},
 {sec_dict_frag2,0},
 {sec_dict_frag3,0},
 {sec_dict_frag4,1},
 {sec_dict_frag5,0},
 {sec_dict_frag6,0},
 {sec_dict_frag7,0},
 {sec_dict_frag8,0}]
(a@sam)17>
```

Management of Fragmented Tables

The function `mnesia:change_table_frag(Tab, Change)` is intended to be used for reconfiguration of fragmented tables. Argument `Change` is to have one of the following values:

`{activate, FragProps}`

Activates the fragmentation properties of an existing table. `FragProps` is either to contain `{node_pool, Nodes}` or be empty.

`deactivate`

Deactivates the fragmentation properties of a table. The number of fragments must be `1`. No other table can refer to this table in its foreign key.

`{add_frag, NodesOrDist}`

Adds a fragment to a fragmented table. All records in one of the old fragments are rehashed and about half of them are moved to the new (last) fragment. All other fragmented tables, which refer to this table in their foreign key, automatically get a new fragment. Also, their records are dynamically rehashed in the same manner as for the main table.

Argument `NodesOrDist` can either be a list of nodes or the result from the function *mnesia:table_info(Tab, frag_dist)*. Argument `NodesOrDist` is assumed to be a sorted list with the best nodes to host new replicas first in the list. The new fragment gets the same number of replicas as the first fragment (see `n_ram_copies`, `n_disc_copies`, and `n_disc_only_copies`). The `NodesOrDist` list must at least contain one element for each replica that needs to be allocated.

`del_frag`

Deletes a fragment from a fragmented table. All records in the last fragment are moved to one of the other fragments. All other fragmented tables, which refer to this table in their foreign key, automatically lose their last fragment. Also, their records are dynamically rehashed in the same manner as for the main table.

`{add_node, Node}`

Adds a node to `node_pool`. The new node pool affects the list returned from the function *mnesia:table_info(Tab, frag_dist)*.

`{del_node, Node}`

Deletes a node from `node_pool`. The new node pool affects the list returned from the function *mnesia:table_info(Tab, frag_dist)*.

Extensions of Existing Functions

The function *mnesia:create_table/2* creates a brand new fragmented table, by setting table property `frag_properties` to some proper values.

The function *mnesia:delete_table/1* deletes a fragmented table including all its fragments. There must however not exist any other fragmented tables that refer to this table in their foreign key.

The function *mnesia:table_info/2* now understands item `frag_properties`.

If the function `mnesia:table_info/2` is started in the activity context of module `mnesia_frag`, information of several new items can be obtained:

`base_table`
The name of the fragmented table

`n_fragments`
The actual number of fragments

`node_pool`
The pool of nodes

`n_ram_copies`

`n_disc_copies`

`n_disc_only_copies`

The number of replicas with storage type `ram_copies`, `disc_copies`, and `disc_only_copies`, respectively. The actual values are dynamically derived from the first fragment. The first fragment serves as a protype. When the actual values need to be computed (for example, when adding new fragments) they are determined by counting the number of each replica for each storage type. This means that when the functions *mnesia:add_table_copy/3*, *mnesia:del_table_copy/2*, and *mnesia:change_table_copy_type/2* are applied on the first fragment, it affects the settings on `n_ram_copies`, `n_disc_copies`, and `n_disc_only_copies`.

`foreign_key`

The foreign key

`foreigners`

All other tables that refer to this table in their foreign key

`frag_names`

The names of all fragments

`frag_dist`

A sorted list of `{Node, Count}` tuples that are sorted in increasing `Count` order. `Count` is the total number of replicas that this fragmented table hosts on each `Node`. The list always contains at least all nodes in `node_pool`. Nodes that do not belong to `node_pool` are put last in the list even if their `Count` is lower.

`frag_size`

A list of `{Name, Size}` tuples, where `Name` is a fragment `Name`, and `Size` is how many records it contains

`frag_memory`

A list of `{Name, Memory}` tuples, where `Name` is a fragment `Name`, and `Memory` is how much memory it occupies

`size`

Total size of all fragments

`memory`

Total memory of all fragments

Load Balancing

There are several algorithms for distributing records in a fragmented table evenly over a pool of nodes. No one is best, it depends on the application needs. The following examples of situations need some attention:

- `permanent change of nodes`. When a new permanent `db_node` is introduced or dropped, it can be time to change the pool of nodes and redistribute the replicas evenly over the new pool of nodes. It can also be time to add or delete a fragment before the replicas are redistributed.
- `size/memory threshold`. When the total size or total memory of a fragmented table (or a single fragment) exceeds some application-specific threshold, it can be time to add a new fragment dynamically to obtain a better distribution of records.
- `temporary node down`. When a node temporarily goes down, it can be time to compensate some fragments with new replicas to keep the desired level of redundancy. When the node comes up again, it can be time to remove the superfluous replica.
- `overload threshold`. When the load on some node exceeds some application-specific threshold, it can be time to either add or move some fragment replicas to nodes with lower load. Take extra care if the table has a foreign key relation to some other table. To avoid severe performance penalties, the same redistribution must be performed for all the related tables.

Use the function `mnesia:change_table_frag/2` to add new fragments and apply the usual schema manipulation functions (such as *mnesia:add_table_copy/3*, *mnesia:del_table_copy/2*, and *mnesia:change_table_copy_type/2*) on each fragment to perform the actual redistribution.

1.6.4 Local Content Tables

Replicated tables have the same content on all nodes where they are replicated. However, it is sometimes advantageous to have tables, but different content on different nodes.

If attribute `{local_content, true}` is specified when you create the table, the table resides on the nodes where you specify the table to exist, but the write operations on the table are only performed on the local copy.

Furthermore, when the table is initialized at startup, the table is only initialized locally, and the table content is not copied from another node.

1.6.5 Disc-Less Nodes

`Mnesia` can be run on nodes that do not have a disc. Replicas of `disc_copies` or `disc_only_copies` are not possible on such nodes. This is especially troublesome for the `schema` table, as `Mnesia` needs the schema to initialize itself.

The schema table can, as other tables, reside on one or more nodes. The storage type of the schema table can either be `disc_copies` or `ram_copies` (but not `disc_only_copies`). At startup, `Mnesia` uses its schema to determine with which nodes it is to try to establish contact. If any other node is started already, the starting node merges its table definitions with the table definitions brought from the other nodes. This also applies to the definition of the schema table itself. Application parameter `extra_db_nodes` contains a list of nodes that `Mnesia` also is to establish contact with besides those found in the schema. Default is `[]` (empty list).

Hence, when a disc-less node needs to find the schema definitions from a remote node on the network, this information must be supplied through application parameter `-mnesia extra_db_nodes NodeList`. Without this configuration parameter set, `Mnesia` starts as a single node system. Also, the function *mnesia:change_config/2* can be used to assign a value to `extra_db_nodes` and force a connection after `Mnesia` has been started, that is, `mnesia:change_config(extra_db_nodes, NodeList)`.

Application parameter `schema_location` controls where `Mnesia` searches for its schema. The parameter can be one of the following atoms:

`disc`

> Mandatory disc. The schema is assumed to be located in the `Mnesia` directory. If the schema cannot be found, `Mnesia` refuses to start.

`ram`

> Mandatory RAM. The schema resides in RAM only. At startup, a tiny new schema is generated. This default schema contains only the definition of the schema table and resides on the local node only. Since no other nodes are found in the default schema, configuration parameter `extra_db_nodes` must be used to let the node share its table definitions with other nodes. (Parameter `extra_db_nodes` can also be used on disc-full nodes.)

`opt_disc`

> Optional disc. The schema can reside on either disc or RAM. If the schema is found on disc, `Mnesia` starts as a disc-full node (the storage type of the schema table is disc_copies). If no schema is found on disc, `Mnesia` starts as a disc-less node (the storage type of the schema table is `ram_copies`). The default for the application parameter is `opt_disc`.

When `schema_location` is set to `opt_disc`, the function *mnesia:change_table_copy_type/3* can be used to change the storage type of the schema. This is illustrated as follows:

```
1> mnesia:start().
ok
2> mnesia:change_table_copy_type(schema, node(), disc_copies).
{atomic, ok}
```

Assuming that the call to *mnesia:start/0* does not find any schema to read on the disc, `Mnesia` starts as a disc-less node, and then change it to a node that use the disc to store the schema locally.

1.6.6 More about Schema Management

Nodes can be added to and removed from a `Mnesia` system. This can be done by adding a copy of the schema to those nodes.

The functions *mnesia:add_table_copy/3* and *mnesia:del_table_copy/2* can be used to add and delete replicas of the schema table. Adding a node to the list of nodes where the schema is replicated affects the following:

- It allows other tables to be replicated to this node.
- It causes `Mnesia` to try to contact the node at startup of disc-full nodes.

The function call `mnesia:del_table_copy(schema, mynode@host)` deletes node `mynode@host` from the `Mnesia` system. The call fails if `Mnesia` is running on `mynode@host`. The other `Mnesia` nodes never try to connect to that node again. Notice that if there is a disc resident schema on node `mynode@host`, the entire `Mnesia` directory is to be deleted. This is done with the function *mnesia:delete_schema/1*. If `Mnesia` is started again on node `mynode@host` and the directory has not been cleared, the behavior of `Mnesia` is undefined.

If the storage type of the schema is `ram_copies`, that is, a disc-less node, `Mnesia` does not use the disc on that particular node. The disc use is enabled by changing the storage type of table `schema` to `disc_copies`.

New schemas are created explicitly with the function *mnesia:create_schema/1* or implicitly by starting `Mnesia` without a disc resident schema. Whenever a table (including the schema table) is created, it is assigned its own unique cookie. The schema table is not created with the function *mnesia:create_table/2* as normal tables.

At startup, `Mnesia` connects different nodes to each other, then they exchange table definitions with each other, and the table definitions are merged. During the merge procedure, `Mnesia` performs a sanity test to ensure that the table definitions are compatible with each other. If a table exists on several nodes, the cookie must be the same, otherwise `Mnesia` shut down one of the nodes. This unfortunate situation occurs if a table has been created on two nodes independently of each other while they were disconnected. To solve this, one of the tables must be deleted (as the cookies differ, it is regarded to be two different tables even if they have the same name).

Merging different versions of the schema table does not always require the cookies to be the same. If the storage type of the schema table is `disc_copies`, the cookie is immutable, and all other `db_nodes` must have the same cookie. When the schema is stored as type `ram_copies`, its cookie can be replaced with a cookie from another node (`ram_copies` or `disc_copies`). The cookie replacement (during merge of the schema table definition) is performed each time a RAM node connects to another node.

Further, the following applies:

- *mnesia:system_info(schema_location)* and *mnesia:system_info(extra_db_nodes)* can be used to determine the actual values of `schema_location` and `extra_db_nodes`, respectively.
- *mnesia:system_info(use_dir)* can be used to determine whether `Mnesia` is actually using the `Mnesia` directory.
- `use_dir` can be determined even before `Mnesia` is started.

The function *mnesia:info/0* can now be used to print some system information even before `Mnesia` is started. When `Mnesia` is started, the function prints more information.

Transactions that update the definition of a table requires that `Mnesia` is started on all nodes where the storage type of the schema is `disc_copies`. All replicas of the table on these nodes must also be loaded. There are a few exceptions to these availability rules:

- Tables can be created and new replicas can be added without starting all the disc-full nodes.
- New replicas can be added before all other replicas of the table have been loaded, provided that at least one other replica is active.

1.6.7 Mnesia Event Handling

System events and table events are the two event categories that `Mnesia` generates in various situations.

A user process can subscribe on the events generated by `Mnesia`. The following two functions are provided:

mnesia:subscribe(Event-Category)
: Ensures that a copy of all events of type `Event-Category` are sent to the calling process

mnesia:unsubscribe(Event-Category)
: Removes the subscription on events of type `Event-Category`

`Event-Category` can be either of the following:

- The atom `system`
- The atom `activity`
- The tuple `{table, Tab, simple}`
- The tuple `{table, Tab, detailed}`

The old event category `{table, Tab}` is the same event category as `{table, Tab, simple}`.

The subscribe functions activate a subscription of events. The events are delivered as messages to the process evaluating the function *mnesia:subscribe/1* The syntax is as follows:

- `{mnesia_system_event, Event}` for system events
- `{mnesia_activity_event, Event}` for activity events
- `{mnesia_table_event, Event}` for table events

The event types are described in the next sections.

All system events are subscribed by the `Mnesia gen_event` handler. The default `gen_event` handler is `mnesia_event`, but it can be changed by using application parameter `event_module`. The value of this parameter must be the name of a module implementing a complete handler, as specified by the *gen_event* module in `STDLIB`.

mnesia:system_info(subscribers) and *mnesia:table_info(Tab, subscribers)* can be used to determine which processes are subscribed to various events.

System Events

The system events are as follows:

`{mnesia_up, Node}`
: Mnesia is started on a node. `Node` is the node name. By default this event is ignored.

`{mnesia_down, Node}`
: Mnesia is stopped on a node. `Node` is the node name. By default this event is ignored.

`{mnesia_checkpoint_activated, Checkpoint}`
: A checkpoint with the name `Checkpoint` is activated and the current node is involved in the checkpoint. Checkpoints can be activated explicitly with the function *mnesia:activate_checkpoint/1* or implicitly at backup, when adding table replicas, at internal transfer of data between nodes, and so on. By default this event is ignored.

`{mnesia_checkpoint_deactivated, Checkpoint}`
A checkpoint with the name `Checkpoint` is deactivated and the current node is involved in the checkpoint. Checkpoints can be deactivated explicitly with the function *mnesia:deactivate/1* or implicitly when the last replica of a table (involved in the checkpoint) becomes unavailable, for example, at node-down. By default this event is ignored.

`{mnesia_overload, Details}`

`Mnesia` on the current node is overloaded and the subscriber is to take action.

A typical overload situation occurs when the applications perform more updates on disc resident tables than `Mnesia` can handle. Ignoring this kind of overload can lead to a situation where the disc space is exhausted (regardless of the size of the tables stored on disc).

Each update is appended to the transaction log and occasionally (depending on how it is configured) dumped to the tables files. The table file storage is more compact than the transaction log storage, especially if the same record is updated repeatedly. If the thresholds for dumping the transaction log are reached before the previous dump is finished, an overload event is triggered.

Another typical overload situation is when the transaction manager cannot commit transactions at the same pace as the applications perform updates of disc resident tables. When this occurs, the message queue of the transaction manager continues to grow until the memory is exhausted or the load decreases.

The same problem can occur for dirty updates. The overload is detected locally on the current node, but its cause can be on another node. Application processes can cause high load if any table resides on another node (replicated or not). By default this event is reported to `error_logger.`

`{inconsistent_database, Context, Node}`
`Mnesia` regards the database as potential inconsistent and gives its applications a chance to recover from the inconsistency. For example, by installing a consistent backup as fallback and then restart the system. An alternative is to pick a `MasterNode` from *mnesia:system_info(db_nodes)* and invoke *mnesia:set_master_node([MasterNode])*. By default an error is reported to `error_logger`.

`{mnesia_fatal, Format, Args, BinaryCore}`

`Mnesia` detected a fatal error and terminates soon. The fault reason is explained in `Format` and `Args`, which can be given as input to `io:format/2` or sent to `error_logger`. By default it is sent to `error_logger`.

`BinaryCore` is a binary containing a summary of the `Mnesia` internal state at the time when the fatal error was detected. By default the binary is written to a unique filename on the current directory. On RAM nodes, the core is ignored.

`{mnesia_info, Format, Args}`
`Mnesia` detected something that can be of interest when debugging the system. This is explained in `Format` and `Args`, which can appear as input to `io:format/2` or sent to `error_logger`. By default this event is printed with `io:format/2`.

`{mnesia_error, Format, Args}`
`Mnesia` has detected an error. The fault reason is explained in `Format` and `Args`, which can be given as input to `io:format/2` or sent to `error_logger`. By default this event is reported to `error_logger`.

`{mnesia_user, Event}`
An application started the function *mnesia:report_event(Event)*. `Event` can be any Erlang data structure. When tracing a system of `Mnesia` applications, it is useful to be able to interleave own events of `Mnesia` with application-related events that give information about the application context. Whenever the application starts with a new and demanding `Mnesia` activity, or enters a new and interesting phase in its execution, it can be a good idea to use `mnesia:report_event/1`.

Activity Events

Currently, there is only one type of activity event:

`{complete, ActivityID}`

This event occurs when a transaction that caused a modification to the database is completed. It is useful for determining when a set of table events (see the next section), caused by a given activity, have been sent. Once this event is received, it is guaranteed that no further table events with the same `ActivityID` will be received. Notice that this event can still be received even if no table events with a corresponding `ActivityID` were received, depending on the tables to which the receiving process is subscribed.

Dirty operations always contain only one update and thus no activity event is sent.

Table Events

Table events are events related to table updates. There are two types of table events, simple and detailed.

The **simple table events** are tuples like `{Oper, Record, ActivityId}`, where:

- `Oper` is the operation performed.
- `Record` is the record involved in the operation.
- `ActivityId` is the identity of the transaction performing the operation.

Notice that the record name is the table name even when `record_name` has another setting.

The table-related events that can occur are as follows:

`{write, NewRecord, ActivityId}`
A new record has been written. `NewRecord` contains the new record value.

`{delete_object, OldRecord, ActivityId}`
A record has possibly been deleted with *mnesia:delete_object/1*. `OldRecord` contains the value of the old record, as stated as argument by the application. Notice that other records with the same key can remain in the table if it is of type `bag`.

`{delete, {Tab, Key}, ActivityId}`
One or more records have possibly been deleted. All records with the key `Key` in the table `Tab` have been deleted.

The **detailed table events** are tuples like `{Oper, Table, Data, [OldRecs], ActivityId}`, where:

- `Oper` is the operation performed.
- `Table` is the table involved in the operation.
- `Data` is the record/OID written/deleted.
- `OldRecs` is the contents before the operation.
- `ActivityId` is the identity of the transaction performing the operation.

The table-related events that can occur are as follows:

`{write, Table, NewRecord, [OldRecords], ActivityId}`
A new record has been written. `NewRecord` contains the new record value and `OldRecords` contains the records before the operation is performed. Notice that the new content depends on the table type.

`{delete, Table, What, [OldRecords], ActivityId}`
Records have possibly been deleted. `What` is either `{Table, Key}` or a record `{RecordName, Key, ...}` that was deleted. Notice that the new content depends on the table type.

1.6.8 Debugging Mnesia Applications

Debugging a `Mnesia` application can be difficult for various reasons, primarily related to difficulties in understanding how the transaction and table load mechanisms work. Another source of confusion can be the semantics of nested transactions.

The debug level of `Mnesia` is set by calling the function *mnesia:set_debug_level(Level)*, where `Level`is one of the following:

`none`
: No trace outputs. This is the default.

`verbose`
: Activates tracing of important debug events. These events generate `{mnesia_info, Format, Args}` system events. Processes can subscribe to these events with the function *mnesia:subscribe/1*. The events are always sent to the `Mnesia` event handler.

`debug`
: Activates all events at the verbose level plus traces of all debug events. These debug events generate `{mnesia_info, Format, Args}` system events. Processes can subscribe to these events with `mnesia:subscribe/1`. The events are always sent to the `Mnesia` event handler. On this debug level, the `Mnesia` event handler starts subscribing to updates in the schema table.

`trace`
: Activates all events at the debug level. On this level, the `Mnesia` event handler starts subscribing to updates on all `Mnesia` tables. This level is intended only for debugging small toy systems, as many large events can be generated.

`false`
: An alias for none.

`true`
: An alias for debug.

The debug level of `Mnesia` itself is also an application parameter, making it possible to start an Erlang system to turn on `Mnesia` debug in the initial startup phase by using the following code:

```
% erl -mnesia debug verbose
```

1.6.9 Concurrent Processes in Mnesia

Programming concurrent Erlang systems is the subject of a separate book. However, it is worthwhile to draw attention to the following features, which permit concurrent processes to exist in a `Mnesia` system:

- A group of functions or processes can be called within a transaction. A transaction can include statements that read, write, or delete data from the DBMS. Many such transactions can run concurrently, and the programmer does not need to explicitly synchronize the processes that manipulate the data.

  All programs accessing the database through the transaction system can be written as if they had sole access to the data. This is a desirable property, as all synchronization is taken care of by the transaction handler. If a program reads or writes data, the system ensures that no other program tries to manipulate the same data at the same time.
- Tables can be moved or deleted, and the layout of a table can be reconfigured in various ways. An important aspect of the implementation of these functions is that user programs can continue to use a table while it is being reconfigured. For example, it is possible to move a table and perform write operations to the table at the same time. This is important for many applications that require continuously available services. For more information, see *Transactions and Other Access Contexts*.

1.6.10 Prototyping

If and when you would like to start and manipulate `Mnesia`, it is often easier to write the definitions and data into an ordinary text file. Initially, no tables and no data exist, or which tables are required. At the initial stages of prototyping, it is prudent to write all data into one file, process that file, and have the data in the file inserted into the database. `Mnesia` can be initialized with data read from a text file. The following two functions can be used to work with text files.

- *mnesia:load_textfile(Filename)* loads a series of local table definitions and data found in the file into `Mnesia`. This function also starts `Mnesia` and possibly creates a new schema. The function operates on the local node only.

- *mnesia:dump_to_textfile(Filename)* dumps all local tables of a `Mnesia` system into a text file, which can be edited (with a normal text editor) and later reloaded.

These functions are much slower than the ordinary store and load functions of `Mnesia`. However, this is mainly intended for minor experiments and initial prototyping. The major advantage of these functions is that they are easy to use.

The format of the text file is as follows:

```
      {tables, [{Typename, [Options]},
      {Typename2 ......}]}.

