

# **Mnesia Database Management System (MNESIA)**

**version 3.7**

**Claes Wikström, Hans Nilsson and Håkan Mattsson**

**1997-05-27**

Typeset in  $\text{\LaTeX}$  from SGML source using the DOCBUILDER 3.0 Document System.



# Contents

<b>1</b>	<b>MNESIA User's Guide</b>	<b>1</b>
1.1	Introduction . . . . .	2
	The Mnesia DataBase Management System (DBMS) . . . . .	2
1.2	Getting Started with Mnesia . . . . .	5
	Starting Mnesia for the first time . . . . .	5
	An introductory example . . . . .	6
1.3	Building A Mnesia Database . . . . .	15
	Defining a Schema . . . . .	15
	The data model . . . . .	16
	Starting Mnesia . . . . .	16
	Creating New Tables . . . . .	19
1.4	Transactions and other access contexts . . . . .	22
	Transaction Properties . . . . .	22
	Locking . . . . .	24
	Dirty Operations . . . . .	27
	Record names versus table names . . . . .	28
	Activity concept and various access contexts . . . . .	30
	Nested transactions . . . . .	32
	Pattern matching . . . . .	32
1.5	Miscellaneous Mnesia Features . . . . .	34
	Indexing . . . . .	34
	Distribution and Fault Tolerance . . . . .	35
	Table Fragmentation . . . . .	36
	Local content tables . . . . .	41
	Disc-less nodes . . . . .	41
	More schema management . . . . .	42
	Mnesia event handling . . . . .	43
	Debugging Mnesia applications . . . . .	45
	Concurrent Processes in Mnesia . . . . .	45
	Prototyping . . . . .	46
	Object Based Programming with Mnesia . . . . .	47

1.6	Mnesia System Information . . . . .	50
	Database Configuration Data . . . . .	50
	Core Dumps . . . . .	50
	Dumping Tables . . . . .	50
	Checkpoints . . . . .	50
	Files . . . . .	52
	Loading of tables at start-up . . . . .	54
	Recovery from communication failure . . . . .	55
	Recovery of transactions . . . . .	56
	Backup, Fallback, and Disaster Recovery . . . . .	57
1.7	Combining Mnesia with SNMP . . . . .	62
	Combining Mnesia and SNMP . . . . .	62
1.8	Appendix A: Mnesia Error Messages . . . . .	63
	Errors in Mnesia . . . . .	63
1.9	Appendix B: The Backup Call Back Interface . . . . .	64
	mnesia_backup callback behavior . . . . .	64
1.10	Appendix C: The Activity Access Call Back Interface . . . . .	68
	mnesia_access callback behavior . . . . .	68
1.11	Mnesia Release Notes . . . . .	71
	Mnesia 3.7.1 . . . . .	71
	Mnesia 3.7 . . . . .	71
	Mnesia 3.6 . . . . .	72
	Mnesia 3.5.4 . . . . .	73
	Mnesia 3.5.3 . . . . .	73
	Mnesia 3.5.2 . . . . .	74
	Mnesia 3.5.1 . . . . .	74
	Mnesia 3.5 . . . . .	75
	Mnesia 3.4.1 . . . . .	75
	Mnesia 3.4 . . . . .	76
	Mnesia 3.3 . . . . .	81
	Mnesia 3.2 . . . . .	81
	Mnesia 3.1.1 . . . . .	82
	Mnesia 3.1 . . . . .	83
	Mnesia 3.0 . . . . .	83
	Mnesia 2.3 . . . . .	85
	Mnesia 2.2 . . . . .	86
	Mnesia 2.1.2 . . . . .	86
	Mnesia 2.1.1 . . . . .	87
	Mnesia 2.1 . . . . .	87
	Mnesia 2.0.2 . . . . .	89

Mnesia 2.0.1 . . . . .	90
Mnesia 2.0 . . . . .	91
Mnesia 1.3.2 . . . . .	94
Mnesia 1.3.1 . . . . .	94
Mnesia 1.3 . . . . .	95
Mnesia 1.2.3 . . . . .	95
Mnesia 1.2.2 . . . . .	96
Mnesia 1.2.1 . . . . .	96
Mnesia 1.2 . . . . .	97
Mnesia 1.1.1 . . . . .	97
Mnesia 1.1.1 . . . . .	98
Mnesia 1.1 . . . . .	98
Mnesia 1.0 . . . . .	99
Mnesia 0.X . . . . .	108
<b>2 Mnesia Reference Manual</b>	<b>109</b>
2.1 mnesia (Module) . . . . .	115
2.2 mnesia_registry (Module) . . . . .	143
 <b>List of Figures</b>	 <b>145</b>
 <b>List of Tables</b>	 <b>147</b>



# Chapter 1

## MNESIA User's Guide

*Mnesia* 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

This book describes the Mnesia DataBase Management System (DBMS). *Mnesia* is a distributed Database Management System, appropriate for telecommunications applications and other Erlang applications which require continuous operation and soft real-time properties. It is one section of the Open Telecom Platform (OTP), which is a control system platform for building telecommunications applications.

## The Mnesia DataBase Management System (DBMS)

### Features

Mnesia contains the following features which combine to produce a fault-tolerant, distributed database management system 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 a large number of functions can be called within one transaction.
- Several transactions can run concurrently, and their execution is fully synchronized by the database management system. 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 so called “dirty operations”, which reduce overheads and run very fast.

Details of these features are described in the following sections.

### Add-on Applications

Mnemosyne and Mnesia Session can be used in conjunction with Mnesia to produce specialized functions which enhance the operational ability of Mnesia. Both Mnesia Session and Mnemosyne have their own documentation as part of the OTP documentation set. Below are the main features of Mnesia Session and Mnemosyne when used in conjunction with Mnesia:

- *Mnemosyne*, a full-featured query language, can be combined with Mnesia to express complex relations between tables and define table views.
- *Mnemosyne* has the ability to optimize the query compiler for the Mnesia Database Management System, essentially making the DBMS more efficient.
- *Mnemosyne*, 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.

- *Mnesia Session* is an interface for the Mnesia Database Management System
- *Mnesia Session* enables access to the Mnesia DBMS from foreign programming languages (i.e. other languages than Erlang).

**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.

On the other hand, Mnesia may not be appropriate with the following types of applications:

- Programs that process plain text or binary data files
- Applications that merely need a look-up dictionary which can be stored to disc can utilize the standard library module `dets`, which is a disc based version of the module `ets`.
- Applications which need disc logging facilities can utilize the module `disc_log` by preference.
- Not suitable for hard real time systems.

## Scope and Purpose

This manual is included in the OTP document set. It describes how to build Mnesia database applications, and how to integrate and utilize the Mnesia database management system with OTP. Programming constructs are described, and numerous programming examples are included to illustrate the use of Mnesia.

## Prerequisites

Readers of this manual are assumed to be familiar with system development principles and database management systems. Readers are also assumed to be familiar with the Erlang programming language.

## About This Book

This book contains the following chapters:

- Chapter 1.2, “Getting Started with Mnesia”, introduces Mnesia with an example database. Examples are shown of 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.
- Chapter 1.3, “Building a Mnesia Database”, more formally describes the steps introduced in Chapter 2, namely the Mnesia functions which define a database schema, start Mnesia, and create the required tables.
- Chapter 1.4, “Transactions and other access contexts”, describes the transactions properties which make Mnesia into a fault tolerant, real-time distributed database management system. This chapter also describes the concept of locking in order to ensure consistency in tables, and so called “dirty operations”, or short cuts which bypass the transaction system to improve speed and reduce overheads.

- Chapter 1.5, “Miscellaneous Mnesia Features”, describes features which enable the construction of more complex database applications. These features includes indexing, checkpoints, distribution and fault tolerance, disc-less nodes, replication manipulation, local content tables, concurrency, and object based programming in Mnesia.
- Chapter 1.6, “Database Queries”, the specifically designed Mnesia query language, Mnemosyne, was previously part of Mnesia but is now a independent application. Mnemosyne has its own documentation which is part of the OTP documentation set.
- Chapter 1.7, “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.
- Chapter 1.8, “Combining Mnesia with SNMP”, is a short chapter which outlines Mnesia integrated with SNMP.
- Appendix A, “Mnesia Errors Messages”, lists Mnesia error messages and their meanings.
- Appendix B, “The Backup Call Back Interface”, is a program listing of the default implementation of this facility.
- Appendix C, “The Activity Access Call Back Interface”, is a program outlining of one possible implementations of this facility.

### Where to Find More Information

Refer to the following documentation for more information:

- the Reference Manual
- the Installation Guide
- the System Architecture Support Libraries (SASL) User Guide
- the SNMP User Guide
- the Erlang 4.4 Extensions User Guide
- the OS Specific Parts User Guide
- the Mnemosyne User Guide
- the Mnesia\_Session User Guide

## 1.2 Getting Started with Mnesia

This chapter introduces Mnesia. Following a brief discussion about the first initial setup, a Mnesia database example is demonstrated. This database example will be referenced in the following chapters, where this example is modified in order to illustrate various program constructs. In this chapter, the following mandatory procedures are illustrated by examples:

- Starting an Erlang session and specifying a directory for the Mnesia database.
- Initializing a database schema.
- Starting Mnesia and creating the required tables.

### Starting Mnesia for the first time

Following is a simplified demonstration of a Mnesia system startup. This is the dialogue from the Erlang shell:

```
unix> erl -mnesia dir "/tmp/funky"
Erlang (JAM) emulator version 4.5-alpha4

Eshell V4.5-alpha4 (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 the example above the following actions were performed:

- The Erlang system was started from the UNIX prompt with a flag `-mnesia dir '/tmp/funky'`. This flag indicates to Mnesia which directory will store the data.
- A new empty schema was initialized on the local node by evaluating `mnesia:create_schema([node()])`. The schema contains information about the database in general. This will be thoroughly explained later on.
- The DBMS was started by evaluating `mnesia:start()`.
- A first table was created, called `funky` by evaluating the expression `mnesia:create_table(funky, [])`. The table was given default properties.
- `mnesia:info()` was evaluated and subsequently displayed information regarding the status of the database on the terminal.

## An introductory example

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

In this example we shall:

- Start an Erlang system, and specify the directory where the database will be located.
- Initiate a new schema with an attribute that specifies on which node, or nodes, the database will operate.
- Start Mnesia itself.
- Create and populate the database tables.

## The Example Database

In this database example, we will create the database and relationships depicted in the following diagram. We will call this database the *Company* database.

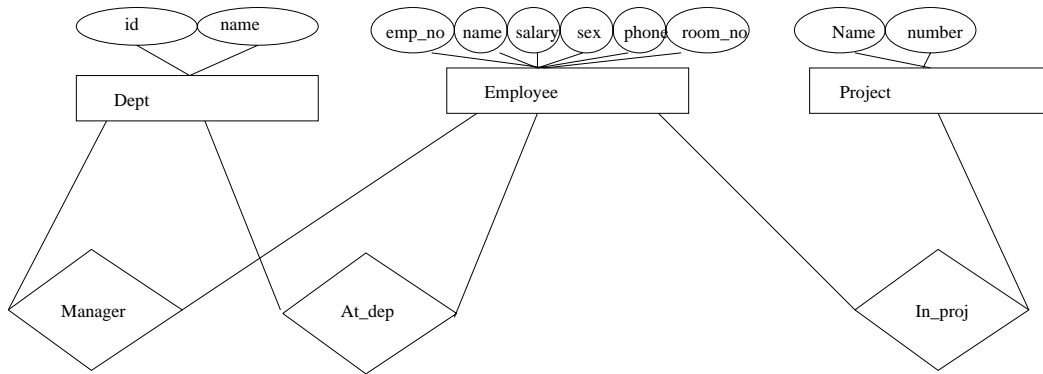


Figure 1.1: Company Entity-Relation Diagram

The database model looks as follows:

- There are three entities: employee, project, and department.
- There are three relationships between these entities:
  1. A department is managed by an employee, hence the *manager* relationship.
  2. An employee works at a department, hence the *at\_dep* relationship.
  3. Each employee works on a number of projects, hence the *in\_proj* relationship.

## Defining Structure and Content

We first enter our record definitions into a text file named `company.hr1`. This file defines the following structure for our sample 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 our database. In Mnesia, the function `mnesia:create_table(Name, ArgList)` is used to create 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 Chapter 4: Record Names Versus Table Names. [page 28]

For example, the table for employees will be 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.

## The Program

The following shell interaction starts Mnesia and initializes the schema for our company database:

```
% erl -mnesia dir '/ldisc/scratch/Mnesia.Company'  
Erlang (JAM) emulator version 4.5-alpha4  
  
Eshell V4.5-alpha4 (abort with ^G)  
1> mnesia:create_schema([node()]).  
ok  
2> mnesia:start().  
ok
```

The following program module creates and populates previously defined tables:

```
-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)}]).
```

## The Program Explained

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

- `% erl -mnesia dir '/ldisc/scratch/Mnesia.Company'`. This is a UNIX command line entry which 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, we have created a non-distributed system using only one node. Schemas are fully explained in Chapter 3: Defining a Schema [page 15].
- `mnesia:start()`. This function starts Mnesia. This function is fully explained in Chapter 3: Starting Mnesia [page 16].

Continuing the dialogue with the Erlang shell will produce the following 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 occupying 269 words of mem
at_dep      : with 0 records occupying 269 words of mem
manager     : with 0 records occupying 269 words of mem
project     : with 0 records occupying 269 words of mem
dept       : with 0 records occupying 269 words of mem
employee    : with 0 records occupying 269 words of mem
schema      : with 7 records occupying 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:

- `mnesia:create_table(Name,ArgList)`. This function is used to create the required database tables. The options available with `ArgList` are explained in Chapter 3: Creating New Tables [page 19].

The `company:init/0` function creates our tables. Two tables are of type bag. This is the manager relation as well the `in_proj` relation. This shall be 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 because an employee can only work in one department. In this data model we have examples of relations that are one-to-one (set), as well as one-to-many (bag).

`mnesia:info()` now indicates that a database which has seven local tables, of which, six are our 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 which 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.
```

- `insert_emp(Emp, DeptId, ProjNames) ->`. The `insert_emp/3` arguments are:

1. `Emp` is an employee record.
2. `DeptId` is the identity of the department where the employee is working.
3. `ProjNames` is a list of the names of the projects where the employee are working.

The `insert_emp(Emp, DeptId, ProjNames) ->` function creates a *functional object*. Functional objects are identified by the term `Fun`. The `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:

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

The function can be used as:

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

**Note:**

Functional Objects (Funs) are described in detail in the Erlang Development Environment User Guide, "Erlang 4.4 Extensions".

## Initial Database Content

After the insertion of the employee named `klacke` we have the following records in the database:

emp_no	name	salary	sex	phone	room_no
104732	klacke	7	male	99586	{221, 015}

Table 1.1: Employee

An employee record has the following Erlang record/tuple representation: `{employee, 104732, klacke, 7, male, 98108, {221, 015}}`

emp	dept_name
klacke	B/SFR

Table 1.2: At\_dep

At\_dep has the following Erlang tuple representation: `{at_dep, klacke, 'B/SFR'}`.

emp	proj_name
klacke	Erlang
klacke	otp
klacke	mnesia

Table 1.3: In\_proj

In\_proj has the following 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 will be used interchangeably throughout this book.

A Mnesia table is populated by Mnesia records. For example, the tuple `{boss, klacke, bjarne}` is an record. The second element in this tuple is the key. In order to uniquely identify a table row 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 `{boss, klacke, bjarne}` record 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.

We were also able to insert the `{boss, klacke, bjarne}` record which contains an implicit reference to another employee which does not yet exist in the database. Mnesia does not enforce this.

## Adding Records and Relationships to the Database

After adding additional record to the Company database, we may end up with 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}.
```

The above three tables, titled `employees`, `dept`, and `projects`, are the tables which are made up of real records. The following database content is stored in the tables which is built on relationships. These tables are titled `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 which consists of a tuple. The first element of the tuple identifies a corridor, and the second element identifies the actual room in the corridor. We could have chosen 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

There are two methods for writing database queries. These are:

- Mnesia functions
- List comprehensions.

This section contains simple introductory examples only. Refer to *Mnemosyne User's Guide* for a full description of the Mnemosyne query language.

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

```

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

```

A list comprehension must always run within a transaction. Consider the following function:

```

females() ->
  F = fun() ->
    Q = query [E.name || E <- table(employee),
              E.sex = female] end,
    mnemosyne:eval(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, the above operation would be called a selection, followed by a projection.

The list comprehension expression shown above contains a number of syntactical elements.

- the first [ bracket should be read as “build the list”
- the || “such that” and the arrow <- should be read as “taken from”

Hence, the above list comprehension demonstrates the formation of the list `E.name` such that `E` is taken from the table of employees and the `sex` attribute of each records unifies with the atom `female`.

Regarding unification, two things are important:

- The syntax “=” is used as opposed to “==” which indicates that the query is not a regular equality test. The syntax with the “=” is usually illegal in regular Erlang list comprehensions.
- Unification of expressions can be used to bind variables on both sides of the = sign.

The whole list comprehension must be enclosed within a query, `end` pair, and then given to the `mnesiosyne:eval/1` function.

It is possible to combine list comprehensions with low level Mnesia functions in the same transaction. If we want to raise the salary of all female employees we execute:

```
raise_females(Amount) ->
  F = fun() ->
    Q = query [E || E <- table(employee),
              E.sex = female] end,
    Fs = mnesiosyne:eval(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. Should an error occur, the value `{aborted, Reason}` is returned. In the case of an error, Mnesia guarantees that the salary is not raised for any employees at all.

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

## 1.3 Building A Mnesia Database

This chapter details the basic steps involved when designing a Mnesia database and the programming constructs which make different solutions available to the programmer. The chapter includes the following sections:

- defining a schema
- the datamodel
- starting Mnesia
- creating new tables.

### Defining a Schema

The configuration of a Mnesia system is described in the schema. The schema is a special table which contains information such as the table names and each table's storage type, (i.e. whether a table should be stored in RAM, on disc or possibly on both, as well as its location).

Unlike data tables, information contained 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. It is possible to move tables, delete tables, or reconfigure the layout of tables.

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.

The following section describes the functions available for schema management, all of which return a tuple:

- {atomic, ok}; or,
- {aborted, Reason} if unsuccessful.

### Schema Functions

- `mnesia:create_schema(NodeList)`. This function is used to initialize 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. The function will fail if a schema is already present on any of the nodes in `NodeList`. This function requires Mnesia to be stopped on the all `db_nodes` contained in the parameter `NodeList`. Applications call this function only once, since it is usually a one-time activity to initialize a new database.
- `mnesia:delete_schema(DiscNodeList)`. This function 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)`. This function permanently deletes all replicas of table `Tab`.

- `mnesia:move_table_copy(Tab, From, To)`. This function 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 node, it remains a RAM table on the new node. It is still possible for other transactions to perform read and write operation to the table while it is being moved.
- `mnesia:add_table_copy(Tab, Node, Type)`. This function creates a replica of the table `Tab` at node `Node`. The `Type` argument must be either of the atoms `ram_copies`, `disc_copies`, or `disc_only_copies`. If we add a copy of the system table `schema` to a node, this means that we want the Mnesia schema to reside there as well. This action then extends the set of nodes that comprise this particular Mnesia system.
- `mnesia:del_table_copy(Tab, Node)`. This function deletes the replica of table `Tab` at node `Node`. When the last replica of a table is removed, the table is deleted.
- `transform_table(Tab, Fun, NewAttributeList)`. This function changes the format on all records in table `Tab`. It applies the argument `Fun` to all records in the table. `Fun` shall be a function which takes an record of old type, and returns the record as new type. The table key may not be changed.
- `change_table_copy_type(Tab, Node, ToType)`. This function changes the storage type of a table. For example, a RAM table is changed to a `disc_table` at the node specified as `Node`.

## The 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 additional tables describing the actual relationships. Each table contains instances of Erlang records and records are represented as Erlang tuples.

Object identifiers, also known as `oid`, are made up of a table name and a key. For example, if we have an employee record represented by the tuple `{employee, 104732, klacke, 7, male, 98108, {221, 015}}`. This record has an object id, (`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`. See Chapter 4: Record Names Versus Table Names [page 28], for more information regarding the relationship between the record name and the table name.

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 could for example be a whole tree of oids leading to other terms in other tables. This type of record is hard to model in traditional relational DBMSs.

## Starting Mnesia

Before we can start Mnesia, we must initialize an empty schema on all the participating nodes.

- The Erlang system must be started.
- Nodes with disc database schema must be defined and implemented with the function `create_schema(NodeList)`.

When running a distributed system, with two or more participating nodes, then the `mnesia:start()` function must be executed on each participating node. Typically this would 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.

## Initializing a Schema and Starting Mnesia

To use a known example, we illustrate how to run the Company database described in Chapter 2 on two separate nodes, which we call `a@gin` and `b@skeppet`. Each of these nodes must have a Mnesia directory as well as 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 our Company database:

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

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

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

1. On the node called gin:

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

2. On the node called skeppet:

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

3. On one of the two nodes:

```
(a@gin1)>mnesia:create_schema([a@gin, b@skeppet]).
```

4. The function `mnesia:start()` is called on both nodes.

5. 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 above, the two directories reside on different nodes, because the `/ldisc/scratch` (the “local” disc) exists on the two different nodes.

By executing these commands we have configured two Erlang nodes 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 which manipulate the data in our tables can be executed on either of the two nodes. Code which 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. Both the `start/0` and the `stop/0` functions work on the “local” Mnesia system, and there are no functions which start or stop a set of nodes.

## The Start-Up Procedure

Mnesia is started by calling the following function:

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

This function initiates the DBMS locally.

The choice of configuration will alter the location and load order of the tables. The alternatives are listed below:

1. Tables that are stored locally only, are initialized from the local Mnesia directory.
2. 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 is the most recent. Mnesia determines which of the tables is the most recent.
3. 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 may take some time, and the application programmer must wait for the tables that the application needs before they can be used. This achieved by using the function:

- `mnesia:wait_for_tables(TabList, Timeout)`

This function 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, the initialization procedure will not proceed. In this situation, a call to `mnesia:wait_for_tables/2` suspends the caller until the remote node has initiated the table from its local disc and the node has copied the table over the network to the local node.

This procedure can be time consuming however, the shortcut function shown below will load all the tables from disc at a faster rate:

- `mnesia:force_load_table(Tab)`. This function forces tables to be loaded from disc regardless of the network situation.

Thus, we can assume that if an application wishes to use tables a and b, then the application must perform some action similar to the below code before it can utilize 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 start-up procedure fails, the `mnesia:start()` function returns the cryptic tuple `{error, {shutdown, {mnesia_sup, start, [normal, []]}}}`. Use command line arguments `-boot start_sasl` as argument to the `erl` script in order to get more information about the start failure.

## Creating New Tables

Mnesia provides one function to create new tables. This function is: `mnesia:create_table(Name, ArgList)`.

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:

- `Name` is the atomic name of the table. It is usually the same name as the name of the records that constitute the table. (See `record_name` for more details.)
- `ArgList` is a list of `{Key, Value}` tuples. The following arguments are valid:
  - `{type, Type}` where `Type` must be either of the atoms `set` or `bag`. The default value is `set`. A table of type `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 will return the list `[{foo,1,3}]` if the `foo` table is of type `set`. However, list `[{foo,1,2}, {foo,1,3}]` will return if the table is of type `bag`. Note the use of `bag` and `set` table types.

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 will reside on disc. Write operations to a table replica of type `disc_copies` will write data to the disc copy as well as 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. The default value is `[]`. This arrangement is desirable if we want the following operational characteristics are required:

1. read operations must be very fast and performed in RAM
2. all write operations must be written to persistent storage.

A write operation on a `disc_copies` table replica will be 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. The default value for `NodeList` is `[node()]`. If the default value is used to create a new table, it will be 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 shall build and maintain. An index table will exist 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 Guide. Basically, if this attribute is present in `ArgList` of `mnesia:create_table/2`, the table is immediately accessible by means of the Simple Network Management Protocol (SNMP).  
It is easy to design applications which 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 which make up a Mnesia table. `[]` is default.
- `{local_content, true}` When an application needs a table whose contents should be locally unique on each node, `local_content` tables may 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. The default value 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 macro pre-processor and returns a list of the record's 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]`. Accordingly, it is possible to provide the attribute names yourself, or to use the `record_info/2` notation.

It is recommended that the `record_info/2` notation be used as it is easier to maintain the program and it will be 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. The `record_name` defaults to the name of the table. For more information see Chapter 4: Record Names Versus Table Names [page 28].

As an example, assume we have the record definition:

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

The below call would create a table which is replicated on two nodes, has an additional index on the `y` attribute, 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 below default code values:

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

would return a table with a RAM copy on the local node, no additional indexes and the attributes defaulted to the list `[key, val]`.

## 1.4 Transactions and other access contexts

This chapter describes the Mnesia transaction system and the transaction properties which make Mnesia a fault tolerant, distributed database management system.

Also covered in this chapter are the locking functions, including table locks and sticky locks, as well as alternative functions which bypass the transaction system in favor of improved speed and reduced overheads. These functions are called “dirty operations”. We also describe the usage of nested transactions. This chapter contains the following sections:

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

### Transaction Properties

Transactions are an important tool 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 which is run as a transaction is called a Functional Object (Fun), and this code can read, write, or delete Mnesia records. The Fun is evaluated as a transaction which either commits, or aborts. If a transaction succeeds in executing Fun it will replicate the action on all nodes involved, or abort if an error occurs.

The following example shows a transaction which raises the salary of certain employee numbers.

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

The transaction `raise(Eno, Raise) - >` contains a Fun made up of four lines of code. 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 of the nodes.
- The Mnesia transactions have four important properties, which we call *Atomicity*, *Consistency*, *Isolation*, and *Durability*, or ACID for short. These properties are described in the following sub-sections.

## Atomicity

*Atomicity* means that database changes which are executed by a transaction take effect on all nodes involved, or on none of the nodes. In other words, the transaction either succeeds entirely, or it fails entirely.

Atomicity is particularly important when we want to atomically write more than one record in the same transaction. The `raise/2` function, shown as an example above, writes one record only. The `insert_emp/3` function, shown in the program listing in Chapter 2, writes the record `employee` as well as employee relations such as `at_dep` and `in_proj` into the database. If we run this latter code inside a transaction, then 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 such applications, it is important that a series of write operations are performed atomically inside a transaction. The atomicity property ensures that a transaction take effect on all nodes, or none at all.

## Consistency

*Consistency*. This transaction property ensures that a transaction always leaves the DBMS in a consistent state. For example, Mnesia ensures that inconsistencies will not occur if Erlang, Mnesia or the computer crashes while a write operation is in progress.

## Isolation

*Isolation*. This transaction property ensures that transactions which execute on different nodes in a network, and access and manipulate the same data records, will not interfere with each other.

The isolation property makes it possible to concurrently execute the `raise/2` function. A classical problem in concurrency control theory is the so called “lost update problem”.

The isolation property is extremely useful if the following circumstances occurs where an employee (with an employee number 123) and two processes, (P1 and P2), are concurrently trying to raise the salary for the employee. The initial value of the employees salary is, for example, 5. Process P1 then starts to execute, it reads the employee record and adds 2 to the salary. At this point in time, process P1 is for some reason preempted and process P2 has the opportunity to run. P2 reads the record, adds 3 to the salary, and finally writes a new employee record with the salary set to 8. Now, process P1 start 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 concurrently execute two or more processes which 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 may be written as if they had sole access to the data.

## Durability

*Durability.* This transaction property ensures that changes made to the DBMS by a transaction are permanent. Once a transaction has been committed, all changes made to the database are durable - i.e. they are written safely to disc and will not be corrupted or disappear.

### Note:

The durability feature described does not entirely apply to situations where Mnesia is configured as a “pure” primary memory database.

## Locking

Different transaction managers employ different strategies to satisfy the isolation property. Mnesia uses the standard technique of two-phase locking. This means that locks are set on records before they are read or written. Mnesia uses five different kinds of locks.

- *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 an 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 which satisfy some particular property, it is most inefficient to set read locks on the records, one by one. It is also very memory consuming, since the read locks themselves may take up considerable space if the table is very large. For this reason, Mnesia can set a read lock on an entire table.
- *Write table locks.* If a transaction writes a large number of records to one table, it is possible to set a write lock on the entire table.
- *Sticky locks.* These are write locks that stay in place at a node after the transaction which 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 have 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 will be evaluated one more time.

For this reason, it is 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 could write the text "Trying to write ... " a thousand times to the terminal. Mnesia does guarantee, however, that each and every 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 which 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, therefore, it is very 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 could bring the whole system to a standstill since other transactions which execute in other processes, or on other nodes, are forced to wait for the defective transaction.

If a transaction terminates abnormally, Mnesia will automatically release the locks held by the transaction.

We have shown examples of a number of functions that can be used inside a transaction. The following list shows the *simplest* Mnesia functions that work with transactions. It is important to realize that these functions must be embedded in a transaction. If no enclosing transaction (or other enclosing Mnesia activity) exists, they will all fail.

- `mnesia:transaction(Fun) -> {aborted, Reason} | {atomic, Value}`. This function executes one transaction with the functional object `Fun` as the single parameter.
- `mnesia:read({Tab, Key}) -> transaction abort | RecordList`. This function 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`, the `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`. This function behaves the same way as the previously listed `read/1` function, except that it acquires a write lock instead of a read lock. If we execute a transaction which reads a record, modifies the record, and then writes the record, it is slightly more efficient to set the write lock immediately. In cases where we issue a `mnesia:read/1`, 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`. This function writes a record into the database. The `Record` argument is an instance of a record. The function returns `ok`, or aborts the transaction if an error should occur.
- `mnesia:delete({Tab, Key}) -> transaction abort | ok`. This function deletes all records with the given key.
- `mnesia:delete_object(Record) -> transaction abort | ok`. This function deletes records with object id `Record`. This function is used when we want 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 we read a record, and lock all replicas of a record when we write a record. However, there are applications which use Mnesia mainly for its fault-tolerant qualities, and these applications may be configured with one node doing all the heavy work, and a standby node which is ready to take over in case the main node fails. Such applications may benefit from using sticky locks instead of the normal locking scheme.

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

```
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:

- one network rpc (2 messages) to acquire the write lock
- three network messages to execute the two-phase commit protocol.

If we use sticky locks, we must first change the code as follows:

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

This code uses the `s_write/1` function instead of the `write/1` function. The `s_write/1` function 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 we set a sticky lock on node `N1`, this lock will then stick to node `N1`. The next time we try to set a sticky lock on the same record at node `N1`, Mnesia will see that the lock is already set and will not do a network operation in order to acquire the lock.

It is much more efficient to set a local lock than it is to set a networked lock, and for this reason sticky locks can benefit application that use a replicated table and perform most of the work on only one of the nodes.

If a record is stuck at node `N1` and we try to set a sticky lock for the record on node `N2`, the record must be unstuck. This operation is expensive and will reduce performance. The unsticking is done automatically if we 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 have to code these operations. However, transactions which read and write a large number of records in a specific table will execute more efficiently if we start the transaction by setting a table lock on this table. This will block other concurrent transactions from the table. The following two function are used to set explicit table locks for read and write operations:

- `mnesia:read_lock_table(Tab)` Sets a read lock on the table `Tab`
- `mnesia:write_lock_table(Tab)` Sets a write lock on the table `Tab`

Alternate syntax for acquisition of table locks is as follows:

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

The matching operations in Mnesia may either lock the entire table or just 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 the `LockItem` on all `Nodes` in the `nodes` list.

## Dirty Operations

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

Dirty operation are useful in many situations, 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. For this reason, Mnesia has functions which manipulate tables without using transactions. This alternative to processing is known as a dirty operation. However, it is important to realize 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 we simultaneously use dirty operations 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 will ensure a certain level of consistency. For example, it is not possible for dirty operations to 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. The following functions are available:

- `mnesia:dirty_read({Tab, Key})`. This function reads record(s) from Mnesia.
- `mnesia:dirty_write(Record)`. This function writes the record `Record`
- `mnesia:dirty_delete({Tab, Key})`. This function deletes record(s) with the key `Key`.
- `mnesia:dirty_delete_object(Record)` This function is the dirty operation alternative to the function `delete_object/1`

- `mnesia:dirty_first(Tab)`. This function returns the “first” key in the table `Tab`. Records in Mnesia tables are not sorted. However, there is a record order which is not known to the user. This means that it is possible to traverse a table by means of this function in conjunction with the `dirty_next/2` function.  
If there are no records at all in the table, this function will return the atom `'$end_of_table'`. It is not recommended to use this atom as the key for any user records.
- `mnesia:dirty_next(Tab, Key)`. This function returns the “next” key in the table `Tab`. This function makes it possible to traverse a table and perform some operation on all records in the table. When the end of the table is reached the special key `'$end_of_table'` is returned. Otherwise, the function returns a key which can be used to read the actual record.  
The behavior is undefined if we perform any write operations on the table while we traverse the table with the `dirty_next/2` function. This is because write operations on a Mnesia table may lead to internal reorganizations of the table itself. This is an implementation detail, but remember the dirty functions are low level functions. `mnesia:read_lock_table(Tab)` may be used to ensure that no transaction protected writes are performed during the iteration.
- `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 `dirty_next/2` function. 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. `mnesia:read_lock_table(Tab)` may 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 will add the `Val` and the counter where `Val` is a positive or negative integer.  
There exists no special counter records in Mnesia. However, records on the form of `{TabName, Key, Integer}` can be used as counters, and can be persistent.  
It is not possible to have transaction protected updates of counter records.  
There are two significant differences when using this function instead of reading the record, performing the arithmetic, and writing the record:
  1. it is much more efficient
  2. the `dirty_update_counter/2` function is performed as an atomic operation although it is not protected by a transaction. Accordingly, no table update is lost if two processes simultaneously execute the `dirty_update_counter/2` function.
- `mnesia:dirty_match_object(Pat)`. This function is the dirty equivalent of `mnesia:match_object/1`.
- `mnesia:dirty_index_match_object(Pat, Pos)`. This function is the dirty equivalent of `mnesia:index_match_object/2`.
- `mnesia:dirty_index_read(Tab, SecondaryKey, Pos)`. This function is the dirty equivalent of `mnesia:index_read/3`.
- `mnesia:dirty_all_keys(Tab)`. This function is the dirty equivalent of `mnesia:all_keys/1`.

## 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 does however not necessarily be the same as the table name. Even

though that it is the case in the most of the examples in this document. If a table is created without the `record_name` property the code below will ensure 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 below, it is possible to store subscriber records 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).
```

In order to access such tables it is not possible to use the simplified access functions as described earlier in the document. For example, writing a subscriber record into a table requires a `mnesia:write/3` function 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 simplified piece of code illustrates the relationship between the simplified access functions used in most 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).
```

## Activity concept and various access contexts

As previously described, a functional object (Fun) performing table access operations as listed below may be passed on as arguments to the function `mnesia:transaction/1,2,3`:

- `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/2` (`match_object/1`)
- `mnesia:all_keys/1`
- `mnesia:index_match_object/4` (`index_match_object/2`)
- `mnesia:index_read/3`
- `mnesia:lock/2` (`read_lock_table/1`, `write_lock_table/1`)

- `mnesia:table_info/2`

These functions will be performed in a transaction context involving mechanisms like locking, logging, replication, checkpoints, subscriptions, commit protocols etc. However, the same function may also be evaluated in other activity contexts.

The following activity access contexts are currently supported:

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

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

By passing the same “fun” as an argument to the function `mnesia:sync_dirty(Fun [, Args])` it will be performed in almost the same context as `mnesia:async_dirty/1,2`. The difference is that the operations are performed synchronously. The caller will wait for the updates to be performed on all active replicas. Using `sync_dirty` is useful for applications that are executing on several nodes and want 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. This is also useful in situations where an application performs frequent or voluminous updates which may overload Mnesia on other nodes.

Mnesia tables with storage type `RAM_copies` and `disc_copies` are implemented internally as “ets-tables” and it is possible for applications to access 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 will be performed but in a very raw context. The operations will be performed directly on the local ets tables assuming that the local storage type are `RAM_copies` and that the table is not replicated on other nodes. Subscriptions will not be triggered nor checkpoints updated, but this operation is blindingly fast. Disc resident tables should not be updated with the `ets`-function since the disc will not be updated.

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

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

In Appendix C “The Activity Access Call Back Interface” the source code for one alternate implementation is provided (`mnesia_frag.erl`). The context sensitive function `mnesia:table_info/2` may be used to provide virtual information about a table. One usage of this is to perform `Mnemosyne` queries within an activity context with a customized callback module. By providing table information about table indices and other `Mnemosyne` requirements, `Mnemosyne` may be used as a generic query language to access virtual tables.

`Mnemosyne` queries may be performed in all these activity contexts (`transaction`, `async_dirty`, `sync_dirty` and `ets`). The `ets` activity will only work if the table has no indices.

**Note:**

The `mnesia:dirty_*` function always executes with `async_dirty` semantics regardless of which activity access contexts are invoked. They may even invoke contexts without any enclosing activity access context.

## Nested transactions

Transactions may be nested in an arbitrary fashion. A child transaction must run in the same process as its parent. When a child transaction aborts, the caller of the child transaction will get the return value `{aborted, Reason}` and any work performed by the child will be erased. If a child transaction commits, the records written by the child will be 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 updates performed by a nested transaction are only propagated in such a manner so that the parent of the nested transaction sees the updates. No final commitment will be done until the top level transaction is terminated. So, although a nested transaction returns `{atomic, Val}`, if the enclosing parent transaction is aborted, the entire nested operation is aborted.

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

Say for example that we have a function that adds a new subscriber to a telephony system:

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

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

## Pattern matching

Mnesia provides the programmer with several functions for matching records against a pattern. The most useful function of these is:

```
mnesia:match_object(Tab, Pattern, LockKind) ->
    transaction abort | RecordList
```

This function matches a `Pattern` against all records in table `Tab`. It is not necessarily performed as an exhaustive search of the entire table. By utilizing indices and bound values in the pattern, the actual work done by the function can be condensed into a few hash lookups.

The pattern provided to the `mnesia:match_object/3` function 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 (otherwise known as an Erlang term).

Use the function `mnesia:table_info(Tab, wild_pattern)` to obtain a basic pattern which matches all records in a table. Do not make the pattern hardcore since it will make your code more vulnerable to future changes of the record definition.

For the employee table the wild pattern will look like:

```
{employee, '__', '__', '__', '__', '__', '__'}
```

In order to constrain the match you must replace some of the `'_'` elements. The code for matching out all female employees, looks like:

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

It is also possible to use the match function if we want to check the equality of different attributes. Assume that we want to find all employees which happens to have a employee number which equals to their room number:

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

If the key attribute is bound in a pattern, the match operation is very 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. If the table is large, this can become a time consuming operation. This can be remedied with indices (refer to Chapter 5: Indexing [page 34]).

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

## 1.5 Miscellaneous Mnesia Features

The earlier chapters of this User Guide described how to get started with Mnesia, and how to build a Mnesia database. In this chapter, we will describe the more advanced features available when building a distributed, fault tolerant Mnesia database. This chapter contains the following sections:

- Indexing
- Distribution and Fault Tolerance
- Table fragmentation.
- Local content tables.
- Disc-less nodes.
- More about schema management
- Debugging a Mnesia application
- Concurrent Processes in Mnesia
- Prototyping
- Object Based Programming with Mnesia.

### Indexing

Data retrieval and matching can be performed very efficiently if we know the key for the record. Conversely, if the key is not known, all records in a table must be searched. The larger the table the more time consuming it will become. To remedy this problem Mnesia's 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 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 which adds an index on the element `salary` can be expressed in the following way:

1. `mnesia:add_table_index(employee, salary)`

The indexing capabilities of Mnesia are utilized with the following three functions, which retrieve and match records on the basis of 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 the `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 the `AttributeName` field of the `Pattern`. The secondary key must be bound.

- `mnesia:match_object(Pattern) -> transaction abort` | `RecordList` Uses indices to avoid exhaustive search of the entire table. Unlike the other functions above, this function may utilize any index as long as the secondary key is bound.

These functions are further described and exemplified in Chapter 4: Pattern matching [page 32] and in *Mnemosyne User's Guide, Database Queries*.