      {Typename, Attribute1, Attribute2 ....}.
      {Typename, Attribute1, Attribute2 ....}.
```

`Options` is a list of `{Key,Value}` tuples conforming to the options that you can give to *mnesia:create_table/2*.

For example, to start playing with a small database for healthy foods, enter the following data into file `FRUITS`:

```
{tables,
 [{fruit, [{attributes, [name, color, taste]}]},
  {vegetable, [{attributes, [name, color, taste, price]}]}]}.


{fruit, orange, orange, sweet}.
{fruit, apple, green, sweet}.
{vegetable, carrot, orange, carrotish, 2.55}.
{vegetable, potato, yellow, none, 0.45}.
```

The following session with the Erlang shell shows how to load the `FRUITS` database:

```
% erl
Erlang (BEAM) emulator version 4.9

Eshell V4.9  (abort with ^G)
1> mnesia:load_textfile("FRUITS").
New table fruit
New table vegetable
{atomic,ok}
2> mnesia:info().
---> Processes holding locks <---
---> Processes waiting for locks <---
---> Pending (remote) transactions <---
---> Active (local) transactions <---
---> Uncertain transactions <---
---> Active tables <---
vegetable      : with 2 records occuping 299 words of mem
fruit          : with 2 records occuping 291 words of mem
schema         : with 3 records occuping 401 words of mem
===> System info in version "1.1", debug level = none <===
opt_disc. Directory "/var/tmp/Mnesia.nonode@nohost" is used.
use fallback at restart = false
running db nodes = [nonode@nohost]
stopped db nodes = []
remote           = []
ram_copies       = [fruit,vegetable]
disc_copies      = [schema]
disc_only_copies = []
[{nonode@nohost,disc_copies}] = [schema]
[{nonode@nohost,ram_copies}] = [fruit,vegetable]
3 transactions committed, 0 aborted, 0 restarted, 2 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
ok
3>
```

It can be seen that the DBMS was initiated from a regular text file.

1.6.11 Object-Based Programming with Mnesia

The `Company` database, introduced in *Getting Started*, has three tables that store records (`employee`, `dept`, `project`), and three tables that store relationships (`manager`, `at_dep`, `in_proj`). This is a normalized data model, which has some advantages over a non-normalized data model.

It is more efficient to do a generalized search in a normalized database. Some operations are also easier to perform on a normalized data model. For example, one project can easily be removed, as the following example illustrates:

```
remove_proj(ProjName) ->
    F = fun() ->
                Ip = qlc:e(qlc:q([X || X <- mnesia:table(in_proj),
          X#in_proj.proj_name == ProjName]
    )),
                mnesia:delete({project, ProjName}),
                del_in_projs(Ip)
        end,
    mnesia:transaction(F).

del_in_projs([Ip|Tail]) ->
    mnesia:delete_object(Ip),
    del_in_projs(Tail);
del_in_projs([]) ->
    done.
```

In reality, data models are seldom fully normalized. A realistic alternative to a normalized database model would be a data model that is not even in first normal form. `Mnesia` is suitable for applications such as telecommunications, because it is easy to organize data in a flexible manner. A `Mnesia` database is always organized as a set of tables. Each table is filled with rows, objects, and records. What sets `Mnesia` apart is that individual fields in a record can contain any type of compound data structures. An individual field in a record can contain lists, tuples, functions, and even record code.

Many telecommunications applications have unique requirements on lookup times for certain types of records. If the `Company` database had been a part of a telecommunications system, it could be to minimize the lookup time of an employee **together** with a list of the projects the employee is working on. If this is the case, a drastically different data model without direct relationships can be chosen. You would then have only the records themselves, and different records could contain either direct references to other records, or contain other records that are not part of the `Mnesia` schema.

The following record definitions can be created:

```
-record(employee, {emp_no,
     name,
     salary,
     sex,
     phone,
     room_no,
     dept,
     projects,
     manager}).


-record(dept, {id,
               name}).

-record(project, {name,
                  number,
                  location}).
```

A record that describes an employee can look as follows:

```
        Me = #employee{emp_no= 104732,
        name = klacke,
        salary = 7,
        sex = male,
        phone = 99586,
        room_no = {221, 015},
        dept = 'B/SFR',
        projects = [erlang, mnesia, otp],
        manager = 114872},
```

This model has only three different tables, and the employee records contain references to other records. The record has the following references:

- `'B/SFR'` refers to a `dept` record.
- `[erlang, mnesia, otp]` is a list of three direct references to three different `projects` records.
- `114872` refers to another employee record.

The `Mnesia` record identifiers (`{Tab, Key}`) can also be used as references. In this case, attribute `dept` would be set to value `{dept, 'B/SFR'}` instead of `'B/SFR'`.

With this data model, some operations execute considerably faster than they do with the normalized data model in the `Company` database. However, some other operations become much more complicated. In particular, it becomes more difficult to ensure that records do not contain dangling pointers to other non-existent, or deleted, records.

The following code exemplifies a search with a non-normalized data model. To find all employees at department `Dep` with a salary higher than `Salary`, use the following code:

```
get_emps(Salary, Dep) ->
    Q = qlc:q(
          [E || E <- mnesia:table(employee),
                E#employee.salary > Salary,
                E#employee.dept == Dep]
  ),
    F = fun() -> qlc:e(Q) end,
    transaction(F).
```

This code is easier to write and to understand, and it also executes much faster.

It is easy to show examples of code that executes faster if a non-normalized data model is used, instead of a normalized model. The main reason is that fewer tables are required. Therefore, data from different tables can more easily be combined in join operations. In the previous example, the function `get_emps/2` is transformed from a join operation into a simple query, which consists of a selection and a projection on one single table.

1.7 Mnesia System Information

The following topics are included:

- Database configuration data
- Core dumps
- Dumping tables
- Checkpoints
- Startup files, log file, and data files
- Loading tables at startup
- Recovery from communication failure
- Recovery of transactions
- Backup, restore, fallback, and disaster recovery

1.7.1 Database Configuration Data

The following two functions can be used to retrieve system information. For details, see the Reference Manual.

- *mnesia:table_info(Tab, Key) -> Info | exit({aborted,Reason})* returns information about one table, for example, the current size of the table and on which nodes it resides.
- *mnesia:system_info(Key) -> Info | exit({aborted, Reason})* returns information about the `Mnesia` system, for example, transaction statistics, `db_nodes`, and configuration parameters.

1.7.2 Core Dumps

If `Mnesia` malfunctions, system information is dumped to file `MnesiaCore.Node.When`. The type of system information contained in this file can also be generated with the function `mnesia_lib:coredump()`. If a `Mnesia` system behaves strangely, it is recommended that a `Mnesia` core dump file is included in the bug report.

1.7.3 Dumping Tables

Tables of type `ram_copies` are by definition stored in memory only. However, these tables can be dumped to disc, either at regular intervals or before the system is shut down. The function *mnesia:dump_tables(TabList)* dumps all replicas of a set of RAM tables to disc. The tables can be accessed while being dumped to disc. To dump the tables to disc, all replicas must have the storage type `ram_copies`.

The table content is placed in a `.DCD` file on the disc. When the `Mnesia` system is started, the RAM table is initially loaded with data from its `.DCD` file.

1.7.4 Checkpoints

A checkpoint is a transaction consistent state that spans over one or more tables. When a checkpoint is activated, the system remembers the current content of the set of tables. The checkpoint retains a transaction consistent state of the tables, allowing the tables to be read and updated while the checkpoint is active. A checkpoint is typically used to back up tables to external media, but they are also used internally in `Mnesia` for other purposes. Each checkpoint is independent and a table can be involved in several checkpoints simultaneously.

Each table retains its old contents in a checkpoint retainer. For performance critical applications, it can be important to realize the processing overhead associated with checkpoints. In a worst case scenario, the checkpoint retainer consumes more memory than the table itself. Also, each update becomes slightly slower on those nodes where checkpoint retainers are attached to the tables.

For each table, it is possible to choose if there is to be one checkpoint retainer attached to all replicas of the table, or if it is enough to have only one checkpoint retainer attached to a single replica. With a single checkpoint retainer per table, the checkpoint consumes less memory, but it is vulnerable to node crashes. With several redundant checkpoint retainers, the checkpoint survives as long as there is at least one active checkpoint retainer attached to each table.

Checkpoints can be explicitly deactivated with the function *mnesia:deactivate_checkpoint(Name)*, where `Name` is the name of an active checkpoint. This function returns `ok` if successful or `{error, Reason}` if there is an error. All tables in a checkpoint must be attached to at least one checkpoint retainer. The checkpoint is automatically deactivated by `Mnesia`, when any table lacks a checkpoint retainer. This can occur when a node goes down or when a replica is deleted. Use arguments `min` and `max` (described in the following list) to control the degree of checkpoint retainer redundancy.

Checkpoints are activated with the function *mnesia:activate_checkpoint(Args)*, where `Args` is a list of the following tuples:

- `{name,Name}`, where `Name` specifies a temporary name of the checkpoint. The name can be reused when the checkpoint has been deactivated. If no name is specified, a name is generated automatically.
- `{max,MaxTabs}`, where `MaxTabs` is a list of tables that are to be included in the checkpoint. Default is `[]` (empty list). For these tables, the redundancy is maximized. The old content of the table is retained in the checkpoint retainer when the main table is updated by the applications. The checkpoint is more fault tolerant if the tables have several replicas. When new replicas are added by the schema manipulation function *mnesia:add_table_copy/3* it also attaches a local checkpoint retainer.
- `{min,MinTabs}`, where `MinTabs` is a list of tables that are to be included in the checkpoint. Default is `[]`. For these tables, the redundancy is minimized, and there is to be single checkpoint retainer per table, preferably at the local node.
- `{allow_remote,Bool}`, where `false` means that all checkpoint retainers must be local. If a table does not reside locally, the checkpoint cannot be activated. `true` allows checkpoint retainers to be allocated on any node. Default is `true`.
- `{ram_overrides_dump,Bool}`. This argument only applies to tables of type `ram_copies`. `Bool` specifies if the table state in RAM is to override the table state on disc. `true` means that the latest committed records in RAM are included in the checkpoint retainer. These are the records that the application accesses. `false` means that the records on the disc `.DAT` file are included in the checkpoint retainer. These records are loaded on startup. Default is `false`.

The function *mnesia:activate_checkpoint(Args)* returns one of the following values:

- `{ok, Name, Nodes}`
- `{error, Reason}`

`Name` is the checkpoint name. `Nodes` are the nodes where the checkpoint is known.

A list of active checkpoints can be obtained with the following functions:

- *mnesia:system_info(checkpoints)* returns all active checkpoints on the current node.
- *mnesia:table_info(Tab, checkpoints)* returns active checkpoints on a specific table.

1.7.5 Startup Files, Log File, and Data Files

This section describes the internal files that are created and maintained by the `Mnesia` system. In particular, the workings of the `Mnesia` log are described.

Startup Files

Start Mnesia states the following prerequisites for starting `Mnesia`:

- An Erlang session must be started and a `Mnesia` directory must be specified for the database.
- A database schema must be initiated, using the function *mnesia:create_schema/1*.

The following example shows how these tasks are performed:

Step 1: Start an Erlang session and specify a `Mnesia` directory for the database:

```
% erl -sname klacke -mnesia dir '"/ldisc/scratch/klacke"'
```

```
Erlang (BEAM) emulator version 4.9

Eshell V4.9  (abort with ^G)
(klacke@gin)1> mnesia:create_schema([node()]).
ok
(klacke@gin)2>
^Z
Suspended
```

Step 2: You can inspect the `Mnesia` directory to see what files have been created:

```
% ls -l /ldisc/scratch/klacke
-rw-rw-r--   1 klacke   staff       247 Aug 12 15:06 FALLBACK.BUP
```

The response shows that the file `FALLBACK.BUP` has been created. This is called a backup file, and it contains an initial schema. If more than one node in the function *mnesia:create_schema/1* had been specified, identical backup files would have been created on all nodes.

Step 3: Start `Mnesia`:

```
(klacke@gin)3>mnesia:start( ).
ok
```

Step 4: You can see the following listing in the `Mnesia` directory:

```
-rw-rw-r--   1 klacke   staff         86 May 26 19:03 LATEST.LOG
-rw-rw-r--   1 klacke   staff      34507 May 26 19:03 schema.DAT
```

The schema in the backup file `FALLBACK.BUP` has been used to generate the file `schema.DAT`. Since there are no other disc resident tables than the schema, no other data files were created. The file `FALLBACK.BUP` was removed after the successful "restoration". You also see some files that are for internal use by `Mnesia`.

Step 5: Create a table:

```
(klacke@gin)4> mnesia:create_table(foo,[{disc_copies, [node()]}]).
{atomic,ok}
```

Step 6: You can see the following listing in the `Mnesia` directory:

```
% ls -l /ldisc/scratch/klacke
-rw-rw-r-- 1 klacke staff    86 May 26 19:07 LATEST.LOG
-rw-rw-r-- 1 klacke staff    94 May 26 19:07 foo.DCD
-rw-rw-r-- 1 klacke staff  6679 May 26 19:07 schema.DAT
```

The file `foo.DCD` has been created. This file will eventually store all data that is written into the `foo` table.

Log File

When starting `Mnesia`, a `.LOG` file called `LATEST.LOG` is created and placed in the database directory. This file is used by `Mnesia` to log disc-based transactions. This includes all transactions that write at least one record in a table that is of storage type `disc_copies` or `disc_only_copies`. The file also includes all operations that manipulate the schema itself, such as creating new tables. The log format can vary with different implementations of `Mnesia`. The `Mnesia` log is currently implemented in the standard library module *disk_log* in `Kernel`.

The log file grows continuously and must be dumped at regular intervals. "Dumping the log file" means that `Mnesia` performs all the operations listed in the log and place the records in the corresponding `.DAT`, `.DCD`, and `.DCL` data files. For example, if the operation "write record `{foo, 4, elvis, 6}`" is listed in the log, `Mnesia` inserts the operation into the file `foo.DCL`. Later, when `Mnesia` thinks that the `.DCL` file is too large, the data is moved to the `.DCD` file. The dumping operation can be time consuming if the log is large. Notice that the `Mnesia` system continues to operate during log dumps.

By default `Mnesia` either dumps the log whenever 1000 records have been written in the log or when three minutes have passed. This is controlled by the two application parameters `-mnesia dump_log_write_threshold WriteOperations` and `-mnesia dump_log_time_threshold MilliSecs`.

Before the log is dumped, the file `LATEST.LOG` is renamed to `PREVIOUS.LOG`, and a new `LATEST.LOG` file is created. Once the log has been successfully dumped, the file `PREVIOUS.LOG` is deleted.

The log is also dumped at startup and whenever a schema operation is performed.

Data Files

The directory listing also contains one `.DAT` file, which contains the schema itself, contained in the `schema.DAT` file. The `DAT` files are indexed files, and it is efficient to insert and search for records in these files with a specific key. The `.DAT` files are used for the schema and for `disc_only_copies` tables. The `Mnesia` data files are currently implemented in the standard library module *dets* in `STDLIB`.

All operations that can be performed on `dets` files can also be performed on the `Mnesia` data files. For example, `dets` contains the function `dets:traverse/2`, which can be used to view the contents of a `Mnesia` `DAT` file. However, this can only be done when `Mnesia` is not running. So, to view the schema file, do as follows;