## Distribution and Fault Tolerance

Mnesia is a distributed, fault tolerant DBMS. It is possible to replicate tables on different Erlang nodes in a variety of ways. The Mnesia programmer does not have to state where the different tables reside, only the names of the different tables are specified in the program code. This is known as “location transparency” and it is an important concept. In particular:

- A program will work regardless of the location of the data. It makes no difference whether the data resides on the local node, or on a remote node. *Note:* The program will run 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 effect the user programs.

We have previously seen 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 will create a new 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 will reside on each node in the list. This is a RAM replica, and it is important to realize that no disc operations are performed when a program executes write operations to these replicas. However, should permanent RAM replicas be a requirement, then the following alternatives are available:
  1. The `mnesia:dump_tables/1` function can be used to dump RAM table replicas to disc.
  2. The table replicas can be backed up; either from RAM, or from disc if dumped there with the above function.
- `disc_copies`. The value of the attribute is a list of Erlang nodes, and a replica of the table will reside both in RAM and on disc on each node in the list. Write operations addressed to the table will 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 will reside 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.

It is also possible to set and change table properties on existing tables. Refer to Chapter 3: Defining the Schema [page 15] for full details.

There are basically two reasons for using more than one table replica: fault tolerance, or speed. It is worthwhile to note that table replication provides a solution to both of these system requirements.

If we have two active table replicas, all information is still available if one of the replicas fail. This can be a very important property in many applications. Furthermore, if a table replica exists at two specific nodes, applications which 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 which reads data often, but writes data seldom, in order 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.

## Table Fragmentation

### The Concept

A concept of table fragmentation has been introduced in order to cope with very large tables. The idea is to split a table into several more manageable fragments. Each fragment is implemented as a first class Mnesia table and may be replicated, have indices etc. as any other table. But the tables may neither have `local_content` nor have the `snmp` connection activated.

In order 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 the `mnesia_frag` module, which implements the `mnesia_access` callback behaviour. Please, read the documentation about `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. Secondly the name of the table fragment is determined from the hash value. And 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. The following piece of code illustrates how an existing Mnesia table is converted to be a fragmented table and how more fragments are added later on.

```
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

There is a table property called `frag_properties` and may be read with `mnesia:table_info(Tab, frag_properties)`. The fragmentation properties is a list of tagged tuples with the 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:

- `{n_fragments, Int}` `n_fragments` regulates how many fragments that the table currently has. This property may 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 may 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 will end up with the same number of replicas. `node_pool` defaults to the return value from `mnesia:system_info(db_nodes)`.
- `{n_ram_copies, Int}` Regulates how many `ram_copies` replicas that each fragment should have. This property may explicitly be set at table creation. The default is 0, but if `n_disc_copies` and `n_disc_only_copies` also are 0, `n_ram_copies` will default be set to 1.
- `{n_disc_copies, Int}` Regulates how many `disc_copies` replicas that each fragment should have. This property may explicitly be set at table creation. The default is 0.

{n\_disc\_only\_copies, Int} Regulates how many `disc_only_copies` replicas that each fragment should have. This property may explicitly be set at table creation. The default is 0.

{foreign\_key, ForeignKey} `ForeignKey` may either be the atom `undefined` or the tuple {`ForeignTab`, `Attr`}, where `Attr` denotes an attribute which should be interpreted as a key in another fragmented table named `ForeignTab`. Mnesia will ensure that the number of fragments in this table and in the foreign table are always the same. When fragments are added or deleted Mnesia will automatically propagate the operation to all fragmented tables that has a foreign key referring to this table. Instead of using the record key to determine which fragment to access, the value of the `Attr` field is used. This feature makes it possible to automatically co-locate records in different tables to the same node. `foreign_key` defaults to `undefined`. However if the foreign key is set to something else it will cause the default values of the other fragmentation properties to be the same values as the actual fragmentation properties of the foreign table.

```
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},
                                         {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. The `Change` argument should have one of the following values:

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

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

`{add_frag, NodesOrDist}` Adds one new fragment to a fragmented table. All records in one of the old fragments will be rehashed and about half of them will be moved to the new (last) fragment. All other fragmented tables, which refers to this table in their foreign key, will automatically get a new fragment, and their records will also be dynamically rehashed in the same manner as for the main table.

The `NodesOrDist` argument may either be a list of nodes or the result from `mnesia:table_info(Tab, frag_dist)`. The `NodesOrDist` argument is assumed to be a sorted list with the best nodes to host new replicas first in the list. The new fragment will get 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 one fragment from a fragmented table. All records in the last fragment will be moved to one of the other fragments. All other fragmented tables which refers to this table in their foreign key, will automatically loose their last fragment and their records will also be dynamically rehashed in the same manner as for the main table.

`{add_node, Node}` Adds a new node to the `node_pool`. The new node pool will affect the list returned from `mnesia:table_info(Tab, frag_dist)`.

`{del_node, Node}` Deletes a new node from the `node_pool`. The new node pool will affect the list returned from `mnesia:table_info(Tab, frag_dist)`.

## Extensions of Existing Functions

The function `mnesia:create_table/2` is used to create a brand new fragmented table, by setting the table property `frag_properties` to some proper values.

The function `mnesia:delete_table/2` is used to delete a fragmented table including all its fragments. There must however not exist any other fragmented tables which refers to this table in their foreign key.

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

If the function `mnesia:table_info/2` is invoked in the activity context of the `mnesia_frag` module, information of several new items may 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 prototype and when the actual values needs to be computed (e.g. when adding new fragments) they are simply determined by counting the number of each replicas for each storage type. This means, 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 will affect the settings on `n_ram_copies`, `n_disc_copies`, and `n_disc_only_copies`.

`foreign_key` the foreign key.

`foreigners` all other tables that refers to this table in their foreign key.

`frag_names` the names of all fragments.

`frag_dist` a sorted list of `{Node, Count}` tuples which is sorted in increasing `Count` order. The `Count` is the total number of replicas that this fragmented table hosts on each `Node`. The list always contains at least all nodes in the `node_pool`. The nodes which not belongs to the `node_pool` will be 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` the 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 simply depends of the application needs. Here follows some examples of situations which may need some attention:

`permanent change of nodes` when a new permanent `db_node` is introduced or dropped, it may be time to change the pool of nodes and re-distribute the replicas evenly over the new pool of nodes. It may also be time to add or delete a fragment before the replicas are re-distributed.

`size/memory threshold` when the total size or total memory of a fragmented table (or a single fragment) exceeds some application specific threshold, it may be time to dynamically add a new fragment in order obtain a better distribution of records.

`temporary node down` when a node temporarily goes down it may be time to compensate some fragments with new replicas in order to keep the desired level of redundancy. When the node comes up again it may be time to remove the superfluous replica.

`overload_threshold` when the load on some node is exceeds some application specific threshold, it may be time to either add or move some fragment replicas to nodes with lesser load. Extra care should be taken if the table has a foreign key relation to some other table. In order to avoid severe performance penalties, the same re-distribution must be performed for all of the related tables.

Use `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 re-distribution.

## 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 we specify the attribute `{local_content, true}` when we create the table, the table will reside on the nodes where we specify that the table shall exist, but the write operations on the table will only be performed on the local copy.

Furthermore, when the table is initialized at start-up, the table will only be initialized locally, and the table content will not be copied from another node.

## Disc-less nodes

It is possible to run Mnesia on nodes that do not have a disc. It is of course not possible to have replicas of neither `disc_copies`, nor `disc_only_copies` on such nodes. This especially troublesome for the schema table since Mnesia need the schema in order to initialize itself.

The schema table may, as other tables, reside on one or more nodes. The storage type of the schema table may either be `disc_copies` or `ram_copies` (not `disc_only_copies`). At start-up Mnesia uses its schema to determine with which nodes it should try to establish contact. If any of the other nodes are already started, 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. The application parameter `extra_db_nodes` contains a list of nodes which Mnesia also should establish contact with besides the ones found in the schema. The default value is the empty list `[]`.

Hence, when a disc-less node needs to find the schema definitions from a remote node on the network, we need to supply this information through the application parameter `-mnesia extra_db_nodes NodeList`. Without this configuration parameter set, Mnesia will start as a single node system.

The application parameter `schema_location` controls where Mnesia will search for its schema. The parameter may be one of the following atoms:

- `disc` Mandatory disc. The schema is assumed to be located on the Mnesia directory. And if the schema cannot be found, Mnesia refuses to start.
- `ram` Mandatory ram. The schema resides in ram only. At start-up a tiny new schema is generated. This default schema contains just the definition of the schema table and only resides on the local node. Since no other nodes are found in the default schema, the configuration parameter `extra_db_nodes` must be used in order to let the node share its table definitions with other nodes. (The `extra_db_nodes` parameter may also be used on disc-full nodes.)

`opt_disc` Optional disc. The schema may 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 value for the application parameter is `opt_disc`.

When the `schema_location` is set to `opt_disc` the function `mnesia:change_table_copy_type/3` may be used to change the storage type of the schema. This is illustrated below:

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

Assuming that the call to `mnesia:start` did not find any schema to read on the disc, then Mnesia has started as a disc-less node, and then changed it to a node that utilizes the disc to locally store the schema.

## More schema management

It is possible to add and remove nodes 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` may be used to add and delete replicas of the schema table. Adding a node to the list of nodes where the schema is replicated will affect two things. First it allows other tables to be replicated to this node. Secondly it will cause Mnesia to try to contact the node at start-up of disc-full nodes.

If the storage type of the schema is `ram_copies`, i.e. we have disc-less node, Mnesia will not use the disc on that particular node. The disc usage is enabled by changing the storage type of the table `schema` to `disc_copies`.

New schemas are created explicitly with `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 `mnesia:create_table/2` as normal tables.

At start-up 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 will shutdown one of the nodes. This unfortunate situation will occur if a table has been created on two nodes independently of each other while they were disconnected. To solve the problem, one of the tables must be deleted (as the cookies differ we regard it to be two different tables even if they happen to 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.

`mnesia:system_info(schema_location)` and `mnesia:system_info(extra_db_nodes)` may be used to determine the actual values of `schema_location` and `extra_db_nodes` respectively.

`mnesia:system_info(use_dir)` may be used to determine whether Mnesia is actually using the Mnesia directory. `use_dir` may be determined even before Mnesia is started. The function `mnesia:info/0` may now be used to printout some system information even before Mnesia is started. When Mnesia is started the function prints out more information.

Transactions which 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 may be created and new replicas may be added without starting all of the `disc_full` nodes. New replicas may be added before all other replicas of the table have been loaded, it will suffice when one other replica is active.

## Mnesia event handling

System events and table events are the two categories of events that Mnesia will generate in various situations.

### System events

The system events are detailed below:

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

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

`{mnesia_checkpoint_activated, Checkpoint}` a checkpoint with the name `Checkpoint` has been activated and that the current node is involved in the checkpoint. Checkpoints may be activated explicitly with `mnesia:activate_checkpoint/1` or implicitly at backup, adding table replicas, internal transfer of data between nodes etc. By default this event is ignored.

`{mnesia_checkpoint_deactivated, Checkpoint}` A checkpoint with the name `Checkpoint` has been deactivated and that the current node was involved in the checkpoint. Checkpoints may explicitly be deactivated with `mnesia:deactivate/1` or implicitly when the last replica of a table (involved in the checkpoint) becomes unavailable, e.g. at node down. By default this event is ignored.

`{mnesia_overload, Details}` Mnesia on the current node is overloaded and the subscriber should take action.

A typical overload situation occurs when the applications are performing more updates on disc resident tables than Mnesia is able to handle. Ignoring this kind of overload may lead into 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 of 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 over and over again. If the thresholds for dumping the transaction log have been reached before the previous dump was 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 are performing updates of disc resident tables. When this happens the message queue of the transaction manager will continue to grow until the memory is exhausted or the load decreases.

The same problem may occur for dirty updates. The overload is detected locally on the current node, but its cause may be on another node. Application processes may cause heavy loads if any table are residing on other nodes (replicated or not). By default this event is reported to the `error_logger`.

`{inconsistent_database, Context, Node}` Mnesia regards the database as potential inconsistent and gives its applications a chance to recover from the inconsistency, e.g. by installing a consistent backup as fallback and then restart the system or pick a `MasterNode` from

`mnesia:system_info(db_nodes)` and invoke `mnesia:set_master_node([MasterNode])`. By default an error is reported to the error logger.

`{mnesia_fatal, Format, Args, BinaryCore}` Mnesia has encountered a fatal error and will (in a short period of time) be terminated. The reason for the fatal error is explained in `Format` and `Args` which may be given as input to `io:format/2` or sent to the `error_logger`. By default it will be sent to the `error_logger`. `BinaryCore` is a binary containing a summary of Mnesia's internal state at the time the when the fatal error was encountered. By default the binary is written to a unique file name on current directory. On RAM nodes the core is ignored.

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

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

`{mnesia_user, Event}` An application has invoked the function `mnesia:report_event(Event)`. `Event` may be any Erlang data structure. When tracing a system of Mnesia applications it is useful to be able to interleave Mnesia's own events 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 may be a good idea to use `mnesia:report_event/1`.

## Table events

Another category of events are table events, which are events related to table updates. The table events are tuples typically looking like this: `{Oper, Record, ActivityId}`. Where `Oper` is the operation performed. `Record` is the record involved in the operation and `ActivityId` is the identity of the transaction performing the operation. Note that the name of the record is the table name even when the `record_name` has another setting. The various table related events that may occur are:

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

`{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. Note that, other records with the same key may be remaining in the table if it is a bag.

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

It is possible for user process to subscribe on the events generated by Mnesia. We have the following two functions:

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

`mnesia:unsubscribe(EventCategory)` Removes the subscription on events of type `EventCategory`

`EventCategory` may either be the atom `system` or the tuple `{table, Tab}`. The subscribe functions activate a subscription of events. The events are delivered as messages to the process evaluating the `mnesia:subscribe/1` function. The syntax of system events is `{mnesia_system_event, Event}` and `{mnesia_table_event, Event}` for table events. What system events and table events means is described above.

All system events are subscribed by Mnesia's `gen_event` handler. The default `gen_event` handler is `mnesia_event`. But it may be changed by using the 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)` may be used to determine which processes are subscribed to various events.

## Debugging Mnesia applications

Debugging a Mnesia application can be difficult due to a number of reasons, primarily related to difficulties in understanding how the transaction and table load mechanisms work. An other source of confusion may be the semantics of nested transactions.

We may set the debug level of Mnesia by calling:

- `mnesia:set_debug_level(Level)`

Where the parameter `Level` is:

`none` no trace outputs at all. This is the default.

`verbose` activates tracing of important debug events. These debug events will generate `{mnesia_info, Format, Args}` system events. Processes may subscribe to these events with `mnesia:subscribe/1`. The events are always sent to Mnesia's event handler.

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

`trace` activates all events at the debug level. On this debug level Mnesia's event handler starts subscribing updates on all Mnesia tables. This level is only intended for debugging small toy systems, since many large events may be generated.

`false` is an alias for `none`.

`true` is an alias for `debug`.

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

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

## 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 may include statements that read, write or delete data from the DBMS. A large number of such transactions can run concurrently, and the programmer does not have to explicitly synchronize the processes which manipulate the data. All programs accessing the database through the transaction system may be written as if they had sole access to the data. This is a very desirable property since 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.

It is possible to move tables, delete tables or reconfigure the layout of a table in various ways. An important aspect of the actual implementation of these functions is that it is possible for user programs to continue to use a table while it is being reconfigured. For example, it is possible to simultaneously move a table and perform write operations to the table. This is important for many applications that require continuously available services. Refer to Chapter 4: Transactions and other access contexts [page 22] for more information.

## Prototyping

If and when we decide that we 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 write all data into one file, process that file and have the data in the file inserted into the database. It is possible to initialize Mnesia with data read from a text file. We have the following two functions to work with text files.

- `mnesia:load_textfile(Filename)` Which 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 only operates on the local node.
- `mnesia:dump_to_textfile(Filename)` Dumps all local tables of a mnesia system into a text file which can then be edited (by means of a normal text editor) and then later reloaded.

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

The format of the text file is:

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

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

Options is a list of {Key,Value} tuples conforming to the options we could give to `mnesia:create_table/2`.

For example, if we want to start playing with a small database for healthy foods, we enter then following data into the 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 then shows how to load the fruits database.

```
% erl
Erlang (JAM) emulator version 4.5-alpha4

Eshell V4.5-alpha4 (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 occupying 299 words of mem
fruit          : with 2 records occupying 291 words of mem
schema        : with 3 records occupying 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>
```

Where we can see that the DBMS was initiated from a regular textfile.

## Object Based Programming with Mnesia

The Company database introduced in Chapter 2 has three tables which store records (employee, dept, project), and three tables which 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, we can easily remove one project, as the following example illustrates:

```
remove_proj(ProjName) ->
  F = fun() ->
    Ip = mnesiosyne:eval(query [X || X <- table(in_proj),
                                      X.proj_name = ProjName]
                        end),
    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 which is not even in first normal form. Mnesia is very suitable for applications such as telecommunications, because it is easy to organize data in a very flexible manner. A Mnesia database is always organized as a set of tables. Each table is filled with rows/objects/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 our Company database had been a part of a telecommunications system, then it could be that the lookup time of an employee *together* with a list of the projects the employee is working on, should be minimized. If this was the case, we might choose a drastically different data model which has no direct relationships. We would only have the records themselves, and different records could contain either direct references to other records, or they could contain other records which are not part of the Mnesia schema.

We could create the following record definitions:

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

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

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

An record which describes an employee might look like this:

```

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 only has three different tables, and the employee records contain references to other records. We have the following references in the record.

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

We could also use the Mnesia record identifiers (`{Tab, Key}`) as references. In this case, the dept attribute would be set to the 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 our Company database. On the other hand, 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) ->
  F = fun() -> eval(query
    [E || E <- table(employee),
      E.dept = Dept,
      E.salary > Salary]
    end)
  end,
  transaction(F).

```

This code is not only easier to write and to understand, but it also executes much faster.

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

## 1.6 Mnesia System Information

### Database Configuration Data

The following two functions can be used to retrieve system information. They are described in detail in the reference manual.

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

### Core Dumps

If Mnesia malfunctions, system information is dumped to a file named `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 be included in the bug report.

### Dumping Tables

Tables of type `ram_copies` are by definition stored in memory only. It is possible, however, to dump these tables to disc, either at regular intervals, or before the system is shutdown. 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 `.DAT` file on the disc. When the Mnesia system is started, the RAM table will initially be loaded with data from its `.DAT` file.

### Checkpoints

A checkpoint is a transaction consistent state that spans over one or more tables. When a checkpoint is activated, the system will remember 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 may be involved in several checkpoints simultaneously.

Each table retains its old contents in a checkpoint retainer and for performance critical applications, it may be important to realize the processing overhead associated with checkpoints. In a worst case scenario, the checkpoint retainer will consume even more memory than the table itself. Each update will also be slightly slower on those nodes where checkpoint retainers are attached to the tables.

For each table it is possible to choose if there should 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 will consume less memory, but it will be vulnerable to node crashes. With several redundant checkpoint retainers the checkpoint will survive as long as there is at least one active checkpoint retainer attached to each table.

Checkpoints may 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}` in the case of an error. All tables in a checkpoint must be attached to at least one checkpoint retainer. The checkpoint is automatically de-activated by Mnesia, when any table lacks a checkpoint retainer. This may happen when a node goes down or when a replica is deleted. Use the `min` and `max` arguments described below, 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}`. `Name` specifies a temporary name of the checkpoint. The name may be re-used when the checkpoint has been de-activated. If no name is specified, a name is generated automatically.
- `{max, MaxTabs}`. `MaxTabs` is a list of tables which will be included in the checkpoint. The default is `[]` (an empty list). For these tables, the redundancy will be maximized. The old contents of the table will be retained in the checkpoint retainer when the main table is updated by the applications. The checkpoint becomes more fault tolerant if the tables have several replicas. When new replicas are added by means of the schema manipulation function `mnesia:add_table_copy/3`, it will also attach a local checkpoint retainer.
- `{min, MinTabs}`. `MinTabs` is a list of tables that should be included in the checkpoint. The default is `[]`. For these tables, the redundancy will be minimized, and there will be a single checkpoint retainer per table, preferably at the local node.
- `{allow_remote, Bool}`. `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. The 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 should 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 are the records that will be loaded on start-up. Default is `false`.

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

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

`Name` is the name of the checkpoint, and `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)`. This function returns all active checkpoints on the current node.
- `mnesia:table_info(Tab, checkpoints)`. This function returns active checkpoints on a specific table.

## Files

This section describes the internal files which are created and maintained by the Mnesia system, in particular, the workings of the Mnesia log is described.

### Start-Up Files

In Chapter 3 we detailed the following pre-requisites for starting Mnesia (refer Chapter 3: Starting Mnesia [page 16]):

- We must start an Erlang session and specify a Mnesia directory for our database.
- We must initiate a database schema, using the function `mnesia:create_schema/1`.

The following example shows how these tasks are performed:

1.

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

2.

```
Erlang (JAM) emulator version 4.5-alpha4
```

```
Eshell V4.5-alpha4 (abort with ^G)
```

```
(klacke@gin)1> mnesia:create_schema([node()]).
```

```
ok
```

```
(klacke@gin)2>
```

```
^Z
```

```
Suspended
```

We can inspect the Mnesia directory to see what files have been created. Enter the following command:

```
% 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 we had specified more than one node in the `mnesia:create_schema/1` function, identical backup files would have been created on all nodes.

3. Continue by starting Mnesia:

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

```
ok
```

We can now see the following listing in the Mnesia directory:

```
-rw-rw-r--  1 klacke  staff      237 May 26 19:03 DECISION.ETS
-rw-rw-r--  1 klacke  staff        89 May 26 19:03 DECISION.LOG
-rw-rw-r--  1 klacke  staff        86 May 26 19:03 LATEST.LOG
drwxrwxr-x  2 klacke  staff      512 May 26 19:02 OLD_DIR
-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 we have no other disc resident tables than the schema, no other data files were created. The file FALLBACK.BUP was removed after the successful “restoration”. We also see a number of files that are for internal use by Mnesia.

4. Enter the following command to create a table:

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

We can now see the following listing in the Mnesia directory:

```
% ls -l /ldisc/scratch/klacke
-rw-rw-r-- 1 klacke staff 237 May 26 19:07 DECISION.ETS
-rw-rw-r-- 1 klacke staff 89 May 26 19:07 DECISION.LOG
-rw-rw-r-- 1 klacke staff 86 May 26 19:07 LATEST.LOG
drwxrwxr-x 2 klacke staff 512 May 26 19:02 OLD_DIR
-rw-rw-r-- 1 klacke staff 34055 May 26 19:07 foo.DAT
-rw-rw-r-- 1 klacke staff 34997 May 26 19:07 schema.DAT
```

Where a file `foo.DAT` has been created. This file stores all data that is written into the `foo` table.

## The Log File

When starting Mnesia, a `.LOG` file called `LATEST.LOG` was 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 which is of storage type `disc_copies`, or `disc_only_copies`. It also includes all operations which manipulate the schema itself, such as creating new tables. The format of the log can vary with different implementations of Mnesia. The Mnesia log is currently implemented with the standard library module `disc_log`.

The log file will grow continuously and must be dumped at regular intervals. “Dumping the log file” means that Mnesia will perform all the operations listed in the log and place the records in the corresponding `.DAT` data files. For example, if the operation “write record {foo, 4, elvis, 6}” is listed in the log, Mnesia inserts the record into the file `foo.DAT`. The dumping operation can be time consuming if the log is very large. However, it is important to realize that the Mnesia system continues to operate during log dumps.

By default Mnesia either dumps the log whenever 100 records have been written in the log or when 3 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 start-up and whenever a schema operation is performed.

## The Data Files

The directory listing also contains two `.DAT` files. These are the data files where the database records are stored, and the schema itself is also 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 Mnesia data files are currently implemented with the standard library module `dets`, and all operations which can be performed on `dets` files can also be performed on the Mnesia data files. For example, `dets` contains a 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 a our schema file, we can:

```
{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).
```

**Note:**

Refer to the Reference Manual, `std_lib` for information about `dets`.

**Warning:**

The DAT files must always be opened with the `{repair, false}` option. This ensures that these files are not automatically repaired. Without this option, the database may become inconsistent, because Mnesia may believe that the files were properly closed. Refer to the reference manual for information about the configuration parameter `auto_repair`.

**Warning:**

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

## Loading of tables at start-up

At start-up Mnesia loads tables in order to make them accessible for its applications. Sometimes Mnesia decides to load all tables that reside locally, and sometimes the tables may not be accessible until Mnesia brings a copy of the table from another node.

To understand the behavior of Mnesia at start-up 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 happens, Mnesia will assume that the other node is no longer running. Whereas, in reality, the communication between the nodes has merely failed.

To overcome this situation, simply 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 start-up, it must be noted that all tables residing on nodes without a `mnesia_down` entry, may have fresher replicas. Their replicas may have been updated after the termination of Mnesia on the current node. In order 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 may 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)` ought to be executed to ensure that the table is accessible from the local node. If the function times out the application may choose to force a load of the local replica with `mnesia:force_load_table(Tab)` and deliberately lose all updates that may have been performed on the other nodes while the local

node was down. If Mnesia already has loaded the table on another node or intends to do so, we will copy the table from that node in order to avoid unnecessary inconsistency.

### Warning:

Keep in mind that it is only one table that is loaded by `mnesia:force_load_table(Tab)` and since committed transactions may have caused updates in several tables, the tables may now become inconsistent due to the forced load.

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

At start up, Mnesia will assume that its local replica is the most recent version and load the table from disc if either situation is detected:

- `mnesia_down` is returned from all other nodes that holds a disc resident replica of the table; or,
- if all replicas are `ram_copies`

This is normally a wise decision, but it may turn out to be disastrous if the nodes have been disconnected due to a communication failure, since Mnesia's normal table load mechanism does not cope with communication failures.

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

## Recovery from communication failure

There are several occasions when Mnesia may detect that the network has been partitioned due to a communication failure.

One is when Mnesia already is up and running and the Erlang nodes gain contact again. Then Mnesia will try 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 Mnesia's event handler and other possible subscribers. The default event handler reports an error to the error logger.

Another occasion when Mnesia may detect that the network has been partitioned due to a communication failure, is at start-up. If Mnesia detects that both the local node and another node received `mnesia_down` from each other it generates a `{inconsistent_database, starting_partitioned_network, Node}` system event and acts as described above.

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

At start-up Mnesia's normal table load algorithm will be bypassed and the table will be loaded from one of the master nodes defined for the table, regardless of potential `mnesia_down` entries in the log. The Nodes may only contain nodes where the table has a replica and if it is empty, the master node recovery mechanism for the particular table will be reset and the normal load mechanism will be used when next restarting.

The function `mnesia:set_master_nodes(Nodes)` sets master nodes for all tables. For each table it will determine its replica nodes and invoke `mnesia:set_master_nodes(Tab, TabNodes)` with those replica nodes that are included in the `Nodes` list (i.e. `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 will be reset and the normal load mechanism will be used at next restart.

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

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

## Recovery of transactions

A Mnesia table may reside on one or more nodes. When a table is updated, Mnesia will ensure that the updates will be replicated to all nodes where the table resides. If a replica happens to be inaccessible for some reason (e.g. due to a temporary node down), Mnesia will then perform the replication later.

On the node where the application is started, there will be a transaction coordinator process. If the transaction is distributed, there will also be 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, (i.e. 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 needs to exchange is few, since Mnesia's table load mechanism takes care of the transaction recovery if the commit protocol gets interrupted. Since all involved tables are replicated symmetrically the transaction will automatically be recovered by loading the involved tables from the same node at start-up of a failing node. We do not really care if the transaction was aborted or committed as long as we can ensure the ACID properties. The lightweight commit protocol is non-blocking, i.e. the surviving participants and their coordinator will finish the transaction, regardless of some node crashes in the middle of the commit protocol or not.

If a node goes down in the middle of a dirty operation the table load mechanism will ensure that the update will be 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 will be used. The heavyweight commit protocol is able to finish the transaction regardless of how the tables are replicated. The typical usage of a heavyweight transaction is when we want to move a replica from one node to another. Then we must ensure that the replica either is entirely moved or left as it was. We must never end up in a situation with replicas on both nodes or 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 will perform recovery work at start-up in order to finish the abort 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 start-up, Mnesia will determine the outcome of the transaction and recover it. Lightweight protocols, heavyweight protocols and dirty updates, are dependent on other nodes to be up and running in order 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 or any of the already running nodes know the outcome of the transaction, Mnesia will by default wait for one. In the worst case scenario all other involved nodes must start before Mnesia can make the correct decision about the transaction and finish its start-up.

This means that Mnesia (on one node) may hang if a double fault occurs, i.e. when two nodes crash simultaneously and one attempts to start when the other refuses to start e.g. due to a hardware error.

It is possible to specify the maximum time that Mnesia will wait for other nodes to respond with a transaction recovery decision. The configuration parameter `max_wait_for_decision` defaults to infinity (which may cause the indefinite hanging as mentioned above) but if it is set to a definite time period (eg. three minutes), Mnesia will then enforce a transaction recovery decision if needed, in order to allow Mnesia to continue with its start-up procedure.

The downside of an enforced transaction recovery decision, is that the decision may be incorrect, due to insufficient information regarding the other nodes' recovery decisions. This may result in an inconsistent database where Mnesia has committed the transaction on some nodes but aborted it on others.

In fortunate cases the inconsistency will only appear in tables belonging to a specific application, but if a schema transaction has been inconsistently recovered due to the enforced transaction recovery decision, the effects of the inconsistency can be fatal. However, if the higher priority is availability rather than consistency, then it may be worth the risk.

If Mnesia encounters a inconsistent transaction decision a `{inconsistent_database, bad_decision, Node}` system event will be generated in order 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 (see above).

## Backup, Fallback, and Disaster Recovery

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

- `mnesia:backup_checkpoint(Name, Opaque, [Mod])`. This function performs a backup of the tables included in the checkpoint.
- `mnesia:backup(Opaque, [Mod])`. This function activates a new checkpoint which covers all Mnesia tables and performs a backup. It is performed with maximum degree of redundancy (also refer to the function `mnesia:activate_checkpoint(Args)` [page 50], `{max, MaxTabs}` and `{min, MinTabs}`).
- `mnesia:traverse_backup(Source, [SourceMod,] Target, [TargetMod,] Fun, Ac)`. This function can be used to read an existing backup, create a new backup from an existing one, or to copy a backup from one type media to another.
- `mnesia:uninstall_fallback()`. This function removes previously installed fallback files.
- `mnesia:restore(Opaque, Args)`. This function restores a set of tables from a previous backup.
- `mnesia:install_fallback(Opaque, [Mod])`. This function can be configured to restart the Mnesia and reload data tables, and possibly 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 sub-sections. Also refer to the section Checkpoints [page 50] in this chapter, which describes the two functions used to activate and de-activate 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 via the `mnesia_backup` module for both read and write. Currently `mnesia_backup` is implemented with the standard library module `disc_log`, but it is possible to write your own module with 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. Use the configuration parameter `-mnesia_backup_module <module>` for this purpose.

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

- `Name` is the name of an activated checkpoint. Refer to the section Checkpoints [page 50] in this chapter, the function `mnesia:activate_checkpoint(ArgList)` for details on how to include table names 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 file name.
- `Mod`. The name of an alternate backup module.

The function `mnesia:backup(Opaque[, Mod])` activates a new checkpoint which 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 the `local_contents` property are backed up as they look on the current node.

It is possible to iterate over a backup, either for the purpose of transforming it into a new backup, or just reading 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:

- `SourceMod` and `TargetMod` are module names.
- `Source` and `Target` are opaque data used exclusively by the modules `SourceMod` and `TargetMod` for the purpose of 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 `{ValidBackupItems, NewAcc}`, where `ValidBackupItems` is a list of valid backup items, and `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. I.e. the last `NewAcc` value that was returned by `Fun`.

It is also possible to perform a read-only traversal of the source backup without updating a target backup. If `TargetMod==read_only`, then no target backup is accessed at all.

By setting `SourceMod` and `TargetMod` to different modules it is possible to copy a backup from one kind of 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. See `mnesia_create_table/2` for more information about `CreateList`.
- `{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. Note 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. It is not possible to mix schema records with other records and all schema records must be located first in the backup.

The schema itself is a table and will possibly be included in the backup. All nodes where the schema table resides are regarded as a `db_node`.

The following example illustrates 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 on-line 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 will be used.
- `{skip_tables, TableList}` Where `TableList` is a list of tables which should not be read from the backup.
- `{clear_tables, TableList}` Where `TableList` is a list of tables which should be cleared, before the records from the backup are inserted, i.e. all records in the tables are deleted before the tables are restored. Schema information about the tables is not cleared or read from backup.
- `{keep_tables, TableList}` Where `TableList` is a list of tables which should be not be cleared, before the records from the backup are inserted, i.e. the records in the backup will be added to the records in the table. Schema information about the tables is not cleared or read from backup.
- `{recreate_tables, TableList}` Where `TableList` is a list of tables which should be re-created, 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 needs to be up and running.
- `{default_op, Operation}` Where `Operation` is one of the following operations `skip_tables`, `clear_tables`, `keep_tables` or `recreate_tables`. The default operation specifies which operation should be used on tables from the backup which are not specified in any of the lists above. If omitted, the operation `clear_tables` will be used.

The argument `Opaque` is forwarded to the backup module. It returns `{atomic, TabList}` if successful, or the tuple `{aborted, Reason}` in the case of an error. `TabList` is a list of the restored tables. Tables which are restored are write locked for the duration of 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 very large, it may not be possible to restore it online. In such a case the old database must be restored by installing a fallback, and then restart.

## Fallbacks

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

Installing a fallback is a distributed operation that is *only* performed on all `db_nodes`. The fallback is used to restore 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 for some reason, it will unconditionally terminate 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 re-installation of the old applications will be 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` in order to restore the old database. The fallback will be automatically de-installed after a successful start-up. The function `mnesia:uninstall_fallback()` may also be used to de-install 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 de-installation of fallbacks require Erlang to be up and running on all `db_nodes`, but it does not matter if Mnesia is running or not.

## Disaster Recovery

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

If Mnesia detects that a file has not been properly closed, possibly as a result of a power failure, it will attempt to repair the bad file in a similar manner. Data may be lost, but Mnesia can be restarted even if the data is inconsistent. The configuration parameter `-mnesia auto_repair <bool>` can be used to control the behavior of Mnesia at start-up. If `<bool>` has the value `true`, Mnesia will attempt to repair the file; if `<bool>` has the value `false`, Mnesia will 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.

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

Replicas of type `disc_only_copies` will only be affected by this parameter during the initial dump of the log file at start-up. When designing applications which have *very* high requirements, it may 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 reason for a reason such as a power failure, these files may 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 has been corrupted, it can be reconstructed from a backup. This should be regarded as a last resort, since the backup contains old data. The data is hopefully consistent, but data will definitely be lost when an old backup is used to restore the database.

## 1.7 Combining Mnesia with SNMP

### Combining 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:

- Not being able to control the application remotely at all.
- Using a proprietary control protocol.
- Using a bridge which maps control messages in a proprietary protocol to a standardized management protocol and vice versa.

All of these approaches have different advantages and disadvantages. Mnesia applications can easily be opened to the SNMP protocol. It is possible to establish a direct one-to-one mapping 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 Mnesia reference manual.

## 1.8 Appendix A: Mnesia Error Messages

Whenever an operation returns an error in Mnesia, a description of the error is available. For example, the functions `mnesia:transaction(Fun)`, or `mnesia:create_table(N,L)` may return the tuple `{aborted, Reason}`, where Reason is a term describing the error. The following function is used to retrieve more detailed information about the error:

- `mnesia:error_description(Error)`

### Errors in Mnesia

The following is a list of valid errors in Mnesia.

- `badarg`. Bad or invalid argument, possibly bad type.
- `no_transaction`. Operation not allowed outside transactions.
- `combine_error`. Table options were 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 (non-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. Record(s) are no longer available elsewhere in the network.
- `not_a_db_node`. A node was mentioned which 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.

The following example illustrates a function which returns an error, and the method to retrieve more detailed error information.

The function `mnesia:create_table(bar, [{attributes, 3.14}])` will return 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.

## 1.9 Appendix B: The Backup Call Back Interface

### 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).
-copyright('Copyright (c) 1991-98 Ericsson Telecom AB').
-author('hakan@erix.ericsson.se').
-behaviour(mnesia_backup).

-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.

```

```
        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,
    Repair = mnesia:system_info(auto_repair),
    case file:file_info(File) of
        {error, Reason} ->
            {error, Reason};
        Info -> %% file exists
            case disk_log:open([file, File],
                               {name, make_ref()},
                               {repair, Repair},
                               {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} ->
      {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 C: The Activity Access Call Back Interface

### mnesia\_access callback behavior

```
-module(mnesia_frag).
-copyright('Copyright (c) 1991-98 Ericsson Telecom AB').
-author('hakan@erix.ericsson.se').
-behaviour(mnesia_access).

%% 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, Pat, LockKind) ->
  Match = [mnesia:match_object(ActivityId, Opaque, Frag, Pat, LockKind)
           || Frag <- frag_names(Tab)],
  lists:flatten(Match).
```

```

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

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:flatten(Match).

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

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_hash.n_fragments;
  foreign_key ->
    FH = lookup_frag_hash(Tab),
    FH#frag_hash.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}));