```
{ok, N} = dets:open_file(schema, [{file, "./schema.DAT"},{repair,false},
{keypos, 2}]),
F = fun(X) -> io:format("~p~n", [X]), continue end,
dets:traverse(N, F),
dets:close(N).
```

Warning:

The `DAT` files must always be opened with option `{repair, false}`. This ensures that these files are not automatically repaired. Without this option, the database can become inconsistent, because `Mnesia` can believe that the files were properly closed. For information about configuration parameter `auto_repair`, see the Reference Manual.

Warning:

It is recommended that the data files are not tampered with while `Mnesia` is running. While not prohibited, the behavior of `Mnesia` is unpredictable.

The `disc_copies` tables are stored on disk with `.DCL` and `.DCD` files, which are standard `disk_log` files.

1.7.6 Loading Tables at Startup

At startup, `Mnesia` loads tables to make them accessible for its applications. Sometimes `Mnesia` decides to load all tables that reside locally, and sometimes the tables are not accessible until `Mnesia` brings a copy of the table from another node.

To understand the behavior of `Mnesia` at startup, it is essential to understand how `Mnesia` reacts when it loses contact with `Mnesia` on another node. At this stage, `Mnesia` cannot distinguish between a communication failure and a "normal" node-down. When this occurs, `Mnesia` assumes that the other node is no longer running, whereas, in reality, the communication between the nodes has failed.

To overcome this situation, try to restart the ongoing transactions that are accessing tables on the failing node, and write a `mnesia_down` entry to a log file.

At startup, notice that all tables residing on nodes without a `mnesia_down` entry can have fresher replicas. Their replicas can have been updated after the termination of `Mnesia` on the current node. To catch up with the latest updates, transfer a copy of the table from one of these other "fresh" nodes. If you are unlucky, other nodes can be down and you must wait for the table to be loaded on one of these nodes before receiving a fresh copy of the table.

Before an application makes its first access to a table, *mnesia:wait_for_tables(TabList, Timeout)* is to be executed to ensure that the table is accessible from the local node. If the function times out, the application can choose to force a load of the local replica with *mnesia:force_load_table(Tab)* and deliberately lose all updates that can have been performed on the other nodes while the local node was down. If `Mnesia` has loaded the table on another node already, or intends to do so, copy the table from that node to avoid unnecessary inconsistency.

Warning:

Only one table is loaded by *mnesia:force_load_table(Tab)*. Since committed transactions can have caused updates in several tables, the tables can become inconsistent because of the forced load.

The allowed `AccessMode` of a table can be defined to be `read_only` or `read_write`. It can be toggled with the function *mnesia:change_table_access_mode(Tab, AccessMode)* in runtime. `read_only` tables and `local_content` tables are always loaded locally, as there is no need for copying the table from other nodes. Other tables are primarily loaded remotely from active replicas on other nodes if the table has been loaded there already, or if the running `Mnesia` has decided to load the table there already.

At startup, `Mnesia` assumes that its local replica is the most recent version and loads the table from disc if either of the following situations is detected:

- `mnesia_down` is returned from all other nodes that hold a disc resident replica of the table.
- All replicas are `ram_copies`.

This is normally a wise decision, but it can be disastrous if the nodes have been disconnected because of a communication failure, as the `Mnesia` normal table load mechanism does not cope with communication failures.

When `Mnesia` loads many tables, the default load order is used. However, the load order can be affected, by explicitly changing property `load_order` for the tables, with the function *mnesia:change_table_load_order(Tab, LoadOrder)*. `LoadOrder` is by default `0` for all tables, but it can be set to any integer. The table with the highest `load_order` is loaded first. Changing the load order is especially useful for applications that need to ensure early availability of fundamental tables. Large peripheral tables are to have a low load order value, perhaps less than `0`

1.7.7 Recovery from Communication Failure

There are several occasions when `Mnesia` can detect that the network has been partitioned because of a communication failure, for example:

- `Mnesia` is operational already and the Erlang nodes gain contact again. Then `Mnesia` tries to contact `Mnesia` on the other node to see if it also thinks that the network has been partitioned for a while. If `Mnesia` on both nodes has logged `mnesia_down` entries from each other, `Mnesia` generates a system event, called `{inconsistent_database, running_partitioned_network, Node}`, which is sent to the `Mnesia` event handler and other possible subscribers. The default event handler reports an error to the error logger.
- If `Mnesia` detects at startup that both the local node and another node received `mnesia_down` from each other, `Mnesia` generates an `{inconsistent_database, starting_partitioned_network, Node}` system event and acts as described in the previous item.

If the application detects that there has been a communication failure that can have caused an inconsistent database, it can use the function *mnesia:set_master_nodes(Tab, Nodes)* to pinpoint from which nodes each table can be loaded.

At startup, the `Mnesia` normal table load algorithm is bypassed and the table is loaded from one of the master nodes defined for the table, regardless of potential `mnesia_down` entries in the log. `Nodes` can only contain nodes where the table has a replica. If `Nodes` is empty, the master node recovery mechanism for the particular table is reset and the normal load mechanism is used at the next restart.

The function *mnesia:set_master_nodes(Nodes)* sets master nodes for all tables. For each table it determines its replica nodes and starts *mnesia:set_master_nodes(Tab, TabNodes)* with those replica nodes that are included in the `Nodes` list (that is, `TabNodes` is the intersection of `Nodes` and the replica nodes of the table). If the intersection is empty, the master node recovery mechanism for the particular table is reset and the normal load mechanism is used at the next restart.

The functions *mnesia:system_info(master_node_tables)* and *mnesia:table_info(Tab, master_nodes)* can be used to obtain information about the potential master nodes.

Determining what data to keep after a communication failure is outside the scope of `Mnesia`. One approach is to determine which "island" contains most of the nodes. Using option `{majority,true}` for critical tables can be a way to ensure that nodes that are not part of a "majority island" cannot update those tables. Notice that this constitutes a reduction in service on the minority nodes. This would be a tradeoff in favor of higher consistency guarantees.

The function *mnesia:force_load_table(Tab)* can be used to force load the table regardless of which table load mechanism that is activated.

1.7.8 Recovery of Transactions

A `Mnesia` table can reside on one or more nodes. When a table is updated, `Mnesia` ensures that the updates are replicated to all nodes where the table resides. If a replica is inaccessible (for example, because of a temporary node-down), `Mnesia` performs the replication later.

On the node where the application is started, there is a transaction coordinator process. If the transaction is distributed, there is also a transaction participant process on all the other nodes where commit-work needs to be performed.

Internally `Mnesia` uses several commit protocols. The selected protocol depends on which table that has been updated in the transaction. If all the involved tables are symmetrically replicated (that is, they all have the same `ram_nodes`, `disc_nodes`, and `disc_only_nodes` currently accessible from the coordinator node), a lightweight transaction commit protocol is used.

The number of messages that the transaction coordinator and its participants need to exchange is few, as the `Mnesia` table load mechanism takes care of the transaction recovery if the commit protocol gets interrupted. Since all involved tables are replicated symmetrically, the transaction is automatically recovered by loading the involved tables from the same node at startup of a failing node. It does not matter if the transaction was committed or terminated as long as the ACID properties can be ensured. The lightweight commit protocol is non-blocking, that is, the surviving participants and their coordinator finish the transaction, even if any node crashes in the middle of the commit protocol.

If a node goes down in the middle of a dirty operation, the table load mechanism ensures that the update is performed on all replicas, or none. Both asynchronous dirty updates and synchronous dirty updates use the same recovery principle as lightweight transactions.

If a transaction involves updates of asymmetrically replicated tables or updates of the schema table, a heavyweight commit protocol is used. This protocol can finish the transaction regardless of how the tables are replicated. The typical use of a heavyweight transaction is when a replica is to be moved from one node to another. Then ensure that the replica either is entirely moved or left as it was. Do never end up in a situation with replicas on both nodes, or on no node at all. Even if a node crashes in the middle of the commit protocol, the transaction must be guaranteed to be atomic. The heavyweight commit protocol involves more messages between the transaction coordinator and its participants than a lightweight protocol, and it performs recovery work at startup to finish the terminating or commit work.

The heavyweight commit protocol is also non-blocking, which allows the surviving participants and their coordinator to finish the transaction regardless (even if a node crashes in the middle of the commit protocol). When a node fails at startup, `Mnesia` determines the outcome of the transaction and recovers it. Lightweight protocols, heavyweight protocols, and dirty updates, are dependent on other nodes to be operational to make the correct heavyweight transaction recovery decision.

If `Mnesia` has not started on some of the nodes that are involved in the transaction **and** neither the local node nor any of the already running nodes know the outcome of the transaction, `Mnesia` waits for one, by default. In the worst case scenario, all other involved nodes must start before `Mnesia` can make the correct decision about the transaction and finish its startup.

Thus, `Mnesia` (on one node) can hang if a double fault occurs, that is, when two nodes crash simultaneously and one attempts to start when the other refuses to start, for example, because of a hardware error.

The maximum time that `Mnesia` waits for other nodes to respond with a transaction recovery decision can be specified. The configuration parameter `max_wait_for_decision` defaults to `infinity`, which can cause the indefinite hanging as mentioned earlier. However, if the parameter is set to a definite time period (for example, three minutes), `Mnesia` then enforces a transaction recovery decision, if needed, to allow `Mnesia` to continue with its startup procedure.

The downside of an enforced transaction recovery decision is that the decision can be incorrect, because of insufficient information about the recovery decisions from the other nodes. This can result in an inconsistent database where `Mnesia` has committed the transaction on some nodes but terminated it on others.

In fortunate cases, the inconsistency is only visible in tables belonging to a specific application. However, if a schema transaction is inconsistently recovered because of the enforced transaction recovery decision, the effects of the inconsistency can be fatal. However, if the higher priority is availability rather than consistency, it can be worth the risk.

If `Mnesia` detects an inconsistent transaction decision, an `{inconsistent_database, bad_decision, Node}` system event is generated to give the application a chance to install a fallback or other appropriate measures to resolve the inconsistency. The default behavior of the `Mnesia` event handler is the same as if the database became inconsistent as a result of partitioned network (as described earlier).

1.7.9 Backup, Restore, Fallback, and Disaster Recovery

The following functions are used to back up data, to install a backup as fallback, and for disaster recovery:

- *mnesia:backup_checkpoint(Name, Opaque, [Mod])* performs a backup of the tables included in the checkpoint.
- *mnesia:backup(Opaque, [Mod])* activates a new checkpoint that covers all `Mnesia` tables and performs a backup. It is performed with maximum degree of redundancy (see also the function *mnesia:activate_checkpoint(Args)*, `{max, MaxTabs} and {min, MinTabs}`).
- *mnesia:traverse_backup(Source, [SourceMod,] Target, [TargetMod,] Fun, Acc)* can be used to read an existing backup, create a backup from an existing one, or to copy a backup from one type media to another.
- *mnesia:uninstall_fallback()* removes previously installed fallback files.
- *mnesia:restore(Opaque, Args)* restores a set of tables from a previous backup.
- *mnesia:install_fallback(Opaque, [Mod])* can be configured to restart `Mnesia` and the reload data tables, and possibly the schema tables, from an existing backup. This function is typically used for disaster recovery purposes, when data or schema tables are corrupted.

These functions are explained in the following sections. See also *Checkpoints*, which describes the two functions used to activate and deactivate checkpoints.

Backup

Backup operation are performed with the following functions:

- *mnesia:backup_checkpoint(Name, Opaque, [Mod])*
- *mnesia:backup(Opaque, [Mod])*
- *mnesia:traverse_backup(Source, [SourceMod,] Target, [TargetMod,] Fun, Acc)*

By default, the actual access to the backup media is performed through module `mnesia_backup` for both read and write. Currently `mnesia_backup` is implemented with the standard library module `disc_log`. However, you can write your own module with the same interface as `mnesia_backup` and configure `Mnesia` so that the alternative module performs the actual accesses to the backup media. The user can therefore put the backup on a media that `Mnesia` does not know about, possibly on hosts where Erlang is not running. Use configuration parameter `-mnesia backup_module <module>` for this purpose.

The source for a backup is an activated checkpoint. The backup function *mnesia:backup_checkpoint(Name, Opaque, [Mod])* is most commonly used and returns `ok` or `{error,Reason}`. It has the following arguments:

- `Name` is the name of an activated checkpoint. For details on how to include table names in checkpoints, see the function `mnesia:activate_checkpoint(ArgList)` in *Checkpoints*.
- `Opaque`. `Mnesia` does not interpret this argument, but it is forwarded to the backup module. The `Mnesia` default backup module `mnesia_backup` interprets this argument as a local filename.
- `Mod` is the name of an alternative backup module.

The function *mnesia:backup(Opaque [,Mod])* activates a new checkpoint that covers all `Mnesia` tables with maximum degree of redundancy and performs a backup. Maximum redundancy means that each table replica has a checkpoint retainer. Tables with property `local_contents` are backed up as they look on the current node.

You can iterate over a backup, either to transform it into a new backup, or only read it. The function *mnesia:traverse_backup(Source, [SourceMod,] Target, [TargetMod,] Fun, Acc)*, which normally returns `{ok, LastAcc}`, is used for both of these purposes.

Before the traversal starts, the source backup media is opened with `SourceMod:open_read(Source)`, and the target backup media is opened with `TargetMod:open_write(Target)`. The arguments are as follows:

- `SourceMod` and `TargetMod` are module names.
- `Source` and `Target` are opaque data used exclusively by the modules `SourceMod` and `TargetMod` for initializing the backup medias.

- `Acc` is an initial accumulator value.
- `Fun(BackupItems, Acc)` is applied to each item in the backup. The Fun must return a tuple `{ValGoodBackupItems, NewAcc}`, where `ValidBackupItems` is a list of valid backup items. `NewAcc` is a new accumulator value. The `ValidBackupItems` are written to the target backup with the function `TargetMod:write/2`.
- `LastAcc` is the last accumulator value, that is, the last `NewAcc` value that was returned by `Fun`.

Also, a read-only traversal of the source backup can be performed without updating a target backup. If `TargetMod==read_only`, no target backup is accessed.

By setting `SourceMod` and `TargetMod` to different modules, a backup can be copied from one backup media to another.

Valid `BackupItems` are the following tuples:

- `{schema, Tab}` specifies a table to be deleted.
- `{schema, Tab, CreateList}` specifies a table to be created. For more information about `CreateList`, see *mnesia:create_table/2*.
- `{Tab, Key}` specifies the full identity of a record to be deleted.
- `{Record}` specifies a record to be inserted. It can be a tuple with `Tab` as first field. Notice that the record name is set to the table name regardless of what `record_name` is set to.

The backup data is divided into two sections. The first section contains information related to the schema. All schema-related items are tuples where the first field equals the atom schema. The second section is the record section. Schema records cannot be mixed with other records and all schema records must be located first in the backup.

The schema itself is a table and is possibly included in the backup. Each node where the schema table resides is regarded as a `db_node`.

The following example shows how *mnesia:traverse_backup* can be used to rename a `db_node` in a backup file:

```
change_node_name(Mod, From, To, Source, Target) ->
    Switch =
        fun(Node) when Node == From -> To;
           (Node) when Node == To -> throw({error, already_exists});
           (Node) -> Node
        end,
    Convert =
        fun({schema, db_nodes, Nodes}, Acc) ->
                {[{schema, db_nodes, lists:map(Switch,Nodes)}], Acc};
           ({schema, version, Version}, Acc) ->
                {[{schema, version, Version}], Acc};
           ({schema, cookie, Cookie}, Acc) ->
                {[{schema, cookie, Cookie}], Acc};
           ({schema, Tab, CreateList}, Acc) ->
                Keys = [ram_copies, disc_copies, disc_only_copies],
                OptSwitch =
                    fun({Key, Val}) ->
                            case lists:member(Key, Keys) of
                                true -> {Key, lists:map(Switch, Val)};
                                false-> {Key, Val}
                            end
                    end,
                {[{schema, Tab, lists:map(OptSwitch, CreateList)}], Acc};
           (Other, Acc) ->
                {[Other], Acc}
        end,
    mnesia:traverse_backup(Source, Mod, Target, Mod, Convert, switched).

view(Source, Mod) ->
    View = fun(Item, Acc) ->
                   io:format("~p.~n",[Item]),
                   {[Item], Acc + 1}
           end,
    mnesia:traverse_backup(Source, Mod, dummy, read_only, View, 0).
```

Restore

Tables can be restored online from a backup without restarting `Mnesia`. A restore is performed with the function *mnesia:restore(Opaque, Args)*, where `Args` can contain the following tuples:

- `{module,Mod}`. The backup module `Mod` is used to access the backup media. If omitted, the default backup module is used.
- `{skip_tables, TableList}`, where `TableList` is a list of tables, which is not to be read from the backup.
- `{clear_tables, TableList}`, where `TableList` is a list of tables, which is to be cleared before the records from the backup are inserted. That is, all records in the tables are deleted before the tables are restored. Schema information about the tables is not cleared or read from the backup.
- `{keep_tables, TableList}`, where `TableList` is a list of tables, which is not to be cleared before the records from the backup are inserted. That is, the records in the backup are added to the records in the table. Schema information about the tables is not cleared or read from the backup.
- `{recreate_tables, TableList}`, where `TableList` is a list of tables, which is to be recreated before the records from the backup are inserted. The tables are first deleted and then created with the schema information from the backup. All the nodes in the backup need to be operational.
- `{default_op, Operation}`, where `Operation` is one of the operations `skip_tables`, `clear_tables`, `keep_tables`, or `recreate_tables`. The default operation specifies which operation is to be used on tables from the backup that are not specified in any of the previous lists. If omitted, the operation `clear_tables` is used.

The argument `Opaque` is forwarded to the backup module. It returns `{atomic, TabList}` if successful, or the tuple `{aborted, Reason}` if there is an error. `TabList` is a list of the restored tables. Tables that are restored are write-locked during the restore operation. However, regardless of any lock conflict caused by this, applications can continue to do their work during the restore operation.

The restoration is performed as a single transaction. If the database is large, it cannot always be restored online. The old database must then be restored by installing a fallback, followed by a restart.

Fallback

The function *mnesia:install_fallback(Opaque, [Mod])* installs a backup as fallback. It uses the backup module `Mod`, or the default backup module, to access the backup media. The function returns `ok` if successful, or `{error, Reason}` if there is an error.

Installing a fallback is a distributed operation, which is **only** performed on all `db_nodes`. The fallback restores the database the next time the system is started. If a `Mnesia` node with a fallback installed detects that `Mnesia` on another node has died, it unconditionally terminates itself.

A fallback is typically used when a system upgrade is performed. A system typically involves the installation of new software versions, and `Mnesia` tables are often transformed into new layouts. If the system crashes during an upgrade, it is highly probable that reinstallation of the old applications is required, and restoration of the database to its previous state. This can be done if a backup is performed and installed as a fallback before the system upgrade begins.

If the system upgrade fails, `Mnesia` must be restarted on all `db_nodes` to restore the old database. The fallback is automatically deinstalled after a successful startup. The function *mnesia:uninstall_fallback()* can also be used to deinstall the fallback after a successful system upgrade. Again, this is a distributed operation that is either performed on all `db_nodes` or none. Both the installation and deinstallation of fallbacks require Erlang to be operational on all `db_nodes`, but it does not matter if `Mnesia` is running or not.

Disaster Recovery

The system can become inconsistent as a result of a power failure. The UNIX feature `fsck` can possibly repair the file system, but there is no guarantee that the file content is consistent.

If `Mnesia` detects that a file has not been properly closed, possibly as a result of a power failure, it tries to repair the bad file in a similar manner. Data can be lost, but `Mnesia` can be restarted even if the data is inconsistent. Configuration parameter `-mnesia auto_repair <bool>` can be used to control the behavior of `Mnesia` at startup. If `<bool>` has the value `true`, `Mnesia` tries to repair the file. If `<bool>` has the value `false`, `Mnesia` does not restart if it detects a suspect file. This configuration parameter affects the repair behavior of log files, `DAT` files, and the default backup media.

Configuration parameter `-mnesia dump_log_update_in_place <bool>` controls the safety level of the function *mnesia:dump_log()* By default, `Mnesia` dumps the transaction log directly into the `DAT` files. If a power failure occurs during the dump, this can cause the randomly accessed `DAT` files to become corrupt. If the parameter is set to `false`, `Mnesia` copies the `DAT` files and target the dump to the new temporary files. If the dump is successful, the temporary files are renamed to their normal `DAT` suffixes. The possibility for unrecoverable inconsistencies in the data files becomes much smaller with this strategy. However, the actual dumping of the transaction log becomes considerably slower. The system designer must decide whether speed or safety is the higher priority.

Replicas of type `disc_only_copies` are only affected by this parameter during the initial dump of the log file at startup. When designing applications with **very** high requirements, it can be appropriate not to use `disc_only_copies` tables at all. The reason for this is the random access nature of normal operating system files. If a node goes down for a reason such as a power failure, these files can be corrupted because they are not properly closed. The `DAT` files for `disc_only_copies` are updated on a per transaction basis.

If a disaster occurs and the `Mnesia` database is corrupted, it can be reconstructed from a backup. Regard this as a last resort, as the backup contains old data. The data is hopefully consistent, but data is definitely lost when an old backup is used to restore the database.

1.8 Combine Mnesia with SNMP

1.8.1 Combine Mnesia and SNMP

Many telecommunications applications must be controlled and reconfigured remotely. It is sometimes an advantage to perform this remote control with an open protocol such as the Simple Network Management Protocol (SNMP). The alternatives to this would be the following:

- Not being able to control the application remotely
- Using a proprietary control protocol
- Using a bridge that maps control messages in a proprietary protocol to a standardized management protocol and conversely

All these approaches have different advantages and disadvantages. Mnesia applications can easily be opened to the SNMP protocol. A direct 1-to-1 mapping can be established between Mnesia tables and SNMP tables. This means that a Mnesia table can be configured to be **both** a Mnesia table and an SNMP table. A number of functions to control this behavior are described in the Reference Manual.

1.9 Appendix A: Backup Callback Interface

1.9.1 mnesia_backup Callback Behavior

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% This module contains one implementation of callback functions
%% used by Mnesia at backup and restore. The user may however
%% write an own module the same interface as mnesia_backup and
%% configure Mnesia so the alternate module performs the actual
%% accesses to the backup media. This means that the user may put
%% the backup on medias that Mnesia does not know about, possibly
%% on hosts where Erlang is not running.
%%
%% The OpaqueData argument is never interpreted by other parts of
%% Mnesia. It is the property of this module. Alternate implementations
%% of this module may have different interpretations of OpaqueData.
%% The OpaqueData argument given to open_write/1 and open_read/1
%% are forwarded directly from the user.
%%
%% All functions must return {ok, NewOpaqueData} or {error, Reason}.
%%
%% The NewOpaqueData arguments returned by backup callback functions will
%% be given as input when the next backup callback function is invoked.
%% If any return value does not match {ok, _} the backup will be aborted.
%%
%% The NewOpaqueData arguments returned by restore callback functions will
%% be given as input when the next restore callback function is invoked
%% If any return value does not match {ok, _} the restore will be aborted.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(mnesia_backup).

-include_lib("kernel/include/file.hrl").

-export([
  %% Write access
         open_write/1,
  write/2,
  commit_write/1,
  abort_write/1,

  %% Read access
         open_read/1,
  read/1,
  close_read/1
        ]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Backup callback interface
-record(backup, {tmp_file, file, file_desc}).

%% Opens backup media for write
%%
%% Returns {ok, OpaqueData} or {error, Reason}
open_write(OpaqueData) ->
    File = OpaqueData,
    Tmp = lists:concat([File,".BUPTMP"]),
    file:delete(Tmp),
    file:delete(File),
    case disk_log:open([{name, make_ref()},
   {file, Tmp},
   {repair, false},
   {linkto, self()}]) of
 {ok, Fd} ->
     {ok, #backup{tmp_file = Tmp, file = File, file_desc = Fd}};
 {error, Reason} ->
     {error, Reason}
```

```
    end.

%% Writes BackupItems to the backup media
%%
%% Returns {ok, OpaqueData} or {error, Reason}
write(OpaqueData, BackupItems) ->
    B = OpaqueData,
    case disk_log:log_terms(B#backup.file_desc, BackupItems) of
        ok ->
            {ok, B};
        {error, Reason} ->
            abort_write(B),
            {error, Reason}
    end.

%% Closes the backup media after a successful backup
%%
%% Returns {ok, ReturnValueToUser} or {error, Reason}
commit_write(OpaqueData) ->
    B = OpaqueData,
    case disk_log:sync(B#backup.file_desc) of
        ok ->
            case disk_log:close(B#backup.file_desc) of
                ok ->
      case file:rename(B#backup.tmp_file, B#backup.file) of
         ok ->
       {ok, B#backup.file};
         {error, Reason} ->
       {error, Reason}
      end;
                {error, Reason} ->
      {error, Reason}
            end;
        {error, Reason} ->
            {error, Reason}
    end.

%% Closes the backup media after an interrupted backup
%%
%% Returns {ok, ReturnValueToUser} or {error, Reason}
abort_write(BackupRef) ->
    Res = disk_log:close(BackupRef#backup.file_desc),
    file:delete(BackupRef#backup.tmp_file),
    case Res of
        ok ->
            {ok, BackupRef#backup.file};
        {error, Reason} ->
            {error, Reason}
    end.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Restore callback interface

-record(restore, {file, file_desc, cont}).

%% Opens backup media for read
%%
%% Returns {ok, OpaqueData} or {error, Reason}
open_read(OpaqueData) ->
    File = OpaqueData,
    case file:read_file_info(File) of
 {error, Reason} ->
     {error, Reason};
 _FileInfo -> %% file exists
     case disk_log:open([{file, File},
```

```
    {name, make_ref()},
    {repair, false},
    {mode, read_only},
    {linkto, self()}]) of
  {ok, Fd} ->
      {ok, #restore{file = File, file_desc = Fd, cont = start}};
  {repaired, Fd, _, {badbytes, 0}} ->
      {ok, #restore{file = File, file_desc = Fd, cont = start}};
  {repaired, Fd, _, _} ->
      {ok, #restore{file = File, file_desc = Fd, cont = start}};
  {error, Reason} ->
      {error, Reason}
     end
    end.

%% Reads BackupItems from the backup media
%%
%% Returns {ok, OpaqueData, BackupItems} or {error, Reason}
%%
%% BackupItems == [] is interpreted as eof
read(OpaqueData) ->
    R = OpaqueData,
    Fd = R#restore.file_desc,
    case disk_log:chunk(Fd, R#restore.cont) of
        {error, Reason} ->
            {error, {"Possibly truncated", Reason}};
        eof ->
            {ok, R, []};
        {Cont, []} ->
            read(R#restore{cont = Cont});
        {Cont, BackupItems, _BadBytes} ->
            {ok, R#restore{cont = Cont}, BackupItems};
        {Cont, BackupItems} ->
            {ok, R#restore{cont = Cont}, BackupItems}
    end.

%% Closes the backup media after restore
%%
%% Returns {ok, ReturnValueToUser} or {error, Reason}
close_read(OpaqueData) ->
    R = OpaqueData,
    case disk_log:close(R#restore.file_desc) of
        ok -> {ok, R#restore.file};
        {error, Reason} -> {error, Reason}
    end.
```