  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.
```

## 1.11 Mnesia Release Notes

This document describes the changes made to the Mnesia system from version to version. The intention of this document is to list all incompatibilities as well as all enhancements and bugfixes for every release of Mnesia. Each release of Mnesia thus constitutes one section in this document. The title of each section is the version number of Mnesia.

### Mnesia 3.7.1

#### Improvements and new features

None.

#### Fixed Bugs and malfunctions

- Fixed a problem with table loading, mnesia terminated if the receiving node disappeared in a certain state.
- `add_table_copy/3` didn't work local tables.

#### Incompatibilities

None.

#### Known bugs and problems

None.

### Mnesia 3.7

#### Improvements and new features

- `mnesia:dirty_all_keys/1` has been optimized.  
Own Id: OTP-2914.  
Aux Id: seq 1389.
- Various optimizations has been performed, such as schema transactions.  
Own Id: OTP-2804
- A concept of table fragmentation has been introduced in order to cope with very large tables. The idea is to split a table into several more manageable fragments. Each fragment is implemented as a first class Mnesia table and may be replicated, have indecies etc. as any other table.  
The functions `mnesia:create_table/2`, `mnesia:delete_table/1` and `mnesia:table_info/2` has been extended to cope with fragmented tables.  
The fragmentation properties of a table can be changed with the new function `mnesia:change_table_frag/2`.  
See the Users Guide for more information.

Own Id: OTP-1748, OTP-2789.

### Fixed Bugs and malfunctions

- Mnesia's internal process architecture has partly been redesigned in order to avoid some rare deadlock situations during startup.  
Own Id: OTP-2495, OTP-2767, OTP-2421.  
Aux Id: seq 4, seq 1138.
- The algorithm for copying of tables between nodes has been improved, in order to better cope with dirty updates during the copy operation.

### Incompatibilities

Mnesia must be restarted on all nodes in order to cope with the following changes:

- The internal process architecture has been changed.
- The message protocol between nodes has been changed.
- The schema representation has been changed.

### Known bugs and problems

The new concept of fragmented tables must be described in the `Users Guide` and `Reference Manual`.

## Mnesia 3.6

### Improvements and new features

- The internal `OLD_DIR` directory is no more, since it was almost like a persistent memory leak.
- The fallback concept has been improved. Installation and uninstallation of fallbacks are now more predictable.

The installation is performed only on the nodes with disc resident schema. The operation will fail if the local node is not one of the disc resident nodes. Which nodes that are disc resident or not, is determined from the schema info in the backup.

The uninstallation is performed only on the nodes with disc resident schema. Which nodes that are disc resident or not, is determined from the schema info in the local fallback. The operation will fail if there is no local fallback installed.

It is now possible to install and uninstall `local` fallbacks. By default, the installation (and uninstallation) scope of a fallback is `global`. But if `{scope, local}` is given as argument, the operation will only affect the local node. The effect of invoking the `install` (or `uninstall`) operation manually on each node with disc resident schema is the same as one single invocation with `global` scope.

Normally local installation and uninstallation is targeted towards the local Mnesia directory (see the `-mnesia_dir` configuration parameter). But if `{mnesia_dir, AlternateDir}` is given as argument, the operation will be performed on the alternate directory regardless of the Mnesia directory configuration parameter setting. After installation of a fallback on an alternate Mnesia directory that directory is fully prepared for usage as an active Mnesia directory. This is a somewhat dangerous feature which must be used with care. By unintentional mixing of directories you may easily end up with a inconsistent database, if the same backup is installed on more than one directory.

Please, read more about `mnesia:install_fallback/1,2` and `mnesia:uninstall_fallback/0,1` in the reference manual.

### Fixed Bugs and malfunctions

Yes.

### Incompatibilities

None.

### Known bugs and problems

None.

## Mnesia 3.5.4

### Improvements and new features

None.

### Fixed Bugs and malfunctions

- This bug is an old one, but in this release the file descriptors are handled better. Notes copied from 3.1.1.  
At startup, when Mnesia recreated the database from a fallback, lots of files was simultaneously opened. For large (or rather medium) size databases this caused the Erlang runtime system to encounter its hard-coded limit of 256 open ports. Mnesia no longer consumes as many open files during startup in order to avoid the system limit. The error message looked like: **\*\* FATAL \*\***  
`mnesia_tm crashed: {"Cannot start from fallback", {'EXIT', {badarg, {os,ask_driver, [open_port, [{spawn,os_drv_}, []]}}}}`

### Incompatibilities

None.

### Known bugs and problems

None.

## Mnesia 3.5.3

### Improvements and new features

None.

## Fixed Bugs and malfunctions

- Meta table information got inconsistent when a table was copied from a node which crashed during the table transfer. This bug could lead to that the table was never loaded.  
Own Id: OTP-2852.

## Incompatibilities

None.

## Known bugs and problems

None.

## Mnesia 3.5.2

### Improvements and new features

None.

### Fixed Bugs and malfunctions

- Dumping an ets table into a dets table caused unexpected memory consumption when the dets table resided in the ram\_file'r.

### Incompatibilities

None.

### Known bugs and problems

None.

## Mnesia 3.5.1

### Improvements and new features

None.

### Fixed Bugs and malfunctions

- Records was sometimes skipped when iterating over a backup with the default backup module `mnesia_backup`. The error affected `mnesia:restore/2`, `mnesia:install_fallback/1` and `mnesia:traverse_backup/4,6`.  
Own Id: OTP-2819.  
Aux Id: seq 1261.

## Incompatibilities

None.

## Known bugs and problems

None.

## Mnesia 3.5

### Improvements and new features

None.

### Fixed Bugs and malfunctions

- Copying of tables between nodes has been improved. Dirty updates performed while the table is copied could sometimes be lost.  
Own Id: OTP-2817.  
Aux Id: seq 1258.
- The outcome (abort/commit) of heavy weight transactions was some time lost after node crashes. A new log format has been introduced and the processing of decisions in the log has been improved.
- Minor bug fixed regarding iteration of checkpoints with `disc_only_copies`-tables. It could cause inconsistent backups and inconsistent replicas if the internal hash tables grew or shrank during the iteration.
- Transient schema information about indecies was not cleaned up after `mnesia:del_table_copy/3`. This caused miscellaneous index related errors in successive operations.
- Recovery of `mnesia:create_table/2` during startup crashed on nodes without any replica.

## Incompatibilities

None.

## Known bugs and problems

None.

## Mnesia 3.4.1

### Improvements and new features

- The documentation has been enhanced.

## Fixed Bugs and malfunctions

- A bug that caused `mnesia:wread/1` to hang infinitely has been fixed. It was a unique situation when the transaction, which invoked `mnesia:wread/1`, was waiting for a write lock and a remote transaction was holding a `sticky_write` lock on the same record. If the remote node crashed in that situation and the remote node holding the `sticky_write` lock was one of the first nodes in the `where_to_write` list of nodes, it could cause `mnesia:wread/1` to never return, instead of simply re-running the transaction.
- Application processes of Mnesia were neither unlinked after invocation of `mnesia:unsubscribe/1` nor shutdown. This caused spurious exit signals to be sent to the subscribing processes, when Mnesia was terminated. Subscribers of `system` events never receive a `{mnesia_down, node()}` event if a local node is stopped normally. These bugs has been fixed now.
- Some minor bugs regarding the `record_name` feature (that was introduced in 3.4) has been fixed.

## Incompatibilities

None.

## Known bugs and problems

None.

## Mnesia 3.4

### Improvements and new features

**Record name may differ from table name** From this release onwards, the record name of records stored in Mnesia may differ from the name of the table that they are stored in. In order to use this new feature the table property `{record_name, Name}` has been introduced. If this property is omitted when the table is created, the table name will be used as record name. For example, if two tables are created like this:

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

it would be possible to store subscriber records in both of them:

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

In order to enable usage of this new support for `record_name` new functions have been added to the Mnesia API:

```

mnesia:dirty_write(Tab, Record)
mnesia:dirty_delete(Tab, Key)
mnesia:dirty_delete_object(Tab, Record)
mnesia:dirty_update_counter(Tab, Key, Incr)
mnesia:dirty_read(Tab, Key)
mnesia:dirty_match_object(Tab, Pattern)
mnesia:dirty_index_match_object(Tab, Pattern, Attr)

mnesia:write(Tab, Record, LockKind)
mnesia:delete(Tab, Key, LockKind)
mnesia:delete_object(Tab, Record, LockKind)
mnesia:read(Tab, Key, LockKind)
mnesia:match_object(Tab, Pattern, LockKind)
mnesia:all_keys(Tab)
mnesia:index_match_object(Tab, Pattern, Attr, LockKind)
mnesia:index_read(Tab, SecondaryKey, Attr)

LockKind ::= read | write | sticky_write | ...

```

The old corresponding functions still exists, but are merely a syntactic sugar for the new ones:

```

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).
```

The earlier function semantics remain unchanged.

Use the function `mnesia:table_info(Tab, record_name)` to determine the record name of a table.

If the name of all tables equals the record name that the table hosts everything is backward compatible. But if the new `record_name` feature is used this may affect old existing applications:

- Functions that provide read access to Mnesia tables may now return records with names that differ from the table name. Make sure your code is able to handle this.
- The backup format is slightly different and this may affect users of the `mnesia:traverse_backup/4,6` functions. When iterating over a backup, the record name is always equal to the table name regardless of the `record_name` setting. Make sure your code is able to handle this.
- The format of table subscription event is slightly different and this may affect users of the `mnesia:subscribe/1` function and user installed `event_module`'s. When a process receives table events, the record name is always equal to the table name regardless of the `record_name` setting.

**New function `mnesia:lock/2`** A new locking function has been introduced:

```
mnesia:lock(LockItem, LockKind)

LockItem ::= {table, Tab} | {global, Item, Nodes} | ...
LockKind ::= read | write | ...
```

The old table locking functions still exists, but are now merely a syntactic sugar for the new functions:

```
mnesia:read_lock_table(Tab) ->
    mnesia:lock({table, Tab}, read).

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

**New function `mnesia:activity/2,3,4`** In the Mnesia API there are some functions whose semantics depends of the execution context:

```
mnesia:lock(LockItem, LockKind)
mnesia:write(Tab, Rec, LockKind)
mnesia:delete(Tab, Key, LockKind)
mnesia:delete_object(Tab, Rec, LockKind)
mnesia:read(Tab, Key, LockKind)
mnesia:match_object(Tab, Pat, LockKind)
mnesia:all_keys(Tab)
mnesia:index_match_object(Tab, Pat, Attr, LockKind)
mnesia:index_read(Tab, SecondaryKey, Attr)
mnesia:table_info(Tab, InfoItem)
```

if these functions are executed within a `mnesia:transaction/1,2,3`, locks are acquired, atomic commit is ensured etc. If the same functions are executed within the context of `mnesia:async_dirty/1,2`, `mnesia:sync_dirty/1,2` or `mnesia:ets/1,2` their semantics are different. Although this is not entirely new, new functions have been introduced:

```
mnesia:activity(ActivityKind, Fun)
mnesia:activity(ActivityKind, Fun, Args)
mnesia:activity(ActivityKind, Fun, Module)
mnesia:activity(ActivityKind, Fun, Args, Module)

ActivityKind ::= transaction |
               {transaction, Retries} |
               async_dirty |
               sync_dirty |
               ets
```

Depending on the `ActivityKind` argument, the evaluation context will be the same as with the functions:

`mnesia:transaction`,  
`mnesia:async_dirty`,  
`mnesia:sync_dirty` and  
`mnesia:ets` respectively. The `Module` argument provides the name of a callback module that will implement the `mnesia_access` behavior. It must export the functions:

```
lock(ActivityId, Opaque, LockItem, LockKind)
write(ActivityId, Opaque, Tab, Rec, LockKind)
delete(ActivityId, Opaque, Tab, Key, LockKind)
delete_object(ActivityId, Opaque, Tab, Rec), LockKind
read(ActivityId, Opaque, Tab, Key, LockKind)
```

```
match_object(ActivityId, Opaque, Tab, Pat, LockKind)
all_keys(ActivityId, Opaque, Tab, LockKind)
index_match_object(ActivityId, Opaque, Tab, Pat, Attr, LockKind)
index_read(ActivityId, Opaque, Tab, SecondaryKey, Attr, LockKind)
table_info(ActivityId, Opaque, Tab, InfoItem)
```

ActivityId ::=

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

Opaque ::=

An opaque data structure which is internal to Mnesia.

`mnesia` and `mnesia_frag` are examples of callback modules. By default the `mnesia` module is used as callback module for accesses within “Mnesia activities”.

For example invoke the function `mnesia:read(Tab, Key, LockKind)`, and the corresponding `Module:read(ActivityId, Opaque, Tab, Key, LockKind)` will be invoked to perform the job (or it will pass it on to `mnesia:read(ActivityId, Opaque, Tab, Key, LockKind)`).

A customized callback module may be used for several purposes, such as providing triggers, integrity constraints, run time statistics, or virtual tables. The callback module does not have to access real Mnesia tables, it is a free agent provided the callback interface is fulfilled.

The context sensitive function `mnesia:table_info/2` may be used to provide virtual information about a table. This function enables the user to perform Mnesosyne queries within an activity context with a customized callback module. By providing table indices information and other Mnesosyne requirements, Mnesosyne can be used as an efficient generic query language for access of virtual tables.

Please, read the “`mnesia_access` callback behavior” in Appendix C for a code example from the `mnesia_frag` module.

**New configuration parameter `access_module`** The new configuration parameter `access_module` has been added. It defaults to the atom `mnesia`, but may be set to any module that fulfills the callback interface with `mnesia_access` behavior.

The `mnesia:activity` functions will use the `access_module` as a callback module if it not is explicitly overridden by the `Module` argument.

Use `mnesia:system_info(access_module)` to determine the actual `access_module` setting.

## Fixed Bugs and malfunctions

- A bug regarding master nodes has been fixed. If all nodes only have a remote master nodes set for a table, the intended behavior is that no node should take the initiative to load the table. This now works as intended.

## Incompatibilities

None as long as all tables only host records with the same name as the table name. Please, read the chapter Improvements and new features about the potential inconsistencies.

## Known bugs and problems

None.

## Mnesia 3.3

### Improvements and new features

#### Fixed Bugs and malfunctions

- `mnesia:change_table_copy/3` on schema from `disc_copies` to `ram_copies` did not work on non Solaris platforms.  
Own Id: OTP-2364.
- Indices of `disc_only_tables` did not work.  
Own Id: OTP-2363.
- Shutdown of Mnesia took 30 seconds due to an internal deadlock in the `application_controller`. This problem is now circumvented by Mnesia.  
Own Id: OTP-2664.
- `mnesia:set_master_nodes/1,2` may now be invoked off-line, i.e. even if Mnesia happens to be stopped.  
Own Id: OTP-2425.

#### Incompatibilities

None.

## Known bugs and problems

No new problems or bugs. See previous release notes.

## Mnesia 3.2

### Improvements and new features

- The function `mnesia:restore/2` has eventually been implemented. The documentation of the function has been adopted to the semantics of the actual implementation. Please, read the documentation about the details.  
Own Id: OTP-1560  
Own Id: OTP-1736  
Own Id: OTP-1824

## Fixed Bugs and malfunctions

- Mnesia crashed if a replica was removed with `mnesia:delete_table_copy/2` before the load of the table was completed on the actual node.  
Own Id: OTP-2519.  
Aux Id: seq 893.
- Long table names was truncated in the printout from `mnesia:info/0`.  
Own Id: OTP-2529.  
Aux Id: seq 895.

## Incompatibilities

None.

## Known bugs and problems

No new problems or bugs. See previous release notes.

## Mnesia 3.1.1

This release is a minor release and the release notes describes the difference between version 3.1.1 and version 3.1 of Mnesia.

## Improvements and new features

None.

## Fixed Bugs and malfunctions

- At startup, when Mnesia recreated the database from a fallback, lots of files was simultaneously opened. For large (or rather medium) size databases this caused the Erlang runtime system to encounter its hard-coded limit of 256 open ports. Mnesia no longer consumes as many open files during startup in order to avoid the system limit. The error message looked like: **\*\* FATAL \*\***  
`mnesia_tm crashed: {"Cannot start from fallback", {'EXIT', {badarg, {os,ask_driver, [open_port, [{spawn,os_drv_}, []]}}}}`  
Own Id: OTP-2534.  
Aux Id: seq 907.

## Incompatibilities

None.

## Known bugs and problems

No new problems or bugs. See previous release notes.

## Mnesia 3.1

### Improvements and new features

- A new configuration parameter `ignore_fallback_at_startup` has been added. It defaults to `false`, but if it is set to `true` it causes Mnesia to ignore the potential fallback that may have been installed with `mnesia:install_fallback/2`. The new configuration parameter enables Mnesia to start with the old database even if a fallback is present.

Own Id: OTP-2530.

Aux Id: seq 903.

Aux Id: HA86394.

- Previously, Mnesia terminated itself on all nodes if a fallback was installed and Mnesia on another node was terminated. It is now possible to bypass this behavior.

When Mnesia's event handler receives a `{mnesia_down, Node}` event it will check if there is any fallback active and then possibly terminate the local Mnesia system. If the user changes the configuration parameter `event_module` to an own module, the new module may simply omit to terminate Mnesia.

Own Id: OTP-2530.

Aux Id: seq 903.

Aux Id: HA86394.

### Fixed Bugs and malfunctions

- When Mnesia encountered a fatal error, Mnesia would terminate itself on all nodes. Now Mnesia will only terminate itself on the local node as intended.

### Incompatibilities

None.

### Known bugs and problems

No new problems or bugs. See previous release notes.

## Mnesia 3.0

This release is a major release and the release notes describes the difference between version 3.0 and version 2.3. 3.0 is classified as major release due to the issues described in the chapter about incompatibilities described below.

## Improvements and new features

- Mnesia's internal storage format on disc has been made more future safe.
- A new attribute in the schema table has been added. Called `user_properties`, it enables applications to store its own meta data about a table together with the Meta data that is built into Mnesia. The `user_properties` can either be set when the table is created, by stating a `{user_properties, PropList}` tuple in the `CreateList` given to `mnesia:create_table(Tab, CreateList)` or later with `mnesia:write_table_property(Tab, Prop)`.  
The user property `Prop` must be a record (a tagged tuple). Subsequent calls to `write_table_property/2` with the same tag will overwrite the old setting. The property record may have any number of fields.  
The user property can be read with `mnesia:read_table_property(Tab, PropName)`. It returns the user property record or exits with `{no_exists, {Tab, PropName}}`.  
The user property can be deleted with `mnesia:delete_table_property(Tab, PropertyName)`.  
All user defined properties for a table can be obtained with `mnesia:table_info(Tab, user_properties)`. It returns a list of the stored property records.
- All table properties that are normally obtained singularly with `mnesia:table_info(Tab, Prop)`, can be read with one single call to `mnesia:table_info(Tab, all)`. It returns a list of `{PropName, PropVal}` tuples.
- All system properties that are normally obtained with `mnesia:system_info(Prop)` singularly, can be read with one single call to `mnesia:system_info(all)`. It returns a list of `{PropName, PropVal}` tuples. If Mnesia is not running, only a subset of the properties will be returned.
- The module `mnesia_registry` has been added. It contains functions which support the customized creation of backing storage tables for registries in `erl_interface`.

## Fixed Bugs and malfunctions

No serious bugs or malfunctions.

## Incompatibilities

Mnesia 3.0 is primary developed for OTP R4, but is still backward compatible with the OTP R3 platform.

The internal database format on disc has been made more future safe. It has also been altered in order to cope with the newly introduced features. The special upgrade procedure is as follows:

- First of all, a full backup must be performed with the old Mnesia system.
- Then the backup must be installed as fallback, preferably by the new version of Mnesia.
- At last the new version of Mnesia may be started.

Mnemosyne has been made into an own separate application. The new application is called `mnemosyne`. Please, read its release notes. This application split implies a few incompatibilities:

- The compiler directive:

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

that was mandatory in all Erlang modules that contains embedded Mnemosyne queries has been replaced with:

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

During an interim period, both compiler directives will be supported. But in a future release the backward compatibility directive will be removed.

- At the startup of Mnesia, the Mnemosyne statistics process `mnemosyne_catalog` has automatically been started by Mnesia. This is not the case anymore. The `mnemosyne` application must be started separately after the start of the `mnesia` application.

However, a temporary configuration parameter `embedded_mnemosyne` has been added to allow the automatic start of Mnemosyne. By default `embedded_mnemosyne` is set to `false`, but if it is set to `true` Mnesia will start Mnemosyne as a supervised part of the Mnesia application as it did in previous releases.

## Known bugs and problems

None of these are newly introduced.

- Indices of `disc_only_tables` does not work.  
Own Id: OTP-2363.
- Changing the storage type of the schema from `disc_copies` to `ram_copies` does not work on Windows systems.  
Own Id: OTP-2364.
- Mnesia will not detect if two nodes are sharing the same Mnesia directory. When two nodes are using the same Mnesia directory the result is totally unpredictable.
- `mnesia:restore/1` is not implemented.  
Own Id: OTP-1736.

## Mnesia 2.3

### Improvements and new features

None.

### Fixed Bugs and malfunctions

- Mnesia will run into an internal deadlock occasionally, at startup if several nodes were brutally terminated simultaneously (eg. after a power failure).  
Own Id: OTP-2501.  
Aux Id: seq 879, HA85350

### Incompatibilities

None.

### Known bugs and problems

No new ones. See previous release notes.

## Mnesia 2.2

### Improvements and new features

None.

### Fixed Bugs and malfunctions

- Mnemosyne's internal query evaluator processes continued to perform work, even after the initiating application process (the process that performed the call to `mnesia:transaction`) died. This would lead to dangling locks which were never released.  
Own Id: OTP-2390, OTP-2340.  
Aux Id: seq 797
- Mnemosyne's internal query evaluator processes continued to perform work, even after the enclosing transaction had started its commit/abort work. This would lead to dangling locks which was never released.
- Mnemosyne's internal query evaluator processes hang when Mnesia detected a potential deadlock and tried to restart the transaction. This led to dangling locks which were never released.
- Mnemosyne's internal query evaluator processes hang when a remote node was terminated while they were waiting for locks. This led to dangling locks which were never released.

### Incompatibilities

None.

### Known bugs and problems

- The evaluation of Mnemosyne queries exhibits undefined behavior if used in conjunction with nested transactions.
- The evaluation of Mnemosyne queries exhibits undefined behavior if the involved tables are updated after the cursor has been initiated.
- Mnemosyne queries should not be used on disc-only tables since the optimizer can't handle such tables.
- Mnesia does not detect if two nodes are sharing the same Mnesia directory. When two nodes are using the same Mnesia directory anything can happen. The result is totally unpredictable.
- `mnesia:restore/1` is not implemented.  
Own Id: OTP-1736.

## Mnesia 2.1.2

### Improvements and new features

None.

### Fixed Bugs and malfunctions

- The `mnesia_locker` process EXIT'ed when trying to move a sticky lock from one node to another. As a result, Mnesia would be terminated.

## Incompatibilities

None.

## Known bugs and problems

See previous release notes.

## Mnesia 2.1.1

### Improvements and new features

None.

### Fixed Bugs and malfunctions

- A bug related to remote load of tables has been fixed. If Mnesia was in the process of loading tables from another node and that node terminated before the remote loading of tables was finished, Mnesia did not automatically load the remaining `ram_copies` tables.  
Own Id: OTP-2339.  
Aux Id: seq 795

## Incompatibilities

None.

## Known bugs and problems

See previous release notes.

## Mnesia 2.1

### Improvements and new features

- A new feature for load regulation of the `mnesia_dumper` processes has been introduced. It is activated by setting the configuration parameter `dump_log_load_regulation` to `true`. This is a temporary solution while waiting for a fair scheduler algorithm in the Erlang emulator. The `mnesia_dumper` process performs many costly BIF invocations and ought to pay for this. But since the Emulator does not handle this properly we must compensate for this with some form of load regulation ourselves in order to not steal all computation power in the Erlang Emulator and make other processes starve or trigger heart to restart the entire node.
- A new example has been introduced. `mnesia_meter` can be used to obtain performance figures for some API functions.

## Fixed Bugs and malfunctions

- A bug related to `master_nodes` has been fixed. The table was not loaded locally, when the local node was set to be master node for a replicated table which was active on another node during startup of Mnesia on the local node.
- Another bug related to `master_nodes` has also been fixed. The table was not loaded from the local node, when the master node was set to the local node for a replicated table, with `ram_copies` as local `storage_type` and `disc_copies` or `disc_only_copies` on the other nodes.
- A bug related to remote load of tables has been fixed. Mnesia could refuse to copy the table from another node if an active checkpoint was deactivated in a critical period at the end of the copy operation.
- In earlier releases `mnesia:async_dirty`, `mnesia:sync_dirty` and `mnesia:ets` caught EXIT's and returned them as `{'EXIT', Reason}` when something went wrong. This behavior was neither intended nor documented. Now the (mis-)behavior has been corrected to do `exit(Reason)` as expected.
- Removed a deadlock situation which occurred when fallbacks was installed/uninstalled while Mnesia was stopped.
- Removed a potential deadlock situation which could occur if Mnesia was terminated on a remote node during commit of "heavyweight" transactions.  
Under some circumstances the bug could cause a crash in the application process with an error message like this: `{{case_clause, {acc_pre_commit, {tid, 523, <0.216.0>}, <87.355.0>}}, {mnesia_tm, t_begin, [#Fun, [], 2, infinity]}}`  
Own Id: OTP-2106.  
Aux Id: seq 607
- Improved recovery of nested transactions when the coordinator process died during commit.
- Better error handling of `mnesia:add_table_copy/3` and `mnesia:move_table_copy/3` when the application tried to add disc resident replicas on disc less nodes. The effect of the bug was that Mnesia could crash with a error message like this: `mnesia_init crashed: {has_no_disc, ...}`
- Removed a deadlock situation that occurred during startup when tables was loaded too early.
- `mnesia_init` could crash if it failed to copy a table from one node and then tried to copy the table from another node.
- The semantics of `master_nodes` have been made more clear. Earlier, unusual things happened in odd situations, especially in conjunction with `mnesia:force_load_table/1`.
- The handling of transaction decisions has been optimized. Fewer messages are sent. The decision log is now only dumped during startup.
- `mnesia_recover` could crash when decision table was saved to disc. The `ets:tab2file/2` is no longer available due to its unsafe behavior.
- `mnesia_recover` could crash during shutdown of Mnesia. Mnesia's internal overload detection mechanism was not synchronized with the shutdown protocol.
- Mnesia could crash while it performed its protocol negotiation with other nodes. The problem occurred when Mnesia tried to detect if the network had been partitioned.  
Own Id: OTP-2234.  
Aux Id: seq 702.
- The algorithm which triggered dumps of the transaction log has been improved. Previously the log would not dumped even if it was necessary. The problem occurred when the dump thresholds were exceeded while the previous dump was still in progress.
- `disc_only_copies` can now be managed via SNMP.
- Locks could be left dangling if the user process crashed in a critical situation.

- The compiled code for the example programs does not reside on `mnesia/ebin` anymore. It can be found on the `mnesia/examples` directory instead.

## Incompatibilities

- Mnesia 2.1 is not bug compatible with earlier releases, see the chapter `Fixed Bugs and malfunctions` about changed behaviors.

## Known bugs and problems

- Mnesia does not detect if two nodes are sharing the same Mnesia directory. When two nodes are using the same Mnesia directory the result is totally unpredictable.
- `mnesia:restore/1` is not implemented.  
Own Id: OTP-1736.
- Mnemosyne queries should not be used on disc-only tables since the optimizer can't handle such tables.

## Mnesia 2.0.2

### Improvements and new features

The performance of the Mnemosyne catalog is improved. There are now some parameters per table available for tuning. Two functions are introduced for this:

```
mnemosyne_catalog:set_parameter(Table, Name, Value)
mnemosyne_catalog:get_parameter(Table, Name)
```

Both return the present value. They may change in a future releases! The possible Names are:

Name	Values	Default	Description
<code>do_local_upd</code>	yes no	yes	Collect statistics for this table on this node
<code>min_upd_interval</code>	integer seconds	10	Minimum allowed interval between two updates
<code>upd_limit</code>	Percent	10	New statistics are collected when more than <code>upd_limit</code> % of the table is updated
<code>max_wait</code>	integer millisec	1000	Maximum time to wait for the initial call from the optimizer to the catalog server

Table 1.4: Mnemosyne configuration parameters

## Fixed Bugs and malfunctions

- Checkpoints could under some circumstances become inconsistent, if the involved tables were updated during the activation of a checkpoint. This could lead to inconsistent backups and databases.  
Own Id: OTP-1733.
- Secondary indices were not handled correctly, `mnesia:index_read/3` could return incorrect result.  
Own Id: OTP-2083.
- The `mnesia_locker` and `mnesia_tm` processes did under some circumstances crash, if tables were updated while their definitions were changed in schema transactions.  
Own Id: OTP-1734.
- Problems that occurred when applications tried to run schema transactions directly after Mnesia was started (but before the schema was merged), have been fixed. Now Mnesia first merges its schema with the schema on other nodes before it starts completely.
- The Mnemosyne catalog function (which gathers statistics for the query optimizer) could stop working in some situations on disk-less nodes. This is corrected.  
Own Id: OTP-1982.  
Aux Id: seq 466.

## Incompatibilities

- The default behavior of Mnesia's event handler has been slightly changed. An `{inconsistent_database, Context, Node}` event will not automatically cause `mnesia:set_master_node` to be invoked anymore. However, an error is still reported to the `error_logger`. For advanced users the opportunity to recover from an inconsistent database remains, by explicit use of the function `mnesia:set_master_node`.

## Known bugs and problems

- Mnesia does not detect if two nodes are sharing the same Mnesia directory. When two nodes are using the same Mnesia directory anything may happen. The result is totally unpredictable.
- `mnesia:restore/1` is not implemented.  
Own Id: OTP-1736.
- Mnemosyne queries should not be used on disc-only tables since the optimizer can't handle such tables.

## Mnesia 2.0.1

### Improvements and new features

None.

## Fixed Bugs and malfunctions

- Customized backup modules would suffer from a bug which caused by an update to the `Opaque` data structure to be lost. The `Opaque` data is a part of the Mnesia backup callback interface. However, Mnesia's default backup module did work despite the bug.
- Due to a timing bug, Mnesia could crash at startup, if tables was accessed before the schema had been merged. The error message looked like `** FATAL ** mnesia_init crashed: {no_exists, {chanRegPpchTable, storage_type}}`. The earlier bugfix made in Mnesia 1.3 did fix one timing bug, but it turned out to be yet another timing bug also :-(.  
Aux Id: seq 384.
- A reference to the non-existent Mnesia module `mnesia_reconfig` has been removed from the configuration file `mnesia.app`.  
Own Id: OTP-1888.
- Enhanced synchronization of asynchronous dirty writes that are performed on non local tables. The bug symptom would enable a dirty read to read the table before a previous dirty update.  
Own Id: OTP-1731.

## Incompatibilities

None.

## Known bugs and problems

- Checkpoints may, under some circumstances, be inconsistent if the involved tables are updated during the activation of checkpoint.  
Own Id: OTP-1733.
- `mnesia_locker` may, under some circumstances, crash if tables are updated while their definitions are being changed in schema transactions.  
Own Id: OTP-1734.
- Mnesia does not detect if two nodes are sharing the same Mnesia directory. When two nodes are using the same Mnesia directory anything may happen. The result is totally unpredictable.
- `mnesia:restore/1` is not implemented.  
Own Id: OTP-1736.

## Mnesia 2.0

The release notes describe the difference between version 2.0 and version 1.3.2 of Mnesia. 2.0 is classified as major release of Mnesia due to changes in the internal database format on disc, see the chapter about incompatibilities for further details.

## Improvements and new features

- It is now possible to have read only tables. By default a table is open for both read and write access, but if the property `{access_mode, read_only}` is given to `mnesia:create_table/2` it will be set to be read only. It may be changed later with the new function `mnesia:change_table_access/2`. Normally at startup, Mnesia will primary load its tables from other nodes where a replica is already active. But for those, possibly large, tables that are set to be read only, the local replica will always be loaded from disc without the risk of introducing

inconsistency to the database. The current allowed access mode of a table may be read with the function `mnesia:table_info/2`.

- The load order of tables may now be controlled. By default all tables are given the load order priority 0, but if the property `{load_order, Integer}` is given to `mnesia:create_table/2` it will be set to the stated priority. The tables with the highest load order priority will be loaded first. The load order may later be changed with the new function `mnesia:change_table_load_order/2`. In systems with many and/or large tables, the applications now have the opportunity to gain early availability to some important applications by ensuring the tables are loaded early. The load order between tables with the same load order priority is still undefined. The current load order priority of a table may be read with the function `mnesia:table_info/2`.
- A new configuration parameter has been introduced. It is called `max_wait_for_decision`. By default it is set to the atom `infinity`, which implies that if Mnesia upon startup encounters a “heavyweight transaction” whose outcome is unclear, the local Mnesia will wait until Mnesia is started on some or possibly all of the other nodes that were involved in the interrupted transaction. This is a very rare situation, but when/if it happens, Mnesia will not guess if the transaction on the other nodes was committed or aborted, it will wait until it knows the outcome and then act accordingly.  
However, it is now possible to force Mnesia to finish the transaction, even if the outcome of the transaction for the moment is unclear. After `max_wait_for_decision` milliseconds, Mnesia will commit/abort the transaction and continue with the startup. This may lead to transaction committing on some nodes and aborting on others. If the schema was updated in the transaction, the inconsistency may be fatal.
- Mnesia tries to guarantee that the database always is consistent, but there are a few situations when inconsistency can be introduced:
  - When two Erlang nodes loose contact due to a communication failure. On both nodes Mnesia will think that the other node is stopped and its applications may possibly continue to update the tables on both nodes in parallel, while the nodes are unconnected. The database has then possibly become inconsistent and Mnesia will detect that either when the Erlang nodes re-gain contact with each other or at startup (on one of the two nodes).
  - When the application has fooled Mnesia to make a wrong transaction decision by setting `max_wait_for_decision` to something other than `infinity`. The database will become inconsistent. Mnesia will detect this later when the remaining nodes (where conflicting transaction decisions has been made) start.
  - When the application has deliberately forced Mnesia to load a table with `mnesia:force_load_table/1`. The database could possibly become inconsistent, but the effect to Mnesia will be minimal.
- When Mnesia detects that the database is possibly inconsistent a newly introduced event will be generated: `{inconsistent_database, Context, Node}`. It the default handling of that event which will pick a `MasterNode` from `mnesia:system_info(db_nodes)` and invoke `mnesia:set_master_node([MasterNode])`.
- The normal table load algorithm, loads replicas depending on whether Mnesia has had the opportunity to perform updates on tables on other nodes, while Mnesia was stopped on the local nodes. But this behavior may now be overridden by the new function `mnesia:set_master_node(Table, MasterNodes)`. At startup Mnesia will primary load replicas from the `MasterNodes` that are defined for a table. However, if there are no `MasterNodes` defined for the table, the normal table load algorithm will apply. It is also possible to set the same master node(s) for all tables by invoking the function `mnesia:set_master_node(MasterNodes)`. The master nodes of a table can be read with the function `mnesia:table_info(Tab, master_nodes)`. A list of all local tables with at least one defined master node can be read with the function `mnesia:system_info(master_node_tables)`. The master node feature is reset, by invoking

`mnesia:set_master_node/1,2` to the empty list [].

- Mnesia now has the ability to set a master node.  
Own Id: OTP-1456.  
Aux Id: HA52431.

## Fixed Bugs and malfunctions

- `mnesia:system_info(db_nodes)` returns the nodes which 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, eg. with `mnesia:add_table_copy/3`. The function can be invoked even if Mnesia is not yet running. There are three bugs related to `db_nodes` that have been fixed:  
The semantics was poorly described,  
It returned the wrong result on disc-less nodes and under some circumstances the `storage_type` for the schema was unknown, which made it impossible to create tables from such a node.  
Own Id: OTP-1790.  
Aux Id: seq 348.
- Mnesia crashed if it was stopped during the load of an SNMP-table.  
Own Id: OTP-1732, OTP-1650.
- Transactions acquiring sticky locks could be hanging indefinitely, if another node already had the lock and crashed during the hand over of the lock.  
Own Id: OTP-1730.
- Spurious {`mnesia_down`, Node} messages were sent to application processes after they had committed a transaction.  
Own Id: OTP-1735.
- The configuration parameter `dump_time_threshold` was ignored.  
Own Id: OTP-1789.  
Aux Id: seq 353.
- `ram_copies` replicas were not automatically loaded at startup.  
Own Id: OTP-1856.  
Aux Id: seq 411.
- `mnesia:force_load_table/1` would load the local replica if it was invoked before the schema merge had been completed.
- The database could get inconsistency if the application process crashed while committing its transaction. The transaction recovery has been enhanced.
- Mnesia now copes with partitioned networks.

## Incompatibilities

Mnesia 2.0 is primary developed for OTP R3, but is still backward compatible with the OTP R1D and OTP R2D platforms.

The internal database format on disc has been changed in order to cope with the new features that have been introduced. This implies a special upgrade procedure:

- First of all, a full backup must be performed with the old Mnesia system.
- Then the backup must be installed as fallback, preferably by the new version of Mnesia.
- At last the new version of Mnesia may be started.

## Known bugs and problems

- Mnesia does not detect if two nodes are sharing the same Mnesia directory. When two nodes are using the same Mnesia directory anything may happen. The result is totally unpredictable.
- `mnesia:restore/1` is not implemented.

## Mnesia 1.3.2

### Improvements and new features

None.

### Fixed Bugs and malfunctions

- When Mnesia goes down while a transaction is committing on a node involved in a distributed transaction, the locks could remain indefinitely. This serious bug was introduced in Mnesia 1.3.1.
- Local load of `ram_copies` tables which does not have any `disc_copies` or `disc_only_copies`, is now immediate. They are either loaded from another active replica or locally if no replica is active.
- Backups of `ram_copies` tables when `ram_overrides_dump` was set to `false` was inconsistent.
- Backups of tables which were updated during the backup were inconsistent.

## Mnesia 1.3.1

### Improvements and new features

- All changes of replica storage types (with `mnesia:change_table_copy_type/3`) does not require that all nodes with schema on disc are up and running anymore. The storage type may be changed from `ram_copies` to `disc_copies` or from `disc_copies` to `disc_only_copies`. But all other changes have the same restrictions as before.

### Fixed Bugs and malfunctions

- A few bugs related to release of locks when Mnesia (on other nodes) went down, have been fixed. The symptom of the bugs was an infinite hang.
- A bug related to the hand over of sticky locks has been fixed. The symptom of the bug was indefinite hanging.
- `mnesia:dirty_update_counter/2` did not work for `snmp-tables`. The symptom of the error was 

```
{{case_clause,update_counter},  
{mnesia_snmp_hook,update,[update_counter,<0.58.0>,1,[1]}}
```

.  
Own Id: OTP-1725.

### Incompatibilities

None.

## Known bugs and problems

See notes about release 1.3

## Mnesia 1.3

This release is a minor bugfix release and the release notes describes the difference between version 1.3 and version 1.2.3 of Mnesia.

## Improvements and new features

None.

## Fixed Bugs and malfunctions

- Due to timing bug, Mnesia could crash at startup under some circumstances when it exchanged definitions of tables that was created on remote nodes while the local node was down. The error message looked like **\*\* FATAL \*\* mnesia\_init crashed: {no\_exists, {eve\_node\_info, storage\_type}}**.  
Aux Id: HA55473.

## Incompatibilities

None.

## Known bugs and problems

See notes about release 1.2.3

## Mnesia 1.2.3

## Improvements and new features

None.

## Fixed Bugs and malfunctions

- Due to a timing bug when creating a table (`mnesia:create_table/2`), Mnesia could crash under some circumstances. The error message looked like **\*\* FATAL \*\* mnesia\_init crashed: {no\_exists, {ftmFaultState, storage\_type}}**.

## Incompatibilities

None.

## Known bugs and problems

See notes about release 1.2.2.

## Mnesia 1.2.2

### Improvements and new features

None.

### Fixed Bugs and malfunctions

- It was not possible change the storage type of the schema table (from ram\_copies to disc\_copies) if it was performed from a remote node by using `mnesia:change_table_copy_type/3`. Now the function may be invoked from any node running Mnesia.
- While the storage type of the schema table, was changed from ram\_copies to disc\_copies (with `mnesia:change_table_copy_type/3`), Mnesia entered a short period of internal inconsistency. If other ongoing transactions was ended during that period, their outcome was logged to disk before the decision log was initiated. When this situation occurred it would cause Mnesia to crash.

### Incompatibilities

None.

### Known bugs and problems

See notes about release 1.2.1.

## Mnesia 1.2.1

### Improvements and new features

None.

### Fixed Bugs and malfunctions

- Mnesia should be able to repair its log files after file system crashes. This is normally done by the `disc_log` module, but if the log file is so badly damaged that `disc_log` does not regard the file as a log file, Mnesia will now delete the file and create a new fresh one. This will only happen if the configuration parameter `auto_repair` is set to true, otherwise no tries to repair the log file will be performed.  
Aux Id: HA49139.
- Ambiguous storage type for tables is not allowed. Mnesia did not check this in earlier releases when tables was created with `mnesia:create_table/2`. Now Mnesia will reject the creation of such tables. This may have led to confusing behavior in many situations for tables with ambiguous storage types. These tables may have become inconsistent, not loaded at all, loaded on some nodes but not on all nodes etc, etc.  
Aux Id: HA52368.
- Tables with the `local_content` property set to true, were not handled properly at startup. The most obvious effect of this was that these tables were not loaded on some nodes.  
Aux Id: HA52368.

## Incompatibilities

None.

## Known bugs and problems

See notes about release 1.2.

## Mnesia 1.2

### Improvements and new features

None.

### Fixed Bugs and malfunctions

- Fatal error when updating schema at Mnesia startup.  
Own Id: OTP-1441.
- The definition of a newly created table was not propagated to all nodes if some of them were down when the table was created. The nodes were not intended to hold an own replica never had a table definition once they were started. From the application's point of view the table seemed to not exist on the nodes that lacked the table definition.
- Incompatibility bug fixed. The bug made it impossible to install older backups as fallback if they contained SNMP tables.

## Incompatibilities

None.

## Known bugs and problems

See notes about release 1.1.1.

## Mnesia 1.1.1

This section describes the changes made to Mnesia in the 1.1.1 version of Mnesia. This release is a minor upgrade from 1.1.

### Improvements and new features

- -

### Fixed Bugs and malfunctions

- Mis-spelling of backup format version corrected. The bug made it impossible to install older backups as fallback.

## Incompatibilities

None.

## Known bugs and problems

See notes about release 1.1.

## Mnesia 1.1.1

This section describes the changes made to Mnesia in the 1.1.1 version of Mnesia. This release is a minor upgrade from 1.1.

## Improvements and new features

- -

## Fixed Bugs and malfunctions

- Spelling error in the backup format corrected. The bug made it impossible to install older backups as fallback.

## Incompatibilities

None.

## Known bugs and problems

See notes about release 1.1.

## Mnesia 1.1

This release is a normal release for general use and it comes with full documentation. The release notes describe the difference between version 1.1 and version 1.0 of Mnesia. 1.1 is a minor release, but the storage format on disc has been changed. In order to use databases created with older versions of Mnesia, a full backup file must be created with the old version of Mnesia. The backup must be installed as fallback by the new version of Mnesia. Then the new version of Mnesia may be started.