1.10 Appendix B: Activity Access Callback Interface

1.10.1 mnesia_access Callback Behavior

```
-module(mnesia_frag).

%% Callback functions when accessed within an activity
-export([
  lock/4,
  write/5, delete/5, delete_object/5,
  read/5, match_object/5, all_keys/4,
  select/5,select/6,select_cont/3,
  index_match_object/6, index_read/6,
  foldl/6, foldr/6, table_info/4,
  first/3, next/4, prev/4, last/3,
  clear_table/4
       ]).
```

```
%% Callback functions which provides transparent
%% access of fragmented tables from any activity
%% access context.

lock(ActivityId, Opaque, {table , Tab}, LockKind) ->
    case frag_names(Tab) of
 [Tab] ->
     mnesia:lock(ActivityId, Opaque, {table, Tab}, LockKind);
 Frags ->
     DeepNs = [mnesia:lock(ActivityId, Opaque, {table, F}, LockKind) ||
    F <- Frags],
     mnesia_lib:uniq(lists:append(DeepNs))
    end;

lock(ActivityId, Opaque, LockItem, LockKind) ->
    mnesia:lock(ActivityId, Opaque, LockItem, LockKind).

write(ActivityId, Opaque, Tab, Rec, LockKind) ->
    Frag = record_to_frag_name(Tab, Rec),
    mnesia:write(ActivityId, Opaque, Frag, Rec, LockKind).

delete(ActivityId, Opaque, Tab, Key, LockKind) ->
    Frag = key_to_frag_name(Tab, Key),
    mnesia:delete(ActivityId, Opaque, Frag, Key, LockKind).

delete_object(ActivityId, Opaque, Tab, Rec, LockKind) ->
    Frag = record_to_frag_name(Tab, Rec),
    mnesia:delete_object(ActivityId, Opaque, Frag, Rec, LockKind).

read(ActivityId, Opaque, Tab, Key, LockKind) ->
    Frag = key_to_frag_name(Tab, Key),
    mnesia:read(ActivityId, Opaque, Frag, Key, LockKind).

match_object(ActivityId, Opaque, Tab, HeadPat, LockKind) ->
    MatchSpec = [{HeadPat, [], ['$_']}],
    select(ActivityId, Opaque, Tab, MatchSpec, LockKind).

select(ActivityId, Opaque, Tab, MatchSpec, LockKind) ->
    do_select(ActivityId, Opaque, Tab, MatchSpec, LockKind).


select(ActivityId, Opaque, Tab, MatchSpec, Limit, LockKind) ->
    init_select(ActivityId, Opaque, Tab, MatchSpec, Limit, LockKind).


all_keys(ActivityId, Opaque, Tab, LockKind) ->
    Match = [mnesia:all_keys(ActivityId, Opaque, Frag, LockKind)
      || Frag <- frag_names(Tab)],
    lists:append(Match).

clear_table(ActivityId, Opaque, Tab, Obj) ->
    [mnesia:clear_table(ActivityId, Opaque, Frag, Obj)  || Frag <- frag_names(Tab)],
    ok.

index_match_object(ActivityId, Opaque, Tab, Pat, Attr, LockKind) ->
    Match =
 [mnesia:index_match_object(ActivityId, Opaque, Frag, Pat, Attr, LockKind)
  || Frag <- frag_names(Tab)],
    lists:append(Match).

index_read(ActivityId, Opaque, Tab, Key, Attr, LockKind) ->
    Match =
 [mnesia:index_read(ActivityId, Opaque, Frag, Key, Attr, LockKind)
      || Frag <- frag_names(Tab)],
    lists:append(Match).
```

```
foldl(ActivityId, Opaque, Fun, Acc, Tab, LockKind) ->
    Fun2 = fun(Frag, A) ->
     mnesia:foldl(ActivityId, Opaque, Fun, A, Frag, LockKind)
    end,
    lists:foldl(Fun2, Acc, frag_names(Tab)).

foldr(ActivityId, Opaque, Fun, Acc, Tab, LockKind) ->
    Fun2 = fun(Frag, A) ->
     mnesia:foldr(ActivityId, Opaque, Fun, A, Frag, LockKind)
    end,
    lists:foldr(Fun2, Acc, frag_names(Tab)).

table_info(ActivityId, Opaque, {Tab, Key}, Item) ->
    Frag = key_to_frag_name(Tab, Key),
    table_info2(ActivityId, Opaque, Tab, Frag, Item);
table_info(ActivityId, Opaque, Tab, Item) ->
    table_info2(ActivityId, Opaque, Tab, Tab, Item).

table_info2(ActivityId, Opaque, Tab, Frag, Item) ->
    case Item of
 size ->
     SumFun = fun({_, Size}, Acc) -> Acc + Size end,
     lists:foldl(SumFun, 0, frag_size(ActivityId, Opaque, Tab));
 memory ->
     SumFun = fun({_, Size}, Acc) -> Acc + Size end,
     lists:foldl(SumFun, 0, frag_memory(ActivityId, Opaque, Tab));
 base_table ->
     lookup_prop(Tab, base_table);
 node_pool ->
     lookup_prop(Tab, node_pool);
 n_fragments ->
     FH = lookup_frag_hash(Tab),
     FH#frag_state.n_fragments;
 foreign_key ->
     FH = lookup_frag_hash(Tab),
     FH#frag_state.foreign_key;
 foreigners ->
     lookup_foreigners(Tab);
 n_ram_copies ->
     length(val({Tab, ram_copies}));
 n_disc_copies ->
     length(val({Tab, disc_copies}));
 n_disc_only_copies ->
     length(val({Tab, disc_only_copies}));
 n_external_copies ->
     length(val({Tab, external_copies}));

 frag_names ->
     frag_names(Tab);
 frag_dist ->
     frag_dist(Tab);
 frag_size ->
     frag_size(ActivityId, Opaque, Tab);
 frag_memory ->
     frag_memory(ActivityId, Opaque, Tab);
 _ ->
     mnesia:table_info(ActivityId, Opaque, Frag, Item)
    end.

first(ActivityId, Opaque, Tab) ->
    case ?catch_val({Tab, frag_hash}) of
 {'EXIT', _} ->
     mnesia:first(ActivityId, Opaque, Tab);
 FH ->
```

```
     FirstFrag = Tab,
     case mnesia:first(ActivityId, Opaque, FirstFrag) of
  '$end_of_table' ->
      search_first(ActivityId, Opaque, Tab, 1, FH);
  Next ->
      Next
     end
    end.

search_first(ActivityId, Opaque, Tab, N, FH) when N < FH#frag_state.n_fragments ->
    NextN = N + 1,
    NextFrag = n_to_frag_name(Tab, NextN),
    case mnesia:first(ActivityId, Opaque, NextFrag) of
 '$end_of_table' ->
     search_first(ActivityId, Opaque, Tab, NextN, FH);
 Next ->
     Next
    end;
search_first(_ActivityId, _Opaque, _Tab, _N, _FH) ->
    '$end_of_table'.

last(ActivityId, Opaque, Tab) ->
    case ?catch_val({Tab, frag_hash}) of
 {'EXIT', _} ->
     mnesia:last(ActivityId, Opaque, Tab);
 FH ->
     LastN = FH#frag_state.n_fragments,
     search_last(ActivityId, Opaque, Tab, LastN, FH)
    end.

search_last(ActivityId, Opaque, Tab, N, FH) when N >= 1 ->
    Frag = n_to_frag_name(Tab, N),
    case mnesia:last(ActivityId, Opaque, Frag) of
 '$end_of_table' ->
     PrevN = N - 1,
     search_last(ActivityId, Opaque, Tab, PrevN, FH);
 Prev ->
     Prev
    end;
search_last(_ActivityId, _Opaque, _Tab, _N, _FH) ->
    '$end_of_table'.

prev(ActivityId, Opaque, Tab, Key) ->
    case ?catch_val({Tab, frag_hash}) of
 {'EXIT', _} ->
     mnesia:prev(ActivityId, Opaque, Tab, Key);
 FH ->
     N = key_to_n(FH, Key),
     Frag = n_to_frag_name(Tab, N),
     case mnesia:prev(ActivityId, Opaque, Frag, Key) of
  '$end_of_table' ->
      search_prev(ActivityId, Opaque, Tab, N);
  Prev ->
      Prev
     end
    end.

search_prev(ActivityId, Opaque, Tab, N) when N > 1 ->
    PrevN = N - 1,
    PrevFrag = n_to_frag_name(Tab, PrevN),
    case mnesia:last(ActivityId, Opaque, PrevFrag) of
 '$end_of_table' ->
     search_prev(ActivityId, Opaque, Tab, PrevN);
 Prev ->
     Prev
```

```
    end;
search_prev(_ActivityId, _Opaque, _Tab, _N) ->
    '$end_of_table'.

next(ActivityId, Opaque, Tab, Key) ->
    case ?catch_val({Tab, frag_hash}) of
 {'EXIT', _} ->
     mnesia:next(ActivityId, Opaque, Tab, Key);
 FH ->
     N = key_to_n(FH, Key),
     Frag = n_to_frag_name(Tab, N),
     case mnesia:next(ActivityId, Opaque, Frag, Key) of
  '$end_of_table' ->
      search_next(ActivityId, Opaque, Tab, N, FH);
  Prev ->
      Prev
     end
    end.

search_next(ActivityId, Opaque, Tab, N, FH) when N < FH#frag_state.n_fragments ->
    NextN = N + 1,
    NextFrag = n_to_frag_name(Tab, NextN),
    case mnesia:first(ActivityId, Opaque, NextFrag) of
 '$end_of_table' ->
     search_next(ActivityId, Opaque, Tab, NextN, FH);
 Next ->
     Next
    end;
search_next(_ActivityId, _Opaque, _Tab, _N, _FH) ->
    '$end_of_table'.
```

1.11 Appendix C: Fragmented Table Hashing Callback Interface

1.11.1 mnesia_frag_hash Callback Behavior

```
-module(mnesia_frag_hash).

%% Fragmented Table Hashing callback functions
-export([
  init_state/2,
  add_frag/1,
  del_frag/1,
  key_to_frag_number/2,
  match_spec_to_frag_numbers/2
 ]).
```

```
-record(hash_state,
 {n_fragments,
  next_n_to_split,
  n_doubles,
  function}).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

init_state(_Tab, State) when State == undefined ->
    #hash_state{n_fragments     = 1,
  next_n_to_split = 1,
  n_doubles       = 0,
  function        = phash2}.

convert_old_state({hash_state, N, P, L}) ->
    #hash_state{n_fragments     = N,
  next_n_to_split = P,
  n_doubles       = L,
  function        = phash}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

add_frag(#hash_state{next_n_to_split = SplitN, n_doubles = L, n_fragments = N} = State) ->
    P = SplitN + 1,
    NewN = N + 1,
    State2 = case power2(L) + 1 of
   P2 when P2 == P ->
       State#hash_state{n_fragments      = NewN,
          n_doubles        = L + 1,
          next_n_to_split = 1};
   _ ->
       State#hash_state{n_fragments     = NewN,
          next_n_to_split = P}
      end,
    {State2, [SplitN], [NewN]};
add_frag(OldState) ->
    State = convert_old_state(OldState),
    add_frag(State).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

del_frag(#hash_state{next_n_to_split = SplitN, n_doubles = L, n_fragments = N} = State) ->
    P = SplitN - 1,
    if
 P < 1 ->
     L2 = L - 1,
     MergeN = power2(L2),
     State2 = State#hash_state{n_fragments     = N - 1,
          next_n_to_split = MergeN,
          n_doubles       = L2},
     {State2, [N], [MergeN]};
 true ->
     MergeN = P,
     State2 = State#hash_state{n_fragments     = N - 1,
          next_n_to_split = MergeN},
     {State2, [N], [MergeN]}
 end;
del_frag(OldState) ->
    State = convert_old_state(OldState),
    del_frag(State).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

key_to_frag_number(#hash_state{function = phash, n_fragments = N, n_doubles = L}, Key) ->
    A = erlang:phash(Key, power2(L + 1)),
```

```
    if
 A > N ->
     A - power2(L);
 true ->
     A
    end;
key_to_frag_number(#hash_state{function = phash2, n_fragments = N, n_doubles = L}, Key) ->
    A = erlang:phash2(Key, power2(L + 1)) + 1,
    if
 A > N ->
     A - power2(L);
 true ->
     A
    end;
key_to_frag_number(OldState, Key) ->
    State = convert_old_state(OldState),
    key_to_frag_number(State, Key).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

match_spec_to_frag_numbers(#hash_state{n_fragments = N} = State, MatchSpec) ->
    case MatchSpec of
 [{HeadPat, _, _}] when is_tuple(HeadPat), tuple_size(HeadPat) > 2 ->
     KeyPat = element(2, HeadPat),
     case has_var(KeyPat) of
  false ->
      [key_to_frag_number(State, KeyPat)];
  true ->
      lists:seq(1, N)
     end;
 _ ->
     lists:seq(1, N)
    end;
match_spec_to_frag_numbers(OldState, MatchSpec) ->
    State = convert_old_state(OldState),
    match_spec_to_frag_numbers(State, MatchSpec).

power2(Y) ->
    1 bsl Y. % trunc(math:pow(2, Y)).
```

2 Reference Manual

The Mnesia application is a distributed Database Management System (DBMS), appropriate for telecommunications applications and other Erlang applications, which require continuous operation and exhibit soft real-time properties.

mnesia

Erlang module

The following are some of the most important and attractive capabilities provided by Mnesia:

- A relational/object hybrid data model that is suitable for telecommunications applications.
- A DBMS query language, Query List Comprehension (QLC) as an add-on library.
- Persistence. Tables can be coherently kept on disc and in the main memory.
- Replication. Tables can be replicated at several nodes.
- Atomic transactions. A series of table manipulation operations can be grouped into a single atomic transaction.
- Location transparency. Programs can be written without knowledge of the actual data location.
- Extremely fast real-time data searches.
- Schema manipulation routines. The DBMS can be reconfigured at runtime without stopping the system.

This Reference Manual describes the Mnesia API. This includes functions that define and manipulate Mnesia tables.

All functions in this Reference Manual can be used in any combination with queries using the list comprehension notation. For information about the query notation, see the *qlc* manual page in STDLIB.