## Improvements and new features

- Logging of "heavy weight transactions" has been optimized.

## Fixed Bugs and malfunctions

- Cleanup after deletion of SNMP-tables and deactivation of checkpoints was not sufficiently performed.  
Own Id: OTP-1367.

## Incompatibilities

As mentioned above, the storage format on disc has been changed.

## Known bugs and problems

See notes about release 1.0.

## Mnesia 1.0

This special version of Mnesia is intended to be used by the GPRS project only. It is released without any new documentation besides this release note. The release notes describes the difference between version 1.0 and version 0.80 of Mnesia. 1.0 is a major release and in order to use databases created with older versions of Mnesia, a full backup file must be created with the old version of Mnesia. The backup must be installed as fallback by the new version of Mnesia. Then the new version of Mnesia may be started.

## Improvements and new features

**Enhanced concept of schema and db\_nodes** The notion of db\_nodes has been extended. Now Mnesia is able to run on disc-less nodes as well as regular nodes that utilize the disc.

The schema table may, as other tables, reside on one or more nodes. The storage type of the schema table may either be `disc_copies` or `ram_copies` (not `disc_only_copies`). At startup Mnesia uses its schema to determine which other nodes that it should try to establish contact with. If any of the other nodes are already started, 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. The application parameter `extra_db_nodes` contains a list of nodes which Mnesia should also establish contact with (besides the ones found in the schema). The default value is the empty list `[]`.

The application parameter `schema_location` controls Mnesia to look for and locate schema. The parameter may be one of the following atoms:

`disc` Mandatory disc. The schema was assumed to be located on the Mnesia directory. And if the schema could not be found, Mnesia refused to start. This was the old behavior.

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

`opt_disc` Optional disc. The schema may 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`). When the `schema_location` is set to `opt_disc` the function `mnesia:change_table_copy_type/3` may be used to change the storage type of the schema. The default is `opt_disc`.

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 will affect two things.

First it allows other tables to be replicated to this node.

Secondly it will cause Mnesia to try to contact the node at startup of disc full nodes.

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

The schema table is not created with `mnesia:create_table/2` as normal tables. New schemas are created explicitly with `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. At startup when Mnesia connects to each other on different nodes, 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 will shutdown one of the nodes. This unfortunate situation will occur if a table has been created on two nodes independently of each other while they were disconnected. To solve this problem, one of the tables must be deleted (as the cookies differ, it is regarded as two different tables even if they happen to 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. But if the storage type of the schema is `ram_copies` its cookie can be replaced with a cookie from another node (`ram_copies` or `disc_copies`). Cookie replacement during a merge of the schema table definition is performed each time a RAM node connects to another node.

The functions `mnesia:add_db_node/1` and `mnesia:del_db_node/3` have been removed from the API. Adding and deleting `db_nodes` are performed as described above.

`mnesia:system_info(schema_location)` and `mnesia:system_info(extra_db_nodes)` may be used to determine the actual values of `schema_location` and `extra_db_nodes` respectively.

`mnesia:system_info(use_dir)` may be used to determine whether Mnesia is actually be using the Mnesia directory or not. `use_dir` may be determined even before Mnesia is started. The function `mnesia:info/0` may now be used to printout some system information even before Mnesia is started. When Mnesia is started the function prints out more information.

Transactions which update the definition of a table, require 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 may be created and new replicas may be added without all disc-full nodes being started. New replicas may be added without all other replicas of the table is being loaded, one other replica is sufficient.

The internal representation of the schema cookie, schema version and `db_nodes` has been changed. Their representation in backup files has also been changed. This affects the function `mnesia:traverse_backup/4,6` slightly. Now the definition of the schema table is represented in the same manner as the definition of other tables. In the backup this means a tuple `{schema, schema, TableDef}` instead of `{schema, cookie, Cookie}`, `{schema, version, Version}` and `{schema, db_nodes, DbNodes}`. Now all tables (including the schema table) have their own cookie and version. The `db_nodes` are found in the list of `ram_copies` and `disc_copies` nodes respectively in the tuple containing the definition of the schema table.

**New concept of handling Mnesia events** As Mnesia has evolved to conform to the application concept, the `mnesia_user` process has been replaced with a `gen_event` server.

In various situations Mnesia generates events. There are several categories of events. First, there are system events, which are important events that serious Mnesia applications should take an interest. The system events are currently:

`{mnesia_up, Node}` This means that Mnesia has been started on a node. `Node` is the name of the node. By default this event is ignored.

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

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

`{mnesia_checkpoint_deactivated, Checkpoint}` A checkpoint with the name `Checkpoint` has been deactivated and that the current node was involved in the checkpoint. Checkpoints may explicitly be deactivated with `mnesia:deactivate/1` or implicitly when the last replica of a table (involved in the checkpoint) becomes unavailable, e.g. at node down. By default this event is ignored.

`{mnesia_overload, Details}` Mnesia on the current node is overloaded and that the application ought to do something about it.

One example of a typical overload situation is when the application is performing more updates on disc resident tables than Mnesia is able to handle. Ignoring this kind of overload may lead into 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 it is dumped to the tables files occasionally depending on how it is configured. The table file storage is more compact than the transaction log storage, especially if the same record is updated many times. If the thresholds for dumping the transaction log has been reached before the previous dump was 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 are performing updates of disc resident tables. When this happens the message queue of the transaction manager will continue to grow until the memory is exhausted or the load decreases. The same problem may occur for dirty updates.

The overload is detected locally on the current node, but the cause may be on another node.

Application processes may cause heavy loads on other nodes if any of the tables are residing on other nodes (replicated or not). By default this event is reported to the `error_logger`.

`{mnesia_fatal, Format, Args, BinaryCore}` Mnesia has encountered a fatal error and will be terminated imminently. The reason for the fatal error is explained in `Format` and `Args` which may be given as input to `io:format/2` or sent to the `error_logger`. By default it is sent to the `error_logger`. `BinaryCore` is a binary containing a summary of Mnesia's internal state when the fatal error was encountered. By default the binary is written to a unique file name on current directory. On RAM nodes the core is ignored.

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

`{mnesia_error, Format, Args}` This means that Mnesia has encountered an error. The reason for the error is explained in `Format` and `Args` which may be given as input to `io:format/2` or sent to the `error_logger`. By default this event is reported to the `error_logger`.

`{mnesia_user, Event}` This means that some application has invoked the function `mnesia:report_event(Event)`. `Event` may be any Erlang data structure. When tracing a system

of Mnesia applications it is useful to be able to interleave Mnesia's own events with application related events that gives information about the application context. Whenever the application starts with some new demanding Mnesia activity or if it is entering a new interesting phase in its execution it may be a good idea to use `mnesia:report_event/1`.

Another category of events are table events, which are events related to table updates. The table events are tuples typically resembling: `{Oper, Record, TransId}`. Where `Oper` is the operation performed. `Record` is the record involved in the operation and `TransId` is the identity of the transaction performing the operation. The various table related events that may occur are:

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

`{delete_object, OldRecord, TransId}` 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. Note that, other records with the same key may be remaining in the table if it is a bag.

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

The function `mnesia:subscribe_config_change/0` has been replaced with the functions `mnesia:subscribe(EventCategory)` and `mnesia:unsubscribe(EventCategory)`. `EventCategory` may either be the atom `system` or the tuple `{table, Tab}`. The subscribe functions activate a subscription of events. The events are delivered as messages to the process evaluating the `mnesia:subscribe/1` function. The syntax of system events is `{mnesia_system_event, Event}` and `{mnesia_table_event, Event}` for table events. What system events and table events means is described above.

All system events are always subscribed by Mnesia's `gen_event` handler. The default `gen_event` handler is `mnesia_event`. But it may be changed with the application parameter `event_module`. The value of this parameter must be a 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)` may be used to determine which processes subscribe to various events.

**Enhanced debugging support** The new subscription mechanism enables building of powerful debugging and configuration tools (like the soon to be released `Xmnesia`).

`mnesia:debug/0` and `mnesia:verbose/0` has been replaced with `mnesia:set_debug_level(Level)`. `Level` is an atom which regulates the debugging level of Mnesia. The following debug levels are supported:

`none` No trace output at all. This is the default.

`verbose` Activates tracing of important debug events. These debug events will generate `{mnesia_info, Format, Args}` system events. Processes may subscribe on these events with `mnesia:subscribe/1`. The events are always sent to Mnesia's event handler.

`debug` Activates all events on the verbose level plus a full trace of all debug events. These debug events will generate `{mnesia_info, Format, Args}` system events. Processes may subscribe to these events with `mnesia:subscribe/1`. The events are always sent to Mnesia's event handler. On this debug level Mnesia's event handler starts subscribing updates to the schema table.

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

`false` is an alias for `none`.

`true` is an alias for `debug`.

**Enhanced error codes** Mnesia functions return `{error, Reason}` or `{aborted, Reason}` when it fails. This is still true but the `Reason` is now in many cases a tuple instead of a cryptic atom. The first field in the tuple tells what kind of error it is and the rest of the tuple contains details about the context or where the error occurred. For example, if a table does not exist `{no_exists, TableName}` is returned instead of just the atom `no_exists`. The function `mnesia:error_description/1` accepts the old atom style and the new tuple style. `{error, Reason}` and `{aborted, Reason}` tuples are also accepted.

**Conformance to the application concept** As Mnesia has evolved to conform with the OTP application concept, the process architecture of Mnesia has been restructured. For processes which not are performance critical `gen_server`, `gen_event` and `gen_fsm` are now used. Supervisors are used to supervise Mnesia's internal long living processes. The startup procedure now conforms with the supervisor concept. The side effect is poorer error codes at startup. `mnesia:start/0` will now return the cryptic tuple `{error, {shutdown, {mnesia_sup, start, [normal, []]}}}` when Mnesia startup fails. Use `-boot start_sasl` as argument to the `erl` script in order to get a little bit more information from start failures.

Mnesia now negotiates with Mnesia on other nodes at startup about which message protocol to use. This means that connecting a node with a future release of Mnesia with a node running this release of Mnesia will work fine, and not cause inconsistency because of protocol mismatches. Mnesia is also prepared for code change without stopping Mnesia. (The file representation on disc was already prepared for future format changes.)

**The transaction concept has been extended** A functional object (Fun) performing operations like:

- `mnesia:read/1`
- `mnesia:wread/1`
- `mnesia:write/1`
- `mnesia:s_write/1`
- `mnesia:delete/1`
- `mnesia:s_delete/1`
- `mnesia:delete_object/1`
- `mnesia:s_delete_object/1`
- `mnesia:all_keys/1`
- `mnesia:match_object/1`
- `mnesia:index_match_object/2`
- `mnesia:index_read/3`
- `mnesia:read_lock_table/1`
- `mnesia:write_lock_table/1`

may be sent to the function `mnesia:transaction/1,2,3` and will be performed in a transaction context involving mechanisms like locking, logging, replication, checkpoints, subscriptions, commit protocols etc. This is still true, but the same function may also be evaluated in other contexts.

By sending the same "fun" to the function `mnesia:async_dirty(Fun [, Args])` it will be performed in dirty context. The function calls will be mapped to the corresponding dirty functions. This will still involve logging, replication and subscriptions but there will be no locking, local transaction storage or

commit protocols involved. Checkpoint retainers will be updated but it will be updated dirty. As for normal `mnesia:dirty_*` operations the operations are performed semi asynchronous. The functions will wait for the operation to be performed on one node but not the others. If the table resides locally no waiting for other nodes is involved.

By sending the same “fun” to the function `mnesia:sync_dirty(Fun [, Args])` it will be performed in almost the same context as `mnesia:async_dirty/1,2`. The difference is that the operations are performed synchronously. The caller will wait for the updates to be performed on all active replicas. Using `sync_dirty` is useful for applications that are executing on several nodes and want to be sure that the update is performed on remote node before a remote process is spawned or a message is sent to a remote process. It may also be useful if the application perform so frequent or voluminous updates that Mnesia becomes overloaded on other nodes.

By sending the same “fun” to the function `mnesia:ets(Fun [, Args])` it will be performed in a very raw context. The operations will be performed directly on the local ets tables assuming that the local storage type is `ram_copies` and that the table is not replicated to other nodes. Subscriptions will not be triggered nor checkpoints updated, but it is blindingly fast.

All these activities (transaction, `async_dirty`, `sync_dirty` and `ets`) may be nested. Yes, we do support nested transactions! A nested activity is always automatically upgraded to be of the same kind as the outer one. For example, a “fun” executed in `async_dirty` inside a transaction will be executing inside transaction context.

Locks acquired by nested transactions will not be released until the outermost transaction has ended. The updates performed in a nested transaction will not be committed until the outermost transaction commits.

Mnemosyne queries may be performed in all these activity contexts (transaction, `async_dirty`, `sync_dirty` and `ets`). The `ets` activity will only work if the table has no indexes.

**An alternate commit protocol has been added** Both the new and old protocols are still used. Mnesia selects the most appropriate transaction commit protocol depending on which tables that has been involved in the transaction.

The old commit protocol is a very fast protocol, with a simple algorithm for recovery. But the drawback is that it only guarantees consistency after recovery for symmetrically replicated tables. If all tables involved in the transaction have the same replica pattern they are regarded as symmetric replicated. For example, if all tables involved in the transaction have a `ram_copies` replica on node A and a `disc_copies` replica on node B they are symmetrically replicated.

If the tables are asymmetrically replicated or if the schema table (containing the table definitions) is involved, the new heavy weight protocol is used to commit the transaction. The new protocol ensures consistency for all kinds of table updates. The protocol is able to recover the tables to a consistent state regardless of provocative applications or when a node crash occurs.

During commit, the new protocol will cause more network messages and disc accesses than the old protocol, but it is safer. At startup after a crash, there may exist transactions in the log that we do not know the outcome of. This should be very rare since the protocol has been deliberately designed to make the period of uncertainty as short as possible. When this rare situation occurs, Mnesia on recovering the node will not be able to start without asking Mnesia on other nodes for an outcome of the transaction in order to decide whether to commit or abort.

With this new approach several of the problems, described in earlier release notes have disappeared. See below:

- Now Mnesia is able to recover crashed schema transactions.
- Mnesia also guarantees consistency of asymmetrically replicated tables.

- The Mnesia database will remain consistent even if Mnesia is terminated under unfortunate circumstances.
- Mnesia now guarantees consistency on SNMP tables even if the application tries to store records that violate the type definition of the SNMP key.

**New startup procedure** When Mnesia starts on one node it may come to the conclusion that some of the tables can be loaded from local disc since no other nodes may hold a replica that is newer than the one on the starting node. The start function now returns without performing any loading of tables. The tables are loaded later in background allowing early availability of the transaction manager for those tables that have been loaded. The application will not have to wait for all tables to be loaded before it can start.

`mnesia:start/0` now returns the atom `ok` or `{error, Reason}`. In embedded systems this function is not used. In such systems `application:start(mnesia)` is used.

A new function `mnesia:start(Config)` is introduced. The `Config` argument is a list of `{Name, Val}` tuples, where `Name` is a name of an application parameter and `Val` is its value. The effect of the new values will remain until the application Mnesia is reloaded. After the transient change of application parameters, Mnesia will be started with `mnesia:start/0` and return its return value. For example `mnesia:start([extra_db_nodes, [svarte_bagge@switchboard]])` would override any old setting of `extra_db_nodes` until the application Mnesia was reloaded.

When an application is started it must synchronize with Mnesia to ensure that all the tables that it needs to access, have really been loaded before it attempts access. The function `mnesia:wait_for_tables(Tables, Timeout)` should be used for this purpose. *Note:* It is even more important to do this now since the start function will return earlier without loading any tables compared to previous releases. Do not forget this, otherwise the application will be less robust.

Each Mnesia table should be owned by *only* one module. This module is responsible for the life cycle of the table. When the application is installed for the first time in a network of nodes, this module must create the necessary tables. In each subsequent release of the application the module owning the table is responsible for performing changes of the table definition if required, (e.g. invoking `mnesia:transform/3` to transform the table).

When the application ceases to exist, it must be uninstalled from the network and the module that owns the table must delete its tables. It might also be a good idea to let this module export functions that allow customized interfaces to some of the Mnesia functions (eg. a special `wait_for_tables` that only waits for certain hard coded tables).

Other applications that need direct access to tables owned by a module in another application must declare a dependency to the other application in its `.app`-file in order to allow the code change, start and stop algorithms in the `application_controller` and `supervisor` modules to work.

**mnesia:dump\_tables/1** `mnesia:dump_tables/1` is now performed as a transaction and returns `{atomic, ok}` or `{aborted, Reason}` as the other functions performed in transaction context.

**mnesia:dirty\_update\_counter/2** `mnesia:dirty_update_counter(Counter, Incr)` now returns the new counter value instead of the atom `ok`.

### **New dirty functions have been introduced**

- `mnesia:dirty_all_keys/1`
- `mnesia:dirty_index_read/3`
- `mnesia:dirty_index_match/2`

They perform the same work as the corresponding function without 'dirty\_' prefix but in dirty context.

**Easier use of indexes** Now attribute names are also allowed to specify index positions. Index positions may still be given as field positions in the tuple corresponding to the record definitions.

The functions `mnesia:match_object/1` and `mnesia:dirty_match_object/1` automatically make use of indexes if any exist. No heuristics are performed in order to select the best index. Use Mnemosyne if this is an issue.

**Enhanced control of thresholds for dump of transaction log** The operations found in the transaction log will occasionally be performed on the actual dets tables. The frequency of the dumps is configurable with two application parameters. One is the `dump_log_time_threshold` which is an integer that specifies the dump log interval in milliseconds (it defaults to 3 minutes). If a dump has not been performed within `dump_log_time_threshold` milliseconds, a new dump will be performed regardless of how many writes have been performed.

The other is `dump_log_write_threshold` which is an integer specifying how many writes to the transaction log that is allowed, before a new dump of the log is to be performed. It defaults to 100 log writes.

Both thresholds may be configured independently of each other. When one of them is exceeded a dump will be performed.

As explained elsewhere, there may occur situations when the application performs updates at a pace faster than Mnesia is able to propagate the updates from the log to the table files. When this occurs, overload events are generated. If availability is important, subscribe to these events and perform a regulation of the update intensity for your application! If you ignore this, you may exhaust the disc space.

The old application parameter `-mnesia_dump_log_interval` has been replaced by the two operations mentioned above.

The function `mnesia:change_dump_log_config/1` has been removed from the API.

**Tables must have at least arity 3** Mnesia will not allow tables with arity 2. All tables must at least have one extra attribute besides the key. Now, an extra check is performed to disallow creation of tables with an arity of less than 3. In earlier releases the table creation succeeded but since Mnesia was not designed for such peculiar tables strange things happened and records were lost.

Check your schema for such tables if you load an old backup file.

**Non-blocking emulator** As mentioned in earlier release notes, match operations in ets tables will block the emulator for a long time if the tables are large. Now, a new BIF which performs partial matching of tables has been introduced. Mnesia is using the new BIF in match operations in order to avoid blocking the emulator with time consuming matches in large ets tables.

---

**Configuration parameters** As Mnesia has evolved and conformed to the application concept, the style of the configuration parameters have been changed. Mnesia is now configured by arguments to the erl script using the syntax stated by the application module in stdlib. Below is an example of parameters in the old style:

- `-mnesia_dir /my/favourite/dir`

Parameters should now resemble:

- `-mnesia_dir "/my/favourite/dir"`
- `-mnesia_extra_db_nodes '[kjetil@gprs, uffe@roof]'`

Following is a brief summary of the new configuration parameters:

- `debug ::= none | verbose | debug | trace`
- `schema_location ::= ram | disc | opt_disc`
- `extra_db_nodes ::= list of nodes`
- `event_module ::= name of gen_event handler module`
- `dump_log_write_threshold ::= number of log writes`
- `dump_log_time_threshold ::= number milliseconds`

**Fixed Bugs and malfunctions** The following

Own Identities; and,

Aux Identities have been solved:

- Own Id: OTP-1132.  
Aux Id: HA37457.
- Own Id: OTP-1164.
- Own Id: OTP-1182.  
Aux Id: HA40250.
- Own Id: OTP-1183 (OTP-1152).
- Own Id: OTP-1303.  
Aux Id: HA44214.