Data in Mnesia is organized as a set of tables. Each table has a name that must be an atom. Each table is made up of Erlang records. The user is responsible for the record definitions. Each table also has a set of properties. The following are some of the properties that are associated with each table:

- `type`. Each table can have `set`, `ordered_set`, or `bag` semantics. Notice that currently `ordered_set` is not supported for `disc_only_copies`.

  If a table is of type `set`, each key leads to either one or zero records.

  If a new item is inserted with the same key as an existing record, the old record is overwritten. However, if a table is of type `bag`, each key can map to several records. All records in type `bag` tables are unique, only the keys can be duplicated.
- `record_name`. All records stored in a table must have the same name. The records must be instances of the same record type.
- `ram_copies`. A table can be replicated on a number of Erlang nodes. Property `ram_copies` specifies a list of Erlang nodes where RAM copies are kept. These copies can be dumped to disc at regular intervals. However, updates to these copies are not written to disc on a transaction basis.
- `disc_copies`. This property specifies a list of Erlang nodes where the table is kept in RAM and on disc. All updates of the table are performed in the actual table and are also logged to disc. If a table is of type `disc_copies` at a certain node, the entire table is resident in RAM memory and on disc. Each transaction performed on the table is appended to a `LOG` file and written into the RAM table.
- `disc_only_copies`. Some, or all, table replicas can be kept on disc only. These replicas are considerably slower than the RAM-based replicas.
- `index`. This is a list of attribute names, or integers, which specify the tuple positions on which Mnesia is to build and maintain an extra index table.
- `local_content`. When an application requires tables whose contents are local to each node, `local_content` tables can be used. The table name is known to all Mnesia nodes, but its content is unique on each node. This means that access to such a table must be done locally. Set field `local_content` to `true` to enable the `local_content` behavior. Default is `false`.
- `majority`. This attribute is `true` or `false`; default is `false`. When `true`, a majority of the table replicas must be available for an update to succeed. Majority checking can be enabled on tables with mission-critical data, where it is vital to avoid inconsistencies because of network splits.

- `snmp`. Each (set-based) Mnesia table can be automatically turned into a Simple Network Management Protocol (SNMP) ordered table as well. This property specifies the types of the SNMP keys.
- `attributes`. The names of the attributes for the records that are inserted in the table.

For information about the complete set of table properties and their details, see `mnesia:create_table/2`.

This Reference Manual uses a table of persons to illustrate various examples. The following record definition is assumed:

```
-record(person, {name,
                 age = 0,
                 address = unknown,
                 salary = 0,
                 children = []}),
```

The first record attribute is the primary key, or key for short.

The function descriptions are sorted in alphabetical order. It is recommended to start to read about `mnesia:create_table/2`, `mnesia:lock/2`, and `mnesia:activity/4` before you continue and learn about the rest.

Writing or deleting in transaction-context creates a local copy of each modified record during the transaction. During iteration, that is, `mnesia:fold[lr]/4`, `mnesia:next/2`, `mnesia:prev/2`, and `mnesia:snmp_get_next_index/2`, Mnesia compensates for every written or deleted record, which can reduce the performance.

If possible, avoid writing or deleting records in the same transaction before iterating over the table.

Exports

`abort(Reason) -> transaction abort`

Makes the transaction silently return the tuple `{aborted, Reason}`. Termination of a Mnesia transaction means that an exception is thrown to an enclosing `catch`. Thus, the expression `catch mnesia:abort(x)` does not terminate the transaction.

`activate_checkpoint(Args) -> {ok,Name,Nodes} | {error,Reason}`

A checkpoint is a consistent view of the system. A checkpoint can be activated on a set of tables. This checkpoint can then be traversed and presents a view of the system as it existed at the time when the checkpoint was activated, even if the tables are being or have been manipulated.

`Args` is a list of the following tuples:

- `{name,Name}`. `Name` is the checkpoint name. Each checkpoint must have a name that is unique to the associated nodes. The name can be reused only once the checkpoint has been deactivated. By default, a name that is probably unique is generated.
- `{max,MaxTabs}`. `MaxTabs` is a list of tables that are to be included in the checkpoint. Default is `[]`. For these tables, the redundancy is maximized and checkpoint information is retained together with all replicas. The checkpoint becomes more fault tolerant if the tables have several replicas. When a new replica is added by the schema manipulation function `mnesia:add_table_copy/3`, a retainer is also attached automatically.
- `{min,MinTabs}`. `MinTabs` is a list of tables that are to be included in the checkpoint. Default is []. For these tables, the redundancy is minimized and the checkpoint information is only retained with one replica, preferably on the local node.
- `{allow_remote,Bool}`. `false` means that all retainers must be local. The checkpoint cannot be activated if a table does not reside locally. `true` allows retainers to be allocated on any node. Default is `true`.

- `{ram_overrides_dump,Bool}`. Only applicable for `ram_copies`. `Bool` allows you to choose to back up the table state as it is in RAM, or as it is on disc. `true` means that the latest committed records in RAM are to be included in the checkpoint. These are the records that the application accesses. `false` means that the records dumped to `DAT` files are to be included in the checkpoint. These records are loaded at startup. Default is `false`.

Returns `{ok,Name,Nodes}` or `{error,Reason}`. `Name` is the (possibly generated) checkpoint name. `Nodes` are the nodes that are involved in the checkpoint. Only nodes that keep a checkpoint retainer know about the checkpoint.

```
activity(AccessContext, Fun [, Args]) -> ResultOfFun | exit(Reason)
```

Calls `mnesia:activity(AccessContext, Fun, Args, AccessMod)`, where `AccessMod` is the default access callback module obtained by `mnesia:system_info(access_module)`. `Args` defaults to `[]` (empty list).

```
activity(AccessContext, Fun, Args, AccessMod) -> ResultOfFun | exit(Reason)
```

Executes the functional object `Fun` with argument `Args`.

The code that executes inside the activity can consist of a series of table manipulation functions, which are performed in an `AccessContext`. Currently, the following access contexts are supported:

`transaction`

Short for `{transaction, infinity}`

`{transaction, Retries}`

Calls `mnesia:transaction(Fun, Args, Retries)`. Notice that the result from `Fun` is returned if the transaction is successful (atomic), otherwise the function exits with an abort reason.

`sync_transaction`

Short for `{sync_transaction, infinity}`

`{sync_transaction, Retries}`

Calls `mnesia:sync_transaction(Fun, Args, Retries)`. Notice that the result from `Fun` is returned if the transaction is successful (atomic), otherwise the function exits with an abort reason.

`async_dirty`

Calls `mnesia:async_dirty(Fun, Args)`.

`sync_dirty`

Calls `mnesia:sync_dirty(Fun, Args)`.

`ets`

Calls `mnesia:ets(Fun, Args)`.

This function (`mnesia:activity/4`) differs in an important way from the functions `mnesia:transaction`, `mnesia:sync_transaction`, `mnesia:async_dirty`, `mnesia:sync_dirty`, and `mnesia:ets`. Argument `AccessMod` is the name of a callback module, which implements the `mnesia_access` behavior.

Mnesia forwards calls to the following functions:

- mnesia:lock/2 (read_lock_table/1, write_lock_table/1)
- mnesia:write/3 (write/1, s_write/1)
- mnesia:delete/3 (delete/1, s_delete/1)
- mnesia:delete_object/3 (delete_object/1, s_delete_object/1)
- mnesia:read/3 (read/1, wread/1)
- mnesia:match_object/3 (match_object/1)

- mnesia:all_keys/1
- mnesia:first/1
- mnesia:last/1
- mnesia:prev/2
- mnesia:next/2
- mnesia:index_match_object/4 (index_match_object/2)
- mnesia:index_read/3
- mnesia:table_info/2

to the corresponding:

- AccessMod:lock(ActivityId, Opaque, LockItem, LockKind)
- AccessMod:write(ActivityId, Opaque, Tab, Rec, LockKind)
- AccessMod:delete(ActivityId, Opaque, Tab, Key, LockKind)
- AccessMod:delete_object(ActivityId, Opaque, Tab, RecXS, LockKind)
- AccessMod:read(ActivityId, Opaque, Tab, Key, LockKind)
- AccessMod:match_object(ActivityId, Opaque, Tab, Pattern, LockKind)
- AccessMod:all_keys(ActivityId, Opaque, Tab, LockKind)
- AccessMod:first(ActivityId, Opaque, Tab)
- AccessMod:last(ActivityId, Opaque, Tab)
- AccessMod:prev(ActivityId, Opaque, Tab, Key)
- AccessMod:next(ActivityId, Opaque, Tab, Key)
- AccessMod:index_match_object(ActivityId, Opaque, Tab, Pattern, Attr, LockKind)
- AccessMod:index_read(ActivityId, Opaque, Tab, SecondaryKey, Attr, LockKind)
- AccessMod:table_info(ActivityId, Opaque, Tab, InfoItem)

`ActivityId` is a record that represents the identity of the enclosing Mnesia activity. The first field (obtained with `element(1, ActivityId)`) contains an atom, which can be interpreted as the activity type: `ets`, `async_dirty`, `sync_dirty`, or `tid`. `tid` means that the activity is a transaction. The structure of the rest of the identity record is internal to Mnesia.

`Opaque` is an opaque data structure that is internal to Mnesia.

`add_table_copy(Tab, Node, Type) -> {aborted, R} | {atomic, ok}`

Makes another copy of a table at the node `Node`. Argument `Type` must be either of the atoms `ram_copies`, `disc_copies`, or `disc_only_copies`. For example, the following call ensures that a disc replica of the `person` table also exists at node `Node`:

```
mnesia:add_table_copy(person, Node, disc_copies)
```

This function can also be used to add a replica of the table named `schema`.

`add_table_index(Tab, AttrName) -> {aborted, R} | {atomic, ok}`

Table indexes can be used whenever the user wants to use frequently some other field than the key field to look up records. If this other field has an associated index, these lookups can occur in constant time and space. For example, if your application wishes to use field `age` to find efficiently all persons with a specific age, it can be a good idea to have an index on field `age`. This can be done with the following call:

```
mnesia:add_table_index(person, age)
```

Indexes do not come for free. They occupy space that is proportional to the table size, and they cause insertions into the table to execute slightly slower.

`all_keys(Tab) -> KeyList | transaction abort`

Returns a list of all keys in the table named `Tab`. The semantics of this function is context-sensitive. For more information, see `mnesia:activity/4`. In transaction-context, it acquires a read lock on the entire table.

`async_dirty(Fun, [, Args]) -> ResultOfFun | exit(Reason)`

Calls the `Fun` in a context that is not protected by a transaction. The Mnesia function calls performed in the `Fun` are mapped to the corresponding dirty functions. This still involves logging, replication, and subscriptions, but there is no locking, local transaction storage, or commit protocols involved. Checkpoint retainers and indexes are updated, but they are updated dirty. As for normal `mnesia:dirty_*` operations, the operations are performed semi-asynchronously. For details, see `mnesia:activity/4` and the User's Guide.

The Mnesia tables can be manipulated without using transactions. This has some serious disadvantages, but is considerably faster, as the transaction manager is not involved and no locks are set. A dirty operation does, however, guarantee a certain level of consistency, and the dirty operations cannot return garbled records. All dirty operations provide location transparency to the programmer, and a program does not have to be aware of the whereabouts of a certain table to function.

Notice that it is more than ten times more efficient to read records dirty than within a transaction.

Depending on the application, it can be a good idea to use the dirty functions for certain operations. Almost all Mnesia functions that can be called within transactions have a dirty equivalent, which is much more efficient.

However, notice that there is a risk that the database can be left in an inconsistent state if dirty operations are used to update it. Dirty operations are only to be used for performance reasons when it is absolutely necessary.

Notice that calling (nesting) `mnesia:[a]sync_dirty` inside a transaction-context inherits the transaction semantics.

`backup(Opaque [, BackupMod]) -> ok | {error,Reason}`

Activates a new checkpoint covering all Mnesia tables, including the schema, with maximum degree of redundancy, and performs a backup using `backup_checkpoint/2/3`. The default value of the backup callback module `BackupMod` is obtained by `mnesia:system_info(backup_module)`.

`backup_checkpoint(Name, Opaque [, BackupMod]) -> ok | {error,Reason}`

The tables are backed up to external media using backup module `BackupMod`. Tables with the local contents property are backed up as they exist on the current node. `BackupMod` is the default backup callback module obtained by `mnesia:system_info(backup_module)`. For information about the exact callback interface (the `mnesia_backup behavior`), see the User's Guide.

`change_config(Config, Value) -> {error, Reason} | {ok, ReturnValue}`

`Config` is to be an atom of the following configuration parameters:

`extra_db_nodes`

> `Value` is a list of nodes that Mnesia is to try to connect to. `ReturnValue` is those nodes in `Value` that Mnesia is connected to.
>
> Notice that this function must only be used to connect to newly started RAM nodes (N.D.R.S.N.) with an empty schema. If, for example, this function is used after the network has been partitioned, it can lead to inconsistent tables.

Notice that Mnesia can be connected to other nodes than those returned in `ReturnValue`.

`dc_dump_limit`

`Value` is a number. See the description in *Section Configuration Parameters*. `ReturnValue` is the new value. Notice that this configuration parameter is not persistent. It is lost when Mnesia has stopped.

`change_table_access_mode(Tab, AccessMode) -> {aborted, R} | {atomic, ok}`

`AcccessMode` is by default the atom `read_write` but it can also be set to the atom `read_only`. If `AccessMode` is set to `read_only`, updates to the table cannot be performed. At startup, Mnesia always loads `read_only` tables locally regardless of when and if Mnesia is terminated on other nodes.

`change_table_copy_type(Tab, Node, To) -> {aborted, R} | {atomic, ok}`

For example:

```
mnesia:change_table_copy_type(person, node(), disc_copies)
```

Transforms the `person` table from a RAM table into a disc-based table at `Node`.

This function can also be used to change the storage type of the table named `schema`. The schema table can only have `ram_copies` or `disc_copies` as the storage type. If the storage type of the schema is `ram_copies`, no other table can be disc-resident on that node.

`change_table_load_order(Tab, LoadOrder) -> {aborted, R} | {atomic, ok}`

The `LoadOrder` priority is by default `0` (zero) but can be set to any integer. The tables with the highest `LoadOrder` priority are loaded first at startup.

`change_table_majority(Tab, Majority) -> {aborted, R} | {atomic, ok}`

`Majority` must be a boolean. Default is `false`. When `true`, a majority of the table replicas must be available for an update to succeed. When used on fragmented tables, `Tab` must be the base table name. Directly changing the majority setting on individual fragments is not allowed.

`clear_table(Tab) -> {aborted, R} | {atomic, ok}`

Deletes all entries in the table `Tab`.

`create_schema(DiscNodes) -> ok | {error,Reason}`

Creates a new database on disc. Various files are created in the local Mnesia directory of each node. Notice that the directory must be unique for each node. Two nodes must never share the same directory. If possible, use a local disc device to improve performance.

`mnesia:create_schema/1` fails if any of the Erlang nodes given as `DiscNodes` are not alive, if Mnesia is running on any of the nodes, or if any of the nodes already have a schema. Use `mnesia:delete_schema/1` to get rid of old faulty schemas.

Notice that only nodes with disc are to be included in `DiscNodes`. Disc-less nodes, that is, nodes where all tables including the schema only resides in RAM, must not be included.

`create_table(Name, TabDef) -> {atomic, ok} | {aborted, Reason}`

Creates a Mnesia table called `Name` according to argument `TabDef`. This list must be a list of `{Item, Value}` tuples, where the following values are allowed:

- `{access_mode, Atom}`. The access mode is by default the atom `read_write` but it can also be set to the atom `read_only`. If `AccessMode` is set to `read_only`, updates to the table cannot be performed.

  At startup, Mnesia always loads `read_only` table locally regardless of when and if Mnesia is terminated on other nodes. This argument returns the access mode of the table. The access mode can be `read_only` or `read_write`.
- `{attributes, AtomList}` is a list of the attribute names for the records that are supposed to populate the table. Default is `[key, val]`. The table must at least have one extra attribute in addition to the key.

  When accessing single attributes in a record, it is not necessary, or even recommended, to hard code any attribute names as atoms. Use construct `record_info(fields, RecordName)` instead. It can be used for records of type `RecordName`.
- `{disc_copies, Nodelist}`, where `Nodelist` is a list of the nodes where this table is supposed to have disc copies. If a table replica is of type `disc_copies`, all write operations on this particular replica of the table are written to disc and to the RAM copy of the table.

  It is possible to have a replicated table of type `disc_copies` on one node and another type on another node. Default is `[]`.
- `{disc_only_copies, Nodelist}`, where `Nodelist` is a list of the nodes where this table is supposed to have `disc_only_copies`. A disc only table replica is kept on disc only and unlike the other replica types, the contents of the replica do not reside in RAM. These replicas are considerably slower than replicas held in RAM.
- `{index, Intlist}`, where `Intlist` is a list of attribute names (atoms) or record fields for which Mnesia is to build and maintain an extra index table. The `qlc` query compiler **may** be able to optimize queries if there are indexes available.
- `{load_order, Integer}`. The load order priority is by default `0` (zero) but can be set to any integer. The tables with the highest load order priority are loaded first at startup.
- `{majority, Flag}`, where `Flag` must be a boolean. If `true`, any (non-dirty) update to the table is aborted, unless a majority of the table replicas are available for the commit. When used on a fragmented table, all fragments are given the same the same majority setting.
- `{ram_copies, Nodelist}`, where `Nodelist` is a list of the nodes where this table is supposed to have RAM copies. A table replica of type `ram_copies` is not written to disc on a per transaction basis. `ram_copies` replicas can be dumped to disc with the function `mnesia:dump_tables(Tabs)`. Default value for this attribute is `[node()]`.
- `{record_name, Name}`, where `Name` must be an atom. All records stored in the table must have this name as the first element. It defaults to the same name as the table name.
- `{snmp, SnmpStruct}`. For a description of `SnmpStruct`, see `mnesia:snmp_open_table/2`. If this attribute is present in `ArgList` to `mnesia:create_table/2`, the table is immediately accessible by SNMP. Therefore applications that use SNMP to manipulate and control the system can be designed easily, since Mnesia provides a direct mapping between the logical tables that make up an SNMP control application and the physical data that makes up a Mnesia table.
- `{storage_properties, [{Backend, Properties}]` forwards more properties to the back end storage. `Backend` can currently be `ets` or `dets`. `Properties` is a list of options sent to the back end storage during table creation. `Properties` cannot contain properties already used by Mnesia, such as `type` or `named_table`.

  For example:

```
mnesia:create_table(table, [{ram_copies, [node()]}, {disc_only_copies, nodes()},
       {storage_properties,
        [{ets, [compressed]}, {dets, [{auto_save, 5000}]} ]}])
```

- `{type, Type}`, where `Type` must be either of the atoms `set`, `ordered_set`, or `bag`. Default is `set`. In a `set`, all records have unique keys. In a `bag`, several records can have the same key, but the record content is unique. If a non-unique record is stored, the old conflicting records are overwritten.

  Notice that currently `ordered_set` is not supported for `disc_only_copies`.
- `{local_content, Bool}`, where `Bool` is `true` or `false`. Default is `false`.

For example, the following call creates the `person` table (defined earlier) and replicates it on two nodes:

```
mnesia:create_table(person,
    [{ram_copies, [N1, N2]},
     {attributes, record_info(fields, person)}]).
```

If it is required that Mnesia must build and maintain an extra index table on attribute `address` of all the `person` records that are inserted in the table, the following code would be issued:

```
mnesia:create_table(person,
    [{ram_copies, [N1, N2]},
     {index, [address]},
     {attributes, record_info(fields, person)}]).
```

The specification of `index` and `attributes` can be hard-coded as `{index, [2]}` and `{attributes, [name, age, address, salary, children]}`, respectively.

`mnesia:create_table/2` writes records into the table `schema`. This function, and all other schema manipulation functions, are implemented with the normal transaction management system. This guarantees that schema updates are performed on all nodes in an atomic manner.

`deactivate_checkpoint(Name) -> ok | {error, Reason}`

The checkpoint is automatically deactivated when some of the tables involved have no retainer attached to them. This can occur when nodes go down or when a replica is deleted. Checkpoints are also deactivated with this function. `Name` is the name of an active checkpoint.

`del_table_copy(Tab, Node) -> {aborted, R} | {atomic, ok}`

Deletes the replica of table `Tab` at node `Node`. When the last replica is deleted with this function, the table disappears entirely.

This function can also be used to delete a replica of the table named `schema`. The Mnesia node is then removed. Notice that Mnesia must be stopped on the node first.

`del_table_index(Tab, AttrName) -> {aborted, R} | {atomic, ok}`

Deletes the index on attribute with name `AttrName` in a table.

`delete({Tab, Key}) -> transaction abort | ok`

Calls `mnesia:delete(Tab, Key, write)`.

`delete(Tab, Key, LockKind) -> transaction abort | ok`

Deletes all records in table `Tab` with the key `Key`.

The semantics of this function is context-sensitive. For details, see `mnesia:activity/4`. In transaction-context, it acquires a lock of type `LockKind` in the record. Currently, the lock types `write` and `sticky_write` are supported.

`delete_object(Record) -> transaction abort | ok`

Calls `mnesia:delete_object(Tab, Record, write)`, where `Tab` is `element(1, Record)`.

`delete_object(Tab, Record, LockKind) -> transaction abort | ok`

If a table is of type `bag`, it can sometimes be needed to delete only some of the records with a certain key. This can be done with the function `delete_object/3`. A complete record must be supplied to this function.

The semantics of this function is context-sensitive. For details, see `mnesia:activity/4`. In transaction-context, it acquires a lock of type `LockKind` on the record. Currently, the lock types `write` and `sticky_write` are supported.

`delete_schema(DiscNodes) -> ok | {error,Reason}`

Deletes a database created with `mnesia:create_schema/1`. `mnesia:delete_schema/1` fails if any of the Erlang nodes given as `DiscNodes` are not alive, or if Mnesia is running on any of the nodes.

After the database is deleted, it can still be possible to start Mnesia as a disc-less node. This depends on how configuration parameter `schema_location` is set.

Warning:

Use this function with extreme caution, as it makes existing persistent data obsolete. Think twice before using it.

`delete_table(Tab) -> {aborted, Reason} | {atomic, ok}`

Permanently deletes all replicas of table `Tab`.

`dirty_all_keys(Tab) -> KeyList | exit({aborted, Reason})`

Dirty equivalent of the function `mnesia:all_keys/1`.

`dirty_delete({Tab, Key}) -> ok | exit({aborted, Reason})`

Calls `mnesia:dirty_delete(Tab, Key)`.

`dirty_delete(Tab, Key) -> ok | exit({aborted, Reason})`

Dirty equivalent of the function `mnesia:delete/3`.

`dirty_delete_object(Record)`

Calls `mnesia:dirty_delete_object(Tab, Record)`, where `Tab` is `element(1, Record)`.

`dirty_delete_object(Tab, Record)`

Dirty equivalent of the function `mnesia:delete_object/3`.

`dirty_first(Tab) -> Key | exit({aborted, Reason})`

Records in `set` or `bag` tables are not ordered. However, there is an ordering of the records that is unknown to the user. Therefore, a table can be traversed by this function with the function `mnesia:dirty_next/2`.

If there are no records in the table, this function returns the atom `'$end_of_table'`. It is therefore highly undesirable, but not disallowed, to use this atom as the key for any user records.

`dirty_index_match_object(Pattern, Pos)`

Starts `mnesia:dirty_index_match_object(Tab, Pattern, Pos)`, where `Tab` is `element(1, Pattern)`.

`dirty_index_match_object(Tab, Pattern, Pos)`

Dirty equivalent of the function `mnesia:index_match_object/4`.

`dirty_index_read(Tab, SecondaryKey, Pos)`

Dirty equivalent of the function `mnesia:index_read/3`.

`dirty_last(Tab) -> Key | exit({aborted, Reason})`

Works exactly like `mnesia:dirty_first/1` but returns the last object in Erlang term order for the `ordered_set` table type. For all other table types, `mnesia:dirty_first/1` and `mnesia:dirty_last/1` are synonyms.

`dirty_match_object(Pattern) -> RecordList | exit({aborted, Reason})`

Calls `mnesia:dirty_match_object(Tab, Pattern)`, where `Tab` is `element(1, Pattern)`.

`dirty_match_object(Tab, Pattern) -> RecordList | exit({aborted, Reason})`

Dirty equivalent of the function `mnesia:match_object/3`.

`dirty_next(Tab, Key) -> Key | exit({aborted, Reason})`

Traverses a table and performs operations on all records in the table. When the end of the table is reached, the special key `'$end_of_table'` is returned. Otherwise, the function returns a key that can be used to read the actual record. The behavior is undefined if another Erlang process performs write operations on the table while it is being traversed with the function `mnesia:dirty_next/2`.

`dirty_prev(Tab, Key) -> Key | exit({aborted, Reason})`

Works exactly like `mnesia:dirty_next/2` but returns the previous object in Erlang term order for the `ordered_set` table type. For all other table types, `mnesia:dirty_next/2` and `mnesia:dirty_prev/2` are synonyms.

`dirty_read({Tab, Key}) -> ValueList | exit({aborted, Reason}`

Calls `mnesia:dirty_read(Tab, Key)`.

`dirty_read(Tab, Key) -> ValueList | exit({aborted, Reason}`

Dirty equivalent of the function `mnesia:read/3`.

`dirty_select(Tab, MatchSpec) -> ValueList | exit({aborted, Reason}`

Dirty equivalent of the function `mnesia:select/2`.

`dirty_slot(Tab, Slot) -> RecordList | exit({aborted, Reason})`

Traverses a table in a manner similar to the function `mnesia:dirty_next/2`. A table has a number of slots that range from 0 (zero) to an unknown upper bound. The function `mnesia:dirty_slot/2` returns the special atom

`'$end_of_table'` when the end of the table is reached. The behavior of this function is undefined if a write operation is performed on the table while it is being traversed.

`dirty_update_counter({Tab, Key}, Incr) -> NewVal | exit({aborted, Reason})`

Calls `mnesia:dirty_update_counter(Tab, Key, Incr)`.

`dirty_update_counter(Tab, Key, Incr) -> NewVal | exit({aborted, Reason})`

Mnesia has no special counter records. However, records of the form `{Tab, Key, Integer}` can be used as (possibly disc-resident) counters when `Tab` is a `set`. This function updates a counter with a positive or negative number. However, counters can never become less than zero. There are two significant differences between this function and the action of first reading the record, performing the arithmetics, and then writing the record:

- It is much more efficient.
- `mnesia:dirty_update_counter/3` is performed as an atomic operation although it is not protected by a transaction.

If two processes perform `mnesia:dirty_update_counter/3` simultaneously, both updates take effect without the risk of losing one of the updates. The new value `NewVal` of the counter is returned.

If `Key` do not exists, a new record is created with value `Incr` if it is larger than 0, otherwise it is set to 0.

`dirty_write(Record) -> ok | exit({aborted, Reason})`

Calls `mnesia:dirty_write(Tab, Record)`, where `Tab` is `element(1, Record)`.

`dirty_write(Tab, Record) -> ok | exit({aborted, Reason})`

Dirty equivalent of the function `mnesia:write/3`.

`dump_log() -> dumped`

Performs a user-initiated dump of the local log file. This is usually not necessary, as Mnesia by default manages this automatically. See configuration parameters *dump_log_time_threshold* and *dump_log_write_threshold*.

`dump_tables(TabList) -> {atomic, ok} | {aborted, Reason}`

Dumps a set of `ram_copies` tables to disc. The next time the system is started, these tables are initiated with the data found in the files that are the result of this dump. None of the tables can have disc-resident replicas.

`dump_to_textfile(Filename)`

Dumps all local tables of a Mnesia system into a text file, which can be edited (by a normal text editor) and then be reloaded with `mnesia:load_textfile/1`. Only use this function for educational purposes. Use other functions to deal with real backups.

`error_description(Error) -> String`

All Mnesia transactions, including all the schema update functions, either return value `{atomic, Val}` or the tuple `{aborted, Reason}`. `Reason` can be either of the atoms in the following list. The function `error_description/1` returns a descriptive string that describes the error.

- `nested_transaction`. Nested transactions are not allowed in this context.
- `badarg`. Bad or invalid argument, possibly bad type.
- `no_transaction`. Operation not allowed outside transactions.

- `combine_error`. Table options illegally combined.
- `bad_index`. Index already exists, or was out of bounds.
- `already_exists`. Schema option to be activated is already on.
- `index_exists`. Some operations cannot be performed on tables with an index.
- `no_exists`. Tried to perform operation on non-existing (not-alive) item.
- `system_limit`. A system limit was exhausted.
- `mnesia_down`. A transaction involves records on a remote node, which became unavailable before the transaction was completed. Records are no longer available elsewhere in the network.
- `not_a_db_node`. A node was mentioned that does not exist in the schema.
- `bad_type`. Bad type specified in argument.
- `node_not_running`. Node is not running.
- `truncated_binary_file`. Truncated binary in file.
- `active`. Some delete operations require that all active records are removed.
- `illegal`. Operation not supported on this record.

`Error` can be `Reason`, `{error, Reason}`, or `{aborted, Reason}`. `Reason` can be an atom or a tuple with `Reason` as an atom in the first field.

The following examples illustrate a function that returns an error, and the method to retrieve more detailed error information:

- The function *mnesia:create_table(bar, [{attributes, 3.14}])* returns the tuple `{aborted,Reason}`, where `Reason` is the tuple `{bad_type,bar,3.14000}`.
- The function *mnesia:error_description(Reason)* returns the term `{"Bad type on some provided arguments",bar,3.14000}`, which is an error description suitable for display.

`ets(Fun, [, Args]) -> ResultOfFun | exit(Reason)`

Calls the `Fun` in a raw context that is not protected by a transaction. The Mnesia function call is performed in the `Fun` and performed directly on the local ETS tables on the assumption that the local storage type is `ram_copies` and the tables are not replicated to other nodes. Subscriptions are not triggered and checkpoints are not updated, but it is extremely fast. This function can also be applied to `disc_copies` tables if all operations are read only. For details, see `mnesia:activity/4` and the User's Guide.

Notice that calling (nesting) a `mnesia:ets` inside a transaction-context inherits the transaction semantics.

`first(Tab) -> Key | transaction abort`

Records in `set` or `bag` tables are not ordered. However, there is an ordering of the records that is unknown to the user. A table can therefore be traversed by this function with the function `mnesia:next/2`.

If there are no records in the table, this function returns the atom `'$end_of_table'`. It is therefore highly undesirable, but not disallowed, to use this atom as the key for any user records.

`foldl(Function, Acc, Table) -> NewAcc | transaction abort`

Iterates over the table `Table` and calls `Function(Record, NewAcc)` for each `Record` in the table. The term returned from `Function` is used as the second argument in the next call to `Function`.

`foldl` returns the same term as the last call to `Function` returned.

`foldr(Function, Acc, Table) -> NewAcc | transaction abort`

Works exactly like `foldl/3` but iterates the table in the opposite order for the `ordered_set` table type. For all other table types, `foldr/3` and `foldl/3` are synonyms.

`force_load_table(Tab) -> yes | ErrorDescription`

The Mnesia algorithm for table load can lead to a situation where a table cannot be loaded. This situation occurs when a node is started and Mnesia concludes, or suspects, that another copy of the table was active after this local copy became inactive because of a system crash.

If this situation is not acceptable, this function can be used to override the strategy of the Mnesia table load algorithm. This can lead to a situation where some transaction effects are lost with an inconsistent database as result, but for some applications high availability is more important than consistent data.

`index_match_object(Pattern, Pos) -> transaction abort | ObjList`

Starts `mnesia:index_match_object(Tab, Pattern, Pos, read)`, where `Tab` is `element(1, Pattern)`.

`index_match_object(Tab, Pattern, Pos, LockKind) -> transaction abort | ObjList`

In a manner similar to the function `mnesia:index_read/3`, any index information can be used when trying to match records. This function takes a pattern that obeys the same rules as the function `mnesia:match_object/3`, except that this function requires the following conditions:

- The table `Tab` must have an index on position `Pos`.
- The element in position `Pos` in `Pattern` must be bound. `Pos` is an integer (`#record.Field`) or an attribute name.

The two index search functions described here are automatically started when searching tables with `qlc` list comprehensions and also when using the low-level `mnesia:[dirty_]match_object` functions.

The semantics of this function is context-sensitive. For details, see `mnesia:activity/4`. In transaction-context, it acquires a lock of type `LockKind` on the entire table or on a single record. Currently, the lock type `read` is supported.

`index_read(Tab, SecondaryKey, Pos) -> transaction abort | RecordList`

Assume that there is an index on position `Pos` for a certain record type. This function can be used to read the records without knowing the actual key for the record. For example, with an index in position 1 of table `person`, the call `mnesia:index_read(person, 36, #person.age)` returns a list of all persons with age 36. `Pos` can also be an attribute name (atom), but if the notation `mnesia:index_read(person, 36, age)` is used, the field position is searched for in runtime, for each call.

The semantics of this function is context-sensitive. For details, see `mnesia:activity/4`. In transaction-context, it acquires a read lock on the entire table.

`info() -> ok`

Prints system information on the terminal. This function can be used even if Mnesia is not started. However, more information is displayed if Mnesia is started.

`install_fallback(Opaque) -> ok | {error,Reason}`

Calls `mnesia:install_fallback(Opaque, Args)`, where `Args` is `[{scope, global}]`.

`install_fallback(Opaque), BackupMod) -> ok | {error,Reason}`

Calls `mnesia:install_fallback(Opaque, Args)`, where `Args` is `[{scope, global}, {module, BackupMod}]`.

`install_fallback(Opaque, Args) -> ok | {error,Reason}`

Installs a backup as fallback. The fallback is used to restore the database at the next startup. Installation of fallbacks requires Erlang to be operational on all the involved nodes, but it does not matter if Mnesia is running or not. The installation of the fallback fails if the local node is not one of the disc-resident nodes in the backup.

`Args` is a list of the following tuples:

- `{module, BackupMod}`. All accesses of the backup media are performed through a callback module named `BackupMod`. Argument `Opaque` is forwarded to the callback module, which can interpret it as it wishes. The default callback module is called `mnesia_backup` and it interprets argument `Opaque` as a local filename. The default for this module is also configurable through configuration parameter `-mnesia mnesia_backup`.
- `{scope, Scope}`. The `Scope` of a fallback is either `global` for the entire database or `local` for one node. By default, the installation of a fallback is a global operation, which either is performed on all nodes with a disc-resident schema or none. Which nodes that are disc-resident is determined from the schema information in the backup.

  If `Scope` of the operation is `local`, the fallback is only installed on the local node.
- `{mnesia_dir, AlternateDir}`. This argument is only valid if the scope of the installation is `local`. Normally the installation of a fallback is targeted to the Mnesia directory, as configured with configuration parameter `-mnesia dir`. But by explicitly supplying an `AlternateDir`, the fallback is installed there regardless of the Mnesia directory configuration parameter setting. After installation of a fallback on an alternative Mnesia directory, that directory is fully prepared for use as an active Mnesia directory.

  This is a dangerous feature that must be used with care. By unintentional mixing of directories, you can easily end up with an inconsistent database, if the same backup is installed on more than one directory.

`is_transaction() -> boolean`

When this function is executed inside a transaction-context, it returns `true`, otherwise `false`.

`last(Tab) -> Key | transaction abort`

Works exactly like `mnesia:first/1`, but returns the last object in Erlang term order for the `ordered_set` table type. For all other table types, `mnesia:first/1` and `mnesia:last/1` are synonyms.

`load_textfile(Filename)`

Loads a series of definitions and data found in the text file (generated with `mnesia:dump_to_textfile/1`) into Mnesia. This function also starts Mnesia and possibly creates a new schema. This function is intended for educational purposes only. It is recommended to use other functions to deal with real backups.

`lock(LockItem, LockKind) -> Nodes | ok | transaction abort`

Write locks are normally acquired on all nodes where a replica of the table resides (and is active). Read locks are acquired on one node (the local node if a local replica exists). Most of the context-sensitive access functions acquire an implicit lock if they are started in a transaction-context. The granularity of a lock can either be a single record or an entire table.

The normal use is to call the function without checking the return value, as it exits if it fails and the transaction is restarted by the transaction manager. It returns all the locked nodes if a write lock is acquired and `ok` if it was a read lock.

The function `mnesia:lock/2` is intended to support explicit locking on tables, but is also intended for situations when locks need to be acquired regardless of how tables are replicated. Currently, two kinds of `LockKind` are supported:

`write`

Write locks are exclusive. This means that if one transaction manages to acquire a write lock on an item, no other transaction can acquire any kind of lock on the same item.

`read`

Read locks can be shared. This means that if one transaction manages to acquire a read lock on an item, other transactions can also acquire a read lock on the same item. However, if someone has a read lock, no one can acquire a write lock at the same item. If someone has a write lock, no one can acquire either a read lock or a write lock at the same item.

Conflicting lock requests are automatically queued if there is no risk of a deadlock. Otherwise the transaction must be terminated and executed again. Mnesia does this automatically as long as the upper limit of the maximum `retries` is not reached. For details, see `mnesia:transaction/3`.

For the sake of completeness, sticky write locks are also described here even if a sticky write lock is not supported by this function:

`sticky_write`

Sticky write locks are a mechanism that can be used to optimize write lock acquisition. If your application uses replicated tables mainly for fault tolerance (as opposed to read access optimization purpose), sticky locks can be the best option available.

When a sticky write lock is acquired, all nodes are informed which node is locked. Then, sticky lock requests from the same node are performed as a local operation without any communication with other nodes. The sticky lock lingers on the node even after the transaction ends. For details, see the User's Guide.

Currently, this function supports two kinds of `LockItem`:

`{table, Tab}`

This acquires a lock of type `LockKind` on the entire table `Tab`.

`{global, GlobalKey, Nodes}`

This acquires a lock of type `LockKind` on the global resource `GlobalKey`. The lock is acquired on all active nodes in the `Nodes` list.

Locks are released when the outermost transaction ends.

The semantics of this function is context-sensitive. For details, see `mnesia:activity/4`. In transaction-context, it acquires locks, otherwise it ignores the request.

`match_object(Pattern) -> transaction abort | RecList`

Calls `mnesia:match_object(Tab, Pattern, read)`, where `Tab` is `element(1, Pattern)`.

`match_object(Tab, Pattern, LockKind) -> transaction abort | RecList`

Takes a pattern with "don't care" variables denoted as a `'_'` parameter. This function returns a list of records that matched the pattern. Since the second element of a record in a table is considered to be the key for the record, the performance of this function depends on whether this key is bound or not.

For example, the call `mnesia:match_object(person, {person, '_', 36, '_', '_'}, read)` returns a list of all person records with an `age` field of 36.

The function `mnesia:match_object/3` automatically uses indexes if these exist. However, no heuristics are performed to select the best index.

The semantics of this function is context-sensitive. For details, see `mnesia:activity/4`. In transaction-context, it acquires a lock of type `LockKind` on the entire table or a single record. Currently, the lock type `read` is supported.

```
move_table_copy(Tab, From, To) -> {aborted, Reason} | {atomic, ok}
```

Moves the copy of table `Tab` from node `From` to node `To`.

The storage type is preserved. For example, a RAM table moved from one node remains a RAM on the new node. Other transactions can still read and write in the table while it is being moved.

This function cannot be used on `local_content` tables.

```
next(Tab, Key) -> Key | transaction abort
```

Traverses a table and performs operations on all records in the table. When the end of the table is reached, the special key `'$end_of_table'` is returned. Otherwise the function returns a key that can be used to read the actual record.

```
prev(Tab, Key) -> Key | transaction abort
```

Works exactly like `mnesia:next/2`, but returns the previous object in Erlang term order for the `ordered_set` table type. For all other table types, `mnesia:next/2` and `mnesia:prev/2` are synonyms.

```
read({Tab, Key}) -> transaction abort | RecordList
```

Calls function `mnesia:read(Tab, Key, read)`.

```
read(Tab, Key) -> transaction abort | RecordList
```

Calls function `mnesia:read(Tab, Key, read)`.

```
read(Tab, Key, LockKind) -> transaction abort | RecordList
```

Reads all records from table `Tab` with key `Key`. This function has the same semantics regardless of the location of `Tab`. If the table is of type `bag`, the function `mnesia:read(Tab, Key)` can return an arbitrarily long list. If the table is of type `set`, the list is either of length 1, or `[]`.

The semantics of this function is context-sensitive. For details, see `mnesia:activity/4`. In transaction-context, it acquires a lock of type `LockKind`. Currently, the lock types `read`, `write`, and `sticky_write` are supported.

If the user wants to update the record, it is more efficient to use `write/sticky_write` as the `LockKind`. If majority checking is active on the table, it is checked as soon as a write lock is attempted. This can be used to end quickly if the majority condition is not met.

```
read_lock_table(Tab) -> ok | transaction abort
```

Calls the function `mnesia:lock({table, Tab}, read)`.

```
report_event(Event) -> ok
```

When tracing a system of Mnesia applications it is useful to be able to interleave Mnesia own events with application-related events that give information about the application context.

Whenever the application begins a new and demanding Mnesia task, or if it enters a new interesting phase in its execution, it can be a good idea to use `mnesia:report_event/1`. `Event` can be any term and generates a `{mnesia_user, Event}` event for any processes that subscribe to Mnesia system events.

`restore(Opaque, Args) -> {atomic, RestoredTabs} |{aborted, Reason}`

With this function, tables can be restored online from a backup without restarting Mnesia. `Opaque` is forwarded to the backup module. `Args` is a list of the following tuples:

- `{module,BackupMod}`. The backup module `BackupMod` is used to access the backup media. If omitted, the default backup module is used.
- `{skip_tables, TabList}`, where `TabList` is a list of tables that is not to be read from the backup.
- `{clear_tables, TabList}`, where `TabList` is a list of tables that is to be cleared before the records from the backup are inserted. That is, all records in the tables are deleted before the tables are restored. Schema information about the tables is not cleared or read from the backup.
- `{keep_tables, TabList}`, where `TabList` is a list of tables that is not to be cleared before the records from the backup are inserted. That is, the records in the backup are added to the records in the table. Schema information about the tables is not cleared or read from the backup.
- `{recreate_tables, TabList}`, where `TabList` is a list of tables that is to be recreated before the records from the backup are inserted. The tables are first deleted and then created with the schema information from the backup. All the nodes in the backup need to be operational.
- `{default_op, Operation}`, where `Operation` is either of the operations `skip_tables`, `clear_tables`, `keep_tables`, or `recreate_tables`. The default operation specifies which operation that is to be used on tables from the backup that is not specified in any of the mentioned lists. If omitted, operation `clear_tables` is used.

The affected tables are write-locked during the restoration. However, regardless of the lock conflicts caused by this, the applications can continue to do their work while the restoration is being performed. The restoration is performed as one single transaction.

If the database is huge, it it not always possible to restore it online. In such cases, restore the old database by installing a fallback and then restart.

`s_delete({Tab, Key}) -> ok | transaction abort`

Calls the function `mnesia:delete(Tab, Key, sticky_write)`

`s_delete_object(Record) -> ok | transaction abort`

Calls the function `mnesia:delete_object(Tab, Record, sticky_write)`, where `Tab` is `element(1, Record)`.

`s_write(Record) -> ok | transaction abort`

Calls the function `mnesia:write(Tab, Record, sticky_write)`, where `Tab` is `element(1, Record)`.

`schema() -> ok`

Prints information about all table definitions on the terminal.

`schema(Tab) -> ok`

Prints information about one table definition on the terminal.

`select(Tab, MatchSpec [, Lock]) -> transaction abort | [Object]`

Matches the objects in table `Tab` using a `match_spec` as described in the *ets:select/3*. Optionally a lock `read` or `write` can be given as the third argument. Default is `read`. The return value depends on `MatchSpec`.

Notice that for best performance, `select` is to be used before any modifying operations are done on that table in the same transaction. That is, do not use `write` or `delete` before a `select`.

In its simplest forms, the `match_spec` look as follows:

- `MatchSpec = [MatchFunction]`
- `MatchFunction = {MatchHead, [Guard], [Result]}`
- `MatchHead = tuple() | record()`
- `Guard = {"Guardtest name", ...}`
- `Result = "Term construct"`

For a complete description of `select`, see the *ERTS* User's Guide and the *ets* manual page in STDLIB.

For example, to find the names of all male persons older than 30 in table `Tab`:

```
MatchHead = #person{name='$1', sex=male, age='$2', _='_'},
Guard = {'>', '$2', 30},
Result = '$1',
mnesia:select(Tab,[{MatchHead, [Guard], [Result]}]),
```

`select(Tab, MatchSpec, NObjects, Lock) -> transaction abort | {[Object],Cont} | '$end_of_table'`

Matches the objects in table `Tab` using a `match_spec` as described in the *ERTS* User's Guide, and returns a chunk of terms and a continuation. The wanted number of returned terms is specified by argument `NObjects`. The lock argument can be `read` or `write`. The continuation is to be used as argument to `mnesia:select/1`, if more or all answers are needed.

Notice that for best performance, `select` is to be used before any modifying operations are done on that table in the same transaction. That is, do not use `mnesia:write` or `mnesia:delete` before a `mnesia:select`. For efficiency, `NObjects` is a recommendation only and the result can contain anything from an empty list to all available results.

`select(Cont) -> transaction abort | {[Object],Cont} | '$end_of_table'`

Selects more objects with the match specification initiated by `mnesia:select/4`.

Notice that any modifying operations, that is, `mnesia:write` or `mnesia:delete`, that are done between the `mnesia:select/4` and `mnesia:select/1` calls are not visible in the result.

`set_debug_level(Level) -> OldLevel`

Changes the internal debug level of Mnesia. For details, see *Section Configuration Parameters*.

`set_master_nodes(MasterNodes) -> ok | {error, Reason}`

For each table Mnesia determines its replica nodes (`TabNodes`) and starts `mnesia:set_master_nodes(Tab, TabMasterNodes)`. where `TabMasterNodes` is the intersection of `MasterNodes` and `TabNodes`. For semantics, see `mnesia:set_master_nodes/2`.

`set_master_nodes(Tab, MasterNodes) -> ok | {error, Reason}`

If the application detects a communication failure (in a potentially partitioned network) that can have caused an inconsistent database, it can use the function `mnesia:set_master_nodes(Tab, MasterNodes)` to define from which nodes each table is to be loaded. At startup, the Mnesia normal table load algorithm is bypassed and the table is loaded from one of the master nodes defined for the table, regardless of when and if Mnesia terminated on other nodes. `MasterNodes` can only contain nodes where the table has a replica. If the `MasterNodes` list is empty,

the master node recovery mechanism for the particular table is reset, and the normal load mechanism is used at the next restart.

The master node setting is always local. It can be changed regardless if Mnesia is started or not.

The database can also become inconsistent if configuration parameter `max_wait_for_decision` is used or if `mnesia:force_load_table/1` is used.

`snmp_close_table(Tab) -> {aborted, R} | {atomic, ok}`

Removes the possibility for SNMP to manipulate the table.

`snmp_get_mnesia_key(Tab, RowIndex) -> {ok, Key} | undefined`

Types:

```
Tab ::= atom()
RowIndex ::= [integer()]
Key ::= key() | {key(), key(), ...}
key() ::= integer() | string() | [integer()]
```

Transforms an SNMP index to the corresponding Mnesia key. If the SNMP table has multiple keys, the key is a tuple of the key columns.

`snmp_get_next_index(Tab, RowIndex) -> {ok, NextIndex} | endOfTable`

Types:

```
Tab ::= atom()
RowIndex ::= [integer()]
NextIndex ::= [integer()]
```

`RowIndex` can specify a non-existing row. Specifically, it can be the empty list. Returns the index of the next lexicographical row. If `RowIndex` is the empty list, this function returns the index of the first row in the table.

`snmp_get_row(Tab, RowIndex) -> {ok, Row} | undefined`

Types:

```
Tab ::= atom()
RowIndex ::= [integer()]
Row ::= record(Tab)
```

Reads a row by its SNMP index. This index is specified as an SNMP Object Identifier, a list of integers.

`snmp_open_table(Tab, SnmpStruct) -> {aborted, R} | {atomic, ok}`

Types:

```
Tab ::= atom()
SnmpStruct ::= [{key, type()}]
type() ::= type_spec() | {type_spec(), type_spec(), ...}
type_spec() ::= fix_string | string | integer
```

A direct one-to-one mapping can be established between Mnesia tables and SNMP tables. Many telecommunication applications are controlled and monitored by the SNMP protocol. This connection between Mnesia and SNMP makes it simple and convenient to achieve this mapping.

Argument `SnmpStruct` is a list of SNMP information. Currently, the only information needed is information about the key types in the table. Multiple keys cannot be handled in Mnesia, but many SNMP tables have multiple keys.

Therefore, the following convention is used: if a table has multiple keys, these must always be stored as a tuple of the keys. Information about the key types is specified as a tuple of atoms describing the types. The only significant type is `fix_string`. This means that a string has a fixed size.

For example, the following causes table `person` to be ordered as an SNMP table:

```
mnesia:snmp_open_table(person, [{key, string}])
```

Consider the following schema for a table of company employees. Each employee is identified by department number and name. The other table column stores the telephone number:

```
mnesia:create_table(employee,
    [{snmp, [{key, {integer, string}}]},
     {attributes, record_info(fields, employees)}]),
```

The corresponding SNMP table would have three columns: `department`, `name`, and `telno`.

An option is to have table columns that are not visible through the SNMP protocol. These columns must be the last columns of the table. In the previous example, the SNMP table could have columns `department` and `name` only. The application could then use column `telno` internally, but it would not be visible to the SNMP managers.

In a table monitored by SNMP, all elements must be integers, strings, or lists of integers.

When a table is SNMP ordered, modifications are more expensive than usual, O(logN). Also, more memory is used.

Notice that only the lexicographical SNMP ordering is implemented in Mnesia, not the actual SNMP monitoring.

`start() -> ok | {error, Reason}`

Mnesia startup is asynchronous. The function call `mnesia:start()` returns the atom `ok` and then starts to initialize the different tables. Depending on the size of the database, this can take some time, and the application programmer must wait for the tables that the application needs before they can be used. This is achieved by using the function `mnesia:wait_for_tables/2`.

The startup procedure for a set of Mnesia nodes is a fairly complicated operation. A Mnesia system consists of a set of nodes, with Mnesia started locally on all participating nodes. Normally, each node has a directory where all the Mnesia files are written. This directory is referred to as the Mnesia directory. Mnesia can also be started on disc-less nodes. For more information about disc-less nodes, see `mnesia:create_schema/1` and the User's Guide.

The set of nodes that makes up a Mnesia system is kept in a schema. Mnesia nodes can be added to or removed from the schema. The initial schema is normally created on disc with the function `mnesia:create_schema/1`. On disc-less nodes, a tiny default schema is generated each time Mnesia is started. During the startup procedure, Mnesia exchanges schema information between the nodes to verify that the table definitions are compatible.

Each schema has a unique cookie, which can be regarded as a unique schema identifier. The cookie must be the same on all nodes where Mnesia is supposed to run. For details, see the User's Guide.

The schema file and all other files that Mnesia needs are kept in the Mnesia directory. The command-line option `-mnesia dir Dir` can be used to specify the location of this directory to the Mnesia system. If no such command-line option is found, the name of the directory defaults to `Mnesia.Node`.

`application:start(mnesia)` can also be used.

`stop() -> stopped`

Stops Mnesia locally on the current node.

`application:stop(mnesia)` can also be used.

`subscribe(EventCategory) -> {ok, Node} | {error, Reason}`

Ensures that a copy of all events of type `EventCategory` is sent to the caller. The available event types are described in the *User's Guide*.

`sync_dirty(Fun, [, Args]) -> ResultOfFun | exit(Reason)`

Calls the `Fun` in a context that is not protected by a transaction. The Mnesia function calls performed in the `Fun` are mapped to the corresponding dirty functions. It is performed in almost the same context as `mnesia:async_dirty/1,2`. The difference is that the operations are performed synchronously. The caller waits for the updates to be performed on all active replicas before the `Fun` returns. For details, see `mnesia:activity/4` and the User's Guide.

`sync_log() -> ok | {error, Reason}`

Ensures that the local transaction log file is synced to disk. On a single node system, data written to disk tables since the last dump can be lost if there is a power outage. See *dump_log/0*.

`sync_transaction(Fun, [[, Args], Retries]) -> {aborted, Reason} | {atomic, ResultOfFun}`

Waits until data have been committed and logged to disk (if disk is used) on every involved node before it returns, otherwise it behaves as `mnesia:transaction/[1,2,3]`.

This functionality can be used to avoid that one process overloads a database on another node.

`system_info(InfoKey) -> Info | exit({aborted, Reason})`

Returns information about the Mnesia system, such as transaction statistics, `db_nodes`, and configuration parameters. The valid keys are as follows:

- `all`. Returns a list of all local system information. Each element is a `{InfoKey, InfoVal}` tuple.

  New `InfoKey`s can be added and old undocumented `InfoKey`s can be removed without notice.
- `access_module`. Returns the name of module that is configured to be the activity access callback module.
- `auto_repair`. Returns `true` or `false` to indicate if Mnesia is configured to start the auto-repair facility on corrupted disc files.
- `backup_module`. Returns the name of the module that is configured to be the backup callback module.
- `checkpoints`. Returns a list of the names of the checkpoints currently active on this node.
- `event_module`. Returns the name of the module that is the event handler callback module.
- `db_nodes`. Returns the nodes that make up the persistent database. Disc-less nodes are only included in the list of nodes if they explicitly have been added to the schema, for example, with `mnesia:add_table_copy/3`. The function can be started even if Mnesia is not yet running.
- `debug`. Returns the current debug level of Mnesia.
- `directory`. Returns the name of the Mnesia directory. It can be called even if Mnesia is not yet running.
- `dump_log_load_regulation`. Returns a boolean that tells if Mnesia is configured to regulate the dumper process load.

  This feature is temporary and will be removed in future releases.
- `dump_log_time_threshold`. Returns the time threshold for transaction log dumps in milliseconds.
- `dump_log_update_in_place`. Returns a boolean that tells if Mnesia is configured to perform the updates in the Dets files directly, or if the updates are to be performed in a copy of the Dets files.
- `dump_log_write_threshold`. Returns the write threshold for transaction log dumps as the number of writes to the transaction log.

- `extra_db_nodes`. Returns a list of extra `db_nodes` to be contacted at startup.
- `fallback_activated`. Returns `true` if a fallback is activated, otherwise `false`.
- `held_locks`. Returns a list of all locks held by the local Mnesia lock manager.
- `is_running`. Returns `yes` or `no` to indicate if Mnesia is running. It can also return `starting` or `stopping`. Can be called even if Mnesia is not yet running.
- `local_tables`. Returns a list of all tables that are configured to reside locally.
- `lock_queue`. Returns a list of all transactions that are queued for execution by the local lock manager.
- `log_version`. Returns the version number of the Mnesia transaction log format.
- `master_node_tables`. Returns a list of all tables with at least one master node.
- `protocol_version`. Returns the version number of the Mnesia inter-process communication protocol.
- `running_db_nodes`. Returns a list of nodes where Mnesia currently is running. This function can be called even if Mnesia is not yet running, but it then has slightly different semantics.

  If Mnesia is down on the local node, the function returns those other `db_nodes` and `extra_db_nodes` that for the moment are operational.

  If Mnesia is started, the function returns those nodes that Mnesia on the local node is fully connected to. Only those nodes that Mnesia has exchanged schema information with are included as `running_db_nodes`. After the merge of schemas, the local Mnesia system is fully operable and applications can perform access of remote replicas. Before the schema merge, Mnesia only operates locally. Sometimes there are more nodes included in the `running_db_nodes` list than all `db_nodes` and `extra_db_nodes` together.
- `schema_location`. Returns the initial schema location.
- `subscribers`. Returns a list of local processes currently subscribing to system events.
- `tables`. Returns a list of all locally known tables.
- `transactions`. Returns a list of all currently active local transactions.
- `transaction_failures`. Returns a number that indicates how many transactions have failed since Mnesia was started.
- `transaction_commits`. Returns a number that indicates how many transactions have terminated successfully since Mnesia was started.
- `transaction_restarts`. Returns a number that indicates how many transactions have been restarted since Mnesia was started.
- `transaction_log_writes`. Returns a number that indicates how many write operations that have been performed to the transaction log since startup.
- `use_dir`. Returns a boolean that indicates if the Mnesia directory is used or not. Can be started even if Mnesia is not yet running.
- `version`. Returns the current version number of Mnesia.

`table(Tab [,[Option]]) -> QueryHandle`

Returns a Query List Comprehension (QLC) query handle, see the *qlc(3)* manual page in STDLIB. The module `qlc` implements a query language that can use Mnesia tables as sources of data. Calling `mnesia:table/1,2` is the means to make the `mnesia` table `Tab` usable to QLC.

`Option` can contain Mnesia options or QLC options. Mnesia recognizes the following options (any other option is forwarded to QLC).

- `{lock, Lock}`, where `lock` can be `read` or `write`. Default is `read`.
- `{n_objects,Number}`, where `n_objects` specifies (roughly) the number of objects returned from Mnesia to QLC. Queries to remote tables can need a larger chunk to reduce network overhead. By default, `100` objects at a time are returned.

- `{traverse, SelectMethod}`, where `traverse` determines the method to traverse the whole table (if needed). The default method is `select`.

There are two alternatives for `select`:

- `select`. The table is traversed by calling `mnesia:select/4` and `mnesia:select/1`. The match specification (the second argument of `select/3`) is assembled by QLC: simple filters are translated into equivalent match specifications. More complicated filters need to be applied to all objects returned by `select/3` given a match specification that matches all objects.
- `{select, MatchSpec}`. As for `select`, the table is traversed by calling `mnesia:select/3` and `mnesia:select/1`. The difference is that the match specification is explicitly given. This is how to state match specifications that cannot easily be expressed within the syntax provided by QLC.

`table_info(Tab, InfoKey) -> Info | exit({aborted, Reason})`

The `table_info/2` function takes two arguments. The first is the name of a Mnesia table. The second is one of the following keys:

- `all`. Returns a list of all local table information. Each element is a `{InfoKey, ItemVal}` tuple.

  New `InfoItem`s can be added and old undocumented `InfoItem`s can be removed without notice.
- `access_mode`. Returns the access mode of the table. The access mode can be `read_only` or `read_write`.
- `arity`. Returns the arity of records in the table as specified in the schema.
- `attributes`. Returns the table attribute names that are specified in the schema.
- `checkpoints`. Returns the names of the currently active checkpoints, which involve this table on this node.
- `cookie`. Returns a table cookie, which is a unique system-generated identifier for the table. The cookie is used internally to ensure that two different table definitions using the same table name cannot accidentally be intermixed. The cookie is generated when the table is created initially.
- `disc_copies`. Returns the nodes where a `disc_copy` of the table resides according to the schema.
- `disc_only_copies`. Returns the nodes where a `disc_only_copy` of the table resides according to the schema.
- `index`. Returns the list of index position integers for the table.
- `load_node`. Returns the name of the node that Mnesia loaded the table from. The structure of the returned value is unspecified, but can be useful for debugging purposes.
- `load_order`. Returns the load order priority of the table. It is an integer and defaults to `0` (zero).
- `load_reason`. Returns the reason of why Mnesia decided to load the table. The structure of the returned value is unspecified, but can be useful for debugging purposes.
- `local_content`. Returns `true` or `false` to indicate if the table is configured to have locally unique content on each node.
- `master_nodes`. Returns the master nodes of a table.
- `memory`. Returns the number of words allocated to the table on this node.
- `ram_copies`. Returns the nodes where a `ram_copy` of the table resides according to the schema.
- `record_name`. Returns the record name, common for all records in the table.
- `size`. Returns the number of records inserted in the table.
- `snmp`. Returns the SNMP struct. `[]` means that the table currently has no SNMP properties.
- `storage_type`. Returns the local storage type of the table. It can be `disc_copies`, `ram_copies`, `disc_only_copies`, or the atom `unknown`. `unknown` is returned for all tables that only reside remotely.
- `subscribers`. Returns a list of local processes currently subscribing to local table events that involve this table on this node.
- `type`. Returns the table type, which is `bag`, `set`, or `ordered_set`.

- `user_properties`. Returns the user-associated table properties of the table. It is a list of the stored property records.
- `version`. Returns the current version of the table definition. The table version is incremented when the table definition is changed. The table definition can be incremented directly when it has been changed in a schema transaction, or when a committed table definition is merged with table definitions from other nodes during startup.
- `where_to_read`. Returns the node where the table can be read. If value `nowhere` is returned, either the table is not loaded or it resides at a remote node that is not running.
- `where_to_write`. Returns a list of the nodes that currently hold an active replica of the table.
- `wild_pattern`. Returns a structure that can be given to the various match functions for a certain table. A record tuple is where all record fields have value `'_'`.

`transaction(Fun [[, Args], Retries]) -> {aborted, Reason} | {atomic, ResultOfFun}`

Executes the functional object `Fun` with arguments `Args` as a transaction.

The code that executes inside the transaction can consist of a series of table manipulation functions. If something goes wrong inside the transaction as a result of a user error or a certain table not being available, the entire transaction is terminated and the function `transaction/1` returns the tuple `{aborted, Reason}`.

If all is going well, `{atomic, ResultOfFun}` is returned, where `ResultOfFun` is the value of the last expression in `Fun`.

A function that adds a family to the database can be written as follows if there is a structure `{family, Father, Mother, ChildrenList}`:

```
add_family({family, F, M, Children}) ->
    ChildOids = lists:map(fun oid/1, Children),
    Trans = fun() ->
        mnesia:write(F#person{children = ChildOids},
        mnesia:write(M#person{children = ChildOids},
        Write = fun(Child) -> mnesia:write(Child) end,
        lists:foreach(Write, Children)
    end,
    mnesia:transaction(Trans).

oid(Rec) -> {element(1, Rec), element(2, Rec)}.
```

This code adds a set of people to the database. Running this code within one transaction ensures that either the whole family is added to the database, or the whole transaction terminates. For example, if the last child is badly formatted, or the executing process terminates because of an `'EXIT'` signal while executing the family code, the transaction terminates. Thus, the situation where half a family is added can never occur.

It is also useful to update the database within a transaction if several processes concurrently update the same records. For example, the function `raise(Name, Amount)`, which adds `Amount` to the salary field of a person, is to be implemented as follows:

```
raise(Name, Amount) ->
    mnesia:transaction(fun() ->
        case mnesia:wread({person, Name}) of
            [P] ->
                Salary = Amount + P#person.salary,
                P2 = P#person{salary = Salary},
                mnesia:write(P2);
            _ ->
                mnesia:abort("No such person")
        end
    end).
```

When this function executes within a transaction, several processes running on different nodes can concurrently execute the function `raise/2` without interfering with each other.

Since Mnesia detects deadlocks, a transaction can be restarted any number of times. This function attempts a restart as specified in `Retries`. `Retries` must be an integer greater than 0 or the atom `infinity`. Default is `infinity`.

`transform_table(Tab, Fun, NewAttributeList, NewRecordName) -> {aborted, R} | {atomic, ok}`

Applies argument `Fun` to all records in the table. `Fun` is a function that takes a record of the old type and returns a transformed record of the new type. Argument `Fun` can also be the atom `ignore`, which indicates that only the metadata about the table is updated. Use of `ignore` is not recommended, but included as a possibility for the user do to an own transformation.

`NewAttributeList` and `NewRecordName` specify the attributes and the new record type of the converted table. Table name always remains unchanged. If `record_name` is changed, only the Mnesia functions that use table identifiers work, for example, `mnesia:write/3` works, but not `mnesia:write/1`.

`transform_table(Tab, Fun, NewAttributeList) -> {aborted, R} | {atomic, ok}`

Calls `mnesia:transform_table(Tab, Fun, NewAttributeList, RecName)`, where `RecName` is `mnesia:table_info(Tab, record_name)`.

`traverse_backup(Source, [SourceMod,] Target, [TargetMod,] Fun, Acc) -> {ok, LastAcc} | {error, Reason}`

Iterates over a backup, either to transform it into a new backup, or read it. The arguments are explained briefly here. For details, see the User's Guide.

- `SourceMod` and `TargetMod` are the names of the modules that actually access the backup media.
- `Source` and `Target` are opaque data used exclusively by modules `SourceMod` and `TargetMod` to initialize the backup media.
- `Acc` is an initial accumulator value.
- `Fun(BackupItems, Acc)` is applied to each item in the backup. The `Fun` must return a tuple `{BackupItems,NewAcc}`, where `BackupItems` is a list of valid backup items, and `NewAcc` is a new accumulator value. The returned backup items are written in the target backup.
- `LastAcc` is the last accumulator value. This is the last `NewAcc` value that was returned by `Fun`.

`uninstall_fallback() -> ok | {error,Reason}`

Calls the function `mnesia:uninstall_fallback([{scope, global}])`.

`uninstall_fallback(Args) -> ok | {error,Reason}`

Deinstalls a fallback before it has been used to restore the database. This is normally a distributed operation that is either performed on all nodes with disc resident schema, or none. Uninstallation of fallbacks requires Erlang to be operational on all involved nodes, but it does not matter if Mnesia is running or not. Which nodes that are considered as disc-resident nodes is determined from the schema information in the local fallback.

`Args` is a list of the following tuples:

- `{module, BackupMod}`. For semantics, see `mnesia:install_fallback/2`.
- `{scope, Scope}`. For semantics, see `mnesia:install_fallback/2`.
- `{mnesia_dir, AlternateDir}`. For semantics, see `mnesia:install_fallback/2`.

`unsubscribe(EventCategory) -> {ok, Node} | {error, Reason}`

Stops sending events of type `EventCategory` to the caller.

`Node` is the local node.

`wait_for_tables(TabList, Timeout) -> ok | {timeout, BadTabList} | {error, Reason}`

Some applications need to wait for certain tables to be accessible to do useful work. `mnesia:wait_for_tables/2` either hangs until all tables in `TabList` are accessible, or until `timeout` is reached.

`wread({Tab, Key}) -> transaction abort | RecordList`

Calls the function `mnesia:read(Tab, Key, write)`.

`write(Record) -> transaction abort | ok`

Calls the function `mnesia:write(Tab, Record, write)`, where `Tab` is `element(1, Record)`.

`write(Tab, Record, LockKind) -> transaction abort | ok`

Writes record `Record` to table `Tab`.

The function returns `ok`, or terminates if an error occurs. For example, the transaction terminates if no `person` table exists.

The semantics of this function is context-sensitive. For details, see `mnesia:activity/4`. In transaction-context, it acquires a lock of type `LockKind`. The lock types `write` and `sticky_write` are supported.

`write_lock_table(Tab) -> ok | transaction abort`

Calls the function `mnesia:lock({table, Tab}, write)`.

Configuration Parameters

Mnesia reads the following application configuration parameters:

- `-mnesia access_module Module`. The name of the Mnesia activity access callback module. Default is `mnesia`.
- `-mnesia auto_repair true | false`. This flag controls if Mnesia automatically tries to repair files that have not been properly closed. Default is `true`.
- `-mnesia backup_module Module`. The name of the Mnesia backup callback module. Default is `mnesia_backup`.

- `-mnesia debug Level`. Controls the debug level of Mnesia. The possible values are as follows:

  `none`

  No trace outputs. This is the default.

  `verbose`

  Activates tracing of important debug events. These events generate `{mnesia_info, Format, Args}` system events. Processes can subscribe to these events with `mnesia:subscribe/1`. The events are always sent to the Mnesia event handler.

  `debug`

  Activates all events at the verbose level plus full trace of all debug events. These debug events generate `{mnesia_info, Format, Args}` system events. Processes can subscribe to these events with `mnesia:subscribe/1`. The events are always sent to the Mnesia event handler. On this debug level, the Mnesia event handler starts subscribing to updates in the schema table.

  `trace`

  Activates all events at the debug level. On this level, the Mnesia event handler starts subscribing to updates on all Mnesia tables. This level is intended only for debugging small toy systems, as many large events can be generated.

  `false`
  An alias for none.
  `true`
  An alias for debug.
- `-mnesia core_dir Directory`. The name of the directory where Mnesia core files is stored, or false. Setting it implies that also RAM-only nodes generate a core file if a crash occurs.
- `-mnesia dc_dump_limit Number`. Controls how often `disc_copies` tables are dumped from memory. Tables are dumped when `filesize(Log) > (filesize(Tab)/Dc_dump_limit)`. Lower values reduce CPU overhead but increase disk space and startup times. Default is 4.
- `-mnesia dir Directory`. The name of the directory where all Mnesia data is stored. The directory name must be unique for the current node. Two nodes must never share the the same Mnesia directory. The results are unpredictable.
- `-mnesia dump_disc_copies_at_startup true | false`. If set to false, this disables the dumping of `disc_copies` tables during startup while tables are being loaded. The default is true.
- `-mnesia dump_log_load_regulation true | false`. Controls if log dumps are to be performed as fast as possible, or if the dumper is to do its own load regulation. Default is `false`.

  This feature is temporary and will be removed in a future release
- `-mnesia dump_log_update_in_place true | false`. Controls if log dumps are performed on a copy of the original data file, or if the log dump is performed on the original data file. Default is `true`
- `-mnesia dump_log_write_threshold Max`. `Max` is an integer that specifies the maximum number of writes allowed to the transaction log before a new dump of the log is performed. Default is `100` log writes.
- `-mnesia dump_log_time_threshold Max`. `Max` is an integer that specifies the dump log interval in milliseconds. Default is 3 minutes. If a dump has not been performed within `dump_log_time_threshold` milliseconds, a new dump is performed regardless of the number of writes performed.
- `-mnesia event_module Module`. The name of the Mnesia event handler callback module. Default is `mnesia_event`.
- `-mnesia extra_db_nodes Nodes` specifies a list of nodes, in addition to the ones found in the schema, with which Mnesia is also to establish contact. Default is `[]` (empty list).

- `-mnesia fallback_error_function {UserModule, UserFunc}`. Specifies a user-supplied callback function, which is called if a fallback is installed and Mnesia goes down on another node. Mnesia calls the function with one argument, the name of the dying node, for example, `UserModule:UserFunc(DyingNode)`. Mnesia must be restarted, otherwise the database can be inconsistent. The default behavior is to terminate Mnesia.
- `-mnesia max_wait_for_decision Timeout`. Specifies how long Mnesia waits for other nodes to share their knowledge about the outcome of an unclear transaction. By default, `Timeout` is set to the atom `infinity`. This implies that if Mnesia upon startup detects a "heavyweight transaction" whose outcome is unclear, the local Mnesia waits until Mnesia is started on some (in the worst case all) of the other nodes that were involved in the interrupted transaction. This is a rare situation, but if it occurs, Mnesia does not guess if the transaction on the other nodes was committed or terminated. Mnesia waits until it knows the outcome and then acts accordingly.

  If `Timeout` is set to an integer value in milliseconds, Mnesia forces "heavyweight transactions" to be finished, even if the outcome of the transaction for the moment is unclear. After `Timeout` milliseconds, Mnesia commits or terminates the transaction and continues with the startup. This can lead to a situation where the transaction is committed on some nodes and terminated on other nodes. If the transaction is a schema transaction, the inconsistency can be fatal.
- `-mnesia no_table_loaders NUMBER`. Specifies the number of parallel table loaders during start. More loaders can be good if the network latency is high or if many tables contain few records. Default is `2`.
- `-mnesia send_compressed Level`. Specifies the level of compression to be used when copying a table from the local node to another one. Default is `0`.

  `Level` must be an integer in the interval `[0, 9]`, where `0` means no compression and `9` means maximum compression. Before setting it to a non-zero value, ensure that the remote nodes understand this configuration.
- `-mnesia schema_location Loc`. Controls where Mnesia looks for its schema. Parameter `Loc` can be one of the following atoms:

  `disc`

  Mandatory disc. The schema is assumed to be located in the Mnesia directory. If the schema cannot be found, Mnesia refuses to start. This is the old behavior.

  `ram`

  Mandatory RAM. The schema resides in RAM only. At startup, a tiny new schema is generated. This default schema only contains the definition of the schema table and only resides on the local node. Since no other nodes are found in the default schema, configuration parameter `extra_db_nodes` must be used to let the node share its table definitions with other nodes.

  Parameter `extra_db_nodes` can also be used on disc based nodes.

  `opt_disc`

  Optional disc. The schema can reside on disc or in RAM. If the schema is found on disc, Mnesia starts as a disc-based node and the storage type of the schema table is `disc_copies`. If no schema is found on disc, Mnesia starts as a disc-less node and the storage type of the schema table is `ram_copies`. Default value for the application parameter is `opt_disc`.

First, the SASL application parameters are checked, then the command-line flags are checked, and finally, the default value is chosen.

See Also

application(3), dets(3), disk_log(3), ets(3), mnesia_registry(3), qlc(3)

mnesia_frag_hash

Erlang module

This module defines a callback behavior for user-defined hash functions of fragmented tables.

Which module that is selected to implement the `mnesia_frag_hash` behavior for a particular fragmented table is specified together with the other `frag_properties`. The `hash_module` defines the module name. The `hash_state` defines the initial hash state.

This module implements dynamic hashing, which is a kind of hashing that grows nicely when new fragments are added. It is well suited for scalable hash tables.

Exports

`init_state(Tab, State) -> NewState | abort(Reason)`

Types:

`Tab = atom()`
`State = term()`
`NewState = term()`
`Reason = term()`

Starts when a fragmented table is created with the function `mnesia:create_table/2` or when a normal (unfragmented) table is converted to be a fragmented table with `mnesia:change_table_frag/2`.

Notice that the function `add_frag/2` is started one time for each of the other fragments (except number 1) as a part of the table creation procedure.

`State` is the initial value of the `hash_state` `frag_property`. `NewState` is stored as `hash_state` among the other `frag_properties`.

`add_frag(State) -> {NewState, IterFrags, AdditionalLockFrags} | abort(Reason)`

Types:

`State = term()`
`NewState = term()`
`IterFrags = [integer()]`
`AdditionalLockFrags = [integer()]`
`Reason = term()`

To scale well, it is a good idea to ensure that the records are evenly distributed over all fragments, including the new one.

`NewState` is stored as `hash_state` among the other `frag_properties`.

As a part of the `add_frag` procedure, Mnesia iterates over all fragments corresponding to the `IterFrags` numbers and starts `key_to_frag_number(NewState,RecordKey)` for each record. If the new fragment differs from the old fragment, the record is moved to the new fragment.

As the `add_frag` procedure is a part of a schema transaction, Mnesia acquires write locks on the affected tables. That is, both the fragments corresponding to `IterFrags` and those corresponding to `AdditionalLockFrags`.

`del_frag(State) -> {NewState, IterFrags, AdditionalLockFrags} | abort(Reason)`

Types:

State = term()
NewState = term()
IterFrags = [integer()]
AdditionalLockFrags = [integer()]
Reason = term()

`NewState` is stored as `hash_state` among the other `frag_properties`.

As a part of the `del_frag` procedure, Mnesia iterates over all fragments corresponding to the `IterFrags` numbers and starts `key_to_frag_number(NewState,RecordKey)` for each record. If the new fragment differs from the old fragment, the record is moved to the new fragment.

Notice that all records in the last fragment must be moved to another fragment, as the entire fragment is deleted.

As the `del_frag` procedure is a part of a schema transaction, Mnesia acquires write locks on the affected tables. That is, both the fragments corresponding to `IterFrags` and those corresponding to `AdditionalLockFrags`.

`key_to_frag_number(State, Key) -> FragNum | abort(Reason)`

Types:

FragNum = integer()
Reason = term()

Starts whenever Mnesia needs to determine which fragment a certain record belongs to. It is typically started at `read`, `write`, and `delete`.

`match_spec_to_frag_numbers(State, MatchSpec) -> FragNums | abort(Reason)`

Types:

MatcSpec = ets_select_match_spec()
FragNums = [FragNum]
FragNum = integer()
Reason = term()

This function is called whenever Mnesia needs to determine which fragments that need to be searched for a `MatchSpec`. It is typically called by `select` and `match_object`.

See Also

mnesia(3)

mnesia_registry

Erlang module

This module is usually part of the `erl_interface` application, but is currently part of the Mnesia application.

This module is mainly intended for internal use within OTP, but it has two functions that are exported for public use.

On C-nodes, `erl_interface` has support for registry tables. These tables reside in RAM on the C-node, but can also be dumped into Mnesia tables. By default, the dumping of registry tables through `erl_interface` causes a corresponding Mnesia table to be created with `mnesia_registry:create_table/1`, if necessary.

Tables that are created with these functions can be administered as all other Mnesia tables. They can be included in backups, replicas can be added, and so on. The tables are normal Mnesia tables owned by the user of the corresponding `erl_interface` registries.

Exports

`create_table(Tab) -> ok | exit(Reason)`

A wrapper function for `mnesia:create_table/2`, which creates a table (if there is no existing table) with an appropriate set of `attributes`. The table only resides on the local node and its storage type is the same as the `schema` table on the local node, that is, `{ram_copies,[node()]}` or `{disc_copies,[node()]}`.

This function is used by `erl_interface` to create the Mnesia table if it does not already exist.

`create_table(Tab, TabDef) -> ok | exit(Reason)`

A wrapper function for `mnesia:create_table/2`, which creates a table (if there is no existing table) with an appropriate set of `attributes`. The attributes and `TabDef` are forwarded to `mnesia:create_table/2`. For example, if the table is to reside as `disc_only_copies` on all nodes, a call looks as follows:

```
          TabDef = [{{disc_only_copies, node()|nodes()]}],
          mnesia_registry:create_table(my_reg, TabDef)
```

See Also

erl_interface(3), mnesia(3)