**Incompatibilities** See the chapter regarding improvements.

### Known bugs and problems

- The documentation describes the 0.80 version of Mnesia. Keep this in mind whenever you read something about startup algorithms, db\_nodes, transactions etc.
- Mnesia does not cope with network partitioning. If two running Mnesia nodes lose contact due to malfunctioning communication between the hosts anything may happen. The result is totally unpredictable.
- Mnesia does not detect if two nodes are sharing the same Mnesia directory. When two nodes are using the same Mnesia directory anything may happen. The result is totally unpredictable.
- `mnesia:restore/1` is not implemented.

---

## Mnesia 0.X

See the historical archives.

# MNESIA Reference Manual

## Short Summaries

- Erlang Module **Mnesia** [page 115] – A Distributed Telecommunications DBMS
- Erlang Module **mnesia\_registry** [page 143] – Dump support for registries in erl\_interface.

## Mnesia

The following functions are exported:

- `abort(Reason) -> transaction abort`  
[page 116] Aborts the current transaction.
- `activate_checkpoint(Args) -> {ok,Name,Nodes} | {error,Reason}`  
[page 116] Activates a checkpoint.
- `activity(AccessContext, Fun [, Args]) -> ResultOfFun | exit(Reason)`  
[page 117] Executes Fun in AccessContext.
- `activity(AccessContext, Fun, Args, AccessMod) -> ResultOfFun | exit(Reason)`  
[page 117] Executes Fun in AccessContext.
- `add_table_copy(Tab, Node, Type) -> {aborted, R} | {atomic, ok}`  
[page 118] Copies a table to a remote node.
- `add_table_index(Tab, AttrName) -> {aborted, R} | {atomic, ok}`  
[page 119] Creates an index for a table.
- `all_keys(Tab) -> KeyList | transaction abort`  
[page 119] Returns all keys in a table.
- `async_dirty(Fun, [, Args]) -> ResultOfFun | exit(Reason)`  
[page 119] Call the Fun in a context which is not protected by a transaction.
- `backup(Opaque [, BackupMod]) -> ok | {error,Reason}`  
[page 119] Backs up all tables in the database.
- `backup_checkpoint(Name, Opaque [, BackupMod]) -> ok | {error,Reason}`  
[page 119] Backs up all tables in a checkpoint.
- `change_table_access_mode(Tab, AccessMode) -> {aborted, R} | {atomic, ok}`  
[page 120] Changes the access mode for the table.

- `change_table_copy_type(Tab, Node, To) -> {aborted, R} | {atomic, ok}`  
[page 120] Changes the storage type of a table.
- `change_table_load_order(Tab, LoadOrder) -> {aborted, R} | {atomic, ok}`  
[page 120] Changes the load order priority for the table.
- `create_schema(DiscNodes) -> ok | {error, Reason}`  
[page 120] Creates a brand new schema on the specified nodes.
- `create_table(Name, TabDef) -> {atomic, ok} | {aborted, Reason}`  
[page 120] Creates a Mnesia table called Name with properties as described by the argument TabDef.
- `deactivate_checkpoint(Name) -> ok | {error, Reason}`  
[page 122] Deactivates a checkpoint.
- `del_table_copy(Tab, Node) -> {aborted, R} | {atomic, ok}`  
[page 122] Deletes the replica of table Tab at node Node.
- `del_table_index(Tab, AttrName) -> {aborted, R} | {atomic, ok}`  
[page 122] Deletes an index in a table.
- `delete({Tab, Key}) -> transaction abort | ok`  
[page 122] Deletes all records in table Tab with the key Key.
- `delete(Tab, Key, LockKind) -> transaction abort | ok`  
[page 122] Deletes all records in table Tab with the key Key.
- `delete_object(Record) -> transaction abort | ok`  
[page 122] Deletes a record
- `delete_object(Tab, Record, LockKind) -> transaction abort | ok`  
[page 122] Deletes a record
- `delete_schema(DiscNodes) -> ok | {error, Reason}`  
[page 123] Deletes the schema on the given nodes
- `delete_table(Tab) -> {aborted, Reason} | {atomic, ok}`  
[page 123] Permanently deletes all replicas of table Tab.
- `dirty_all_keys(Tab) -> KeyList | exit({aborted, Reason}).`  
[page 123] Dirty search for all record keys in table.
- `dirty_delete({Tab, Key}) -> ok | exit({aborted, Reason})`  
[page 123] Dirty delete of a record.
- `dirty_delete(Tab, Key) -> ok | exit({aborted, Reason})`  
[page 123] Dirty delete of a record.
- `dirty_delete_object(Record)`  
[page 123] Dirty delete of a record.
- `dirty_delete_object(Tab, Record)`  
[page 123] Dirty delete of a record.
- `dirty_first(Tab) -> Key | exit({aborted, Reason})`  
[page 123] Returns the key for the first record in a table.
- `dirty_index_match_object(Pattern, Pos)`  
[page 124] Dirty pattern match using index.
- `dirty_index_match_object(Tab, Pattern, Pos)`  
[page 124] Dirty pattern match using index.
- `dirty_index_read(Tab, SecondaryKey, Pos)`  
[page 124] Dirty read using index.

- `dirty_match_object(Pattern) -> RecordList | exit({aborted, Reason})`.  
[page 124] Dirty pattern match pattern.
- `dirty_match_object(Tab, Pattern) -> RecordList | exit({aborted, Reason})`.  
[page 124] Dirty pattern match pattern.
- `dirty_next(Tab, Key) -> Key | exit({aborted, Reason})`  
[page 124] Returns the next key in a table.
- `dirty_read({Tab, Key}) -> ValueList | exit({aborted, Reason})`  
[page 124] Dirty read of records.
- `dirty_read(Tab, Key) -> ValueList | exit({aborted, Reason})`  
[page 124] Dirty read of records.
- `dirty_slot(Tab, Slot) -> RecordList | exit({aborted, Reason})`  
[page 124] Returns the list of records that are associated with Slot in a table.
- `dirty_update_counter({Tab, Key}, Incr) -> NewVal | exit({aborted, Reason})`  
[page 124] Dirty update of a counter record.
- `dirty_update_counter(Tab, Key, Incr) -> NewVal | exit({aborted, Reason})`  
[page 124] Dirty update of a counter record.
- `dirty_write(Record) -> ok | exit({aborted, Reason})`  
[page 125] Dirty write of a record.
- `dirty_write(Tab, Record) -> ok | exit({aborted, Reason})`  
[page 125] Dirty write of a record.
- `dump_log() -> dumped`  
[page 125] Performs a user initiated dump of the local log file.
- `dump_tables(TabList) -> {atomic, ok} | {aborted, Reason}`  
[page 125] Dumps all RAM tables to disc.
- `dump_to_textfile(Filename)`  
[page 125] Dumps local tables into a text file.
- `error_description(Error) -> String`  
[page 125] Returns a string describing a particular Mnesia error.
- `ets(Fun, [, Args]) -> ResultOfFun | exit(Reason)`  
[page 126] Calls the Fun in a raw context which is not protected by a transaction.
- `force_load_table(Tab) -> yes | ErrorDescription`  
[page 126] Forces a table to be loaded into the system
- `index_match_object(Pattern, Pos) -> transaction abort | ObjList`  
[page 126] Matches records and utilizes index information.
- `index_match_object(Tab, Pattern, Pos, LockKind) -> transaction abort | ObjList`  
[page 126] Matches records and utilizes index information.
- `index_read(Tab, SecondaryKey, Pos) -> transaction abort | RecordList`  
[page 127] Reads records via index table.
- `info() -> ok`  
[page 127] Prints some information about the system on the tty.
- `install_fallback(Opaque) -> ok | {error, Reason}`  
[page 127] Installs a backup as fallback.

- `install_fallback(Opaque, BackupMod) -> ok | {error, Reason}`  
[page 127] Installs a backup as fallback.
- `install_fallback(Opaque, Args) -> ok | {error, Reason}`  
[page 127] Installs a backup as fallback.
- `load_textfile(Filename)`  
[page 128] Loads tables from a text file.
- `lock(LockItem, LockKind) -> GoodNodes | transaction abort`  
[page 128]
- `match_object(Pattern) -> transaction abort | RecList`  
[page 129] Match Pattern for records.
- `match_object(Tab, Pattern, LockKind) -> transaction abort | RecList`  
[page 129] Match Pattern for records.
- `move_table_copy(Tab, From, To) -> {aborted, Reason} | {atomic, ok}`  
[page 129] Moves the copy of table Tab from node From to node To.
- `read({Tab, Key}) -> transaction abort | RecordList`  
[page 130] Reads records(s) with a given key.
- `read(Tab, Key, LockKind) -> transaction abort | RecordList`  
[page 130] Reads records(s) with a given key.
- `read_lock_table(Tab) -> ok | transaction abort`  
[page 130] Sets a read lock on an entire table.
- `report_event(Event) -> ok`  
[page 130] Reports a user event to Mnesia's event handler.
- `restore(Opaque, Args) -> {atomic, RestoredTabs} | {aborted, Reason}`  
[page 130] Online restore of backup.
- `s_delete({Tab, Key}) -> ok | transaction abort`  
[page 131] Sets sticky lock and delete records.
- `s_delete_object(Record) -> ok | transaction abort`  
[page 131] Sets sticky lock and delete record.
- `s_write(Record) -> ok | transaction abort`  
[page 131] Writes Record and sets stick lock.
- `schema() -> ok`  
[page 131] Prints information about all table definitions on the tty.
- `schema(Tab) -> ok`  
[page 131] Prints information about one table definition on the tty.
- `set_debug_level(Level) -> OldLevel`  
[page 131] Changes the internal debug level of Mnesia
- `set_master_nodes(MasterNodes) -> ok | {error, Reason}`  
[page 131] Sets the master nodes for all tables
- `set_master_nodes(Tab, MasterNodes) -> ok | {error, Reason}`  
[page 132] Sets the master nodes for a table
- `snmp_close_table(Tab) -> {aborted, R} | {atomic, ok}`  
[page 132] Removes the possibility for SNMP to manipulate the table.
- `snmp_get_mnesia_key(Tab, RowIndex) -> {ok, Key} | undefined`  
[page 132] Gets the corresponding Mnesia key from an SNMP index.
- `snmp_get_next_index(Tab, RowIndex) -> {ok, NextIndex} | endOfTable`  
[page 132] Gets the index of the next lexicographical row.

- `snmp_get_row(Tab, RowIndex) -> {ok, Row} | undefined`  
[page 132] Retrieves a row indexed by an SNMP index.
- `snmp_open_table(Tab, SnmpStruct) -> {aborted, R} | {atomic, ok}`  
[page 132] Organizes a Mnesia table as an SNMP table.
- `start() -> ok | {error, Reason}`  
[page 133] Starts a local Mnesia system.
- `stop() -> stopped`  
[page 134] Stops Mnesia locally.
- `subscribe(EventCategory)`  
[page 134] Subscribes to events of type `EventCategory`.
- `sync_dirty(Fun, [, Args]) -> ResultOfFun | exit(Reason)`  
[page 134] Calls the `Fun` in a context which is not protected by a transaction.
- `system_info(InfoKey) -> Info | exit({aborted, Reason})`  
[page 134] Returns information about the Mnesia system
- `table_info(Tab, InfoKey) -> Info | exit({aborted, Reason})`  
[page 136] Returns local information about table.
- `transaction(Fun [[, Args], Retries]) -> {aborted, Reason} | {atomic, ResultOfFun}`  
[page 137] Executes a transaction.
- `transform_table(Tab, Fun, NewAttributeList) -> {aborted, R} | {atomic, ok}`  
[page 138] Changes format on all records in table. `Tab`
- `traverse_backup(Source, [SourceMod,] Target, [TargetMod,] Fun, Acc) -> {ok, LastAcc} | {error, Reason}`  
[page 139] Traversal of a backup.
- `uninstall_fallback() -> ok | {error, Reason}`  
[page 139] Uninstall a fallback.
- `uninstall_fallback(Args) -> ok | {error, Reason}`  
[page 139] Uninstall a fallback.
- `unsubscribe(EventCategory)`  
[page 139] Subscribes to events of type `EventCategory`.
- `wait_for_tables(TabList, Timeout) -> ok | {timeout, BadTabList}`  
[page 139] Wait for tables to be accessible.
- `wread({Tab, Key}) -> transaction abort | RecordList`  
[page 139] Reads records with given key.
- `write(Record) -> transaction abort | ok`  
[page 140] Writes a record into the database.
- `write(Tab, Record, LockKind) -> transaction abort | ok`  
[page 140] Writes an record into the database.
- `write_lock_table(Tab) -> ok | transaction abort`  
[page 140] Sets write lock on an entire table.

## mnesia\_registry

The following functions are exported:

- `create_table(Tab) -> ok | exit(Reason)`  
[page 143] Creates a registry table in Mnesia.
- `create_table(Tab, TabDef) -> ok | exit(Reason)`  
[page 143] Creates a customized registry table in Mnesia.

# mnesia (Module)

Mnesia 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.

Listed below are some of the most important and attractive capabilities, Mnesia provides:

- A relational/object hybrid data model which is suitable for telecommunications applications.
- A specifically designed DBMS query language, Mnemosyne (as an add-on application).
- Persistence. Tables may be coherently kept on disc as well as in main memory.
- Replication. Tables may 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 location of data.
- Extremely fast real time data searches.
- Schema manipulation routines. It is possible to reconfigure the DBMS at runtime without stopping the system.

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

All functions documented in these pages can be used in any combination with queries using the list comprehension notation. The query notation is described in the Mnemosyne User's Guide.

Data in Mnesia is organized as a set of tables. Each table has a name which 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. Below are some of the properties that are associated with each table:

- `type`. Each table can either have 'set' or 'bag' semantics. If a table is of type 'set' it means that 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. On the other hand, if a table is of type 'bag', each key can map to several records. However, all records in type bag tables are unique, only the keys may be duplicated.
- `record_name`. All records stored in a table must have the same name. You may say that the records must be instances of the same record type.
- `ram_copies`. A table can be replicated on a number of Erlang nodes. The `ram_copies` property 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` The `disc_copies` property specifies a list of Erlang nodes where the table is kept in RAM as well as on disc. All updates of the table are performed on the actual table and are also logged to disc. If a table is of type `disc_copies` at a certain node, it means that the entire table is resident in RAM memory as well as on disc. Each transaction performed on the table is appended to a LOG file as well as 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 shall build and maintain an extra index table.
- `local_content` When an application requires tables whose contents is local to each node, `local_content` tables may be used. The name of the table is known to all Mnesia nodes, but its contents is unique on each node. This means that access to such a table must be done locally. Set the `local_content` field to `true` if you want to enable the `local_content` behavior. The default is `false`.
- `snmp` Each (set based) Mnesia table can be automatically turned into an 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.

See `mnesia:create_table/2` about the complete set of table properties and their details.

This document 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 attribute of the record is the primary key, or key for short.

The function descriptions are sorted in alphabetic order. *Hint*: start to read about `mnesia:create_table/2`, `mnesia:lock/2` and `mnesia:activity/4` before you continue on and learn about the rest.

## Exports

```
abort(Reason) -> transaction abort
```

Makes the transaction silently return the tuple `{aborted, Reason}`. The abortion of a Mnesia transaction means that an exception will be thrown to an enclosing `catch`. Thus, the expression `catch mnesia:abort(x)` does not abort 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 will present 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 of checkpoint. Each checkpoint must have a name which is unique to the associated nodes. The name can be reused only once the checkpoint has been deactivated. By default, a name which is probably unique is generated.
- `{max, MaxTabs}` MaxTabs is a list of tables that should be included in the checkpoint. The default is `[]`. For these tables, the redundancy will be maximized and checkpoint information will be retained together with all replicas. The checkpoint becomes more fault tolerant if the tables have several replicas. When a new replica is added by means of the schema manipulation function `mnesia:add_table_copy/3`, a retainer will also be attached automatically.
- `{min, MinTabs}`. MinTabs is a list of tables that should be included in the checkpoint. The default is `[]`. For these tables, the redundancy will be minimized and the checkpoint information will only be 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 set to `true`.
- `{ram_overrides_dump, Bool}` Only applicable for `ram_copies`. `Bool` allows you to choose to backup the table state as it is in RAM, or as it is on disc. `true` means that the latest committed records in RAM should be included in the checkpoint. These are the records that the application accesses. `false` means that the records dumped to DAT files should be included in the checkpoint. These are the records that will be loaded at startup. Default is `false`.

Returns `{ok, Name, Nodes}` or `{error, Reason}`. Name is the (possibly generated) name of the checkpoint. 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)
```

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

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

This function executes the functional object `Fun` with the arguments `Args`.

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

`transaction` Short for `{transaction, infinity}`

`{transaction, Retries}` Invokes `mnesia:transaction(Fun, Args, Retries)`.

Note that the result from the `Fun` is returned if the transaction was successful (atomic), otherwise the function exits with an abort reason.

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

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

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

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

Mnesia will forward calls to the following functions:

- `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:index_match_object/4` (`index_match_object/2`)
- `mnesia:index_read/3`
- `mnesia:lock/2` (`read_lock_table/1`, `write_lock_table/1`)
- `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: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)`

where `ActivityId` is a record which represents the identity of the enclosing Mnesia activity. The first field (obtained with `element(1, ActivityId)`) contains an atom which may be interpreted as the type of the activity: `'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 which is internal to Mnesia.

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

This function makes another copy of a table at the node `Node`. The `Type` argument 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 indices can and should be used whenever the user wants to frequently use some other field than the key field to look up records. If this other field has an index associated with it, these lookups can occur in constant time and space. For example, if our application wishes to use the age field of persons to efficiently find all person with a specific age, it might be a good idea to have an index on the age field. This can be accomplished with the following call:

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

Indices do not come free, they occupy space which is proportional to the size of the table. They also cause insertions into the table to execute slightly slower.

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

This function returns a list of all keys in the table named `Tab`. The semantics of this function is context sensitive. See `mnesia:activity/4` for more information. In transaction context it acquires a read lock on the entire table.

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

Call the `Fun` in a context which 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 indices are updated, but they will be updated dirty. As for normal `mnesia:dirty_*` operations, the operations are performed semi-asynchronously. See `mnesia:activity/4` and the Mnesia User's Guide for more details.

It is possible to manipulate the Mnesia tables without using transactions. This has some serious disadvantages, but is considerably faster since the transaction manager is not involved and no locks are set. A dirty operation does, however, guarantee a certain level of consistency and it is not possible for the dirty operations to 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 in order to function.

*Note:* It is more than 10 times more efficient to read records dirty than within a transaction.

Depending on the application, it may be a good idea to use the dirty functions for certain operations. Almost all Mnesia functions which can be called within transactions have a dirty equivalent which is much more efficient. However, it must be noted that it is possible for the database to be left in an inconsistent state if dirty operations are used to update it. Dirty operations should only be used for performance reasons when it is absolutely necessary.

```
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 the backup module `BackupMod`. Tables with the `local_contents` property is being backed up as they exist on the current node. `BackupMod` is the default backup callback module obtained by `mnesia:system_info(backup_module)`. See the User's Guide about the exact callback interface (the `mnesia_backup` behavior).

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

The `AccessMode` is by default the `atom_read_write` but it may also be set to the `atom_read_only`. If the `AccessMode` is set to `read_only`, it means that it is not possible to perform updates to the table. At startup Mnesia always loads `read_only` tables locally regardless of when and if Mnesia was 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 our `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 may be set to any integer. The tables with the highest `LoadOrder` priority will be loaded first at startup.

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

Creates a new database on disc. Various files are created in the local Mnesia directory of each node. Note that the directory must be unique for each node. Two nodes may never share the same directory. If possible, use a local disc device in order 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 anyone of the nodes, or if anyone of the nodes already has a schema. Use `mnesia:delete_schema/1` to get rid of old faulty schemas.

*Note:* Only nodes with disc should be included in `DiscNodes`. Disc-less nodes, that is nodes where all tables including the schema only resides in RAM, may not be included.

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

This function creates a Mnesia table called `Name` according to the 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 may also be set to the `atom_read_only`. If the `AccessMode` is set to `read_only`, it means that it is not possible to perform updates to the table.

At startup Mnesia always loads `read_only` tables locally regardless of when and if Mnesia was terminated on other nodes. This argument returns the access mode of the table. The access mode may either be `read_only` or `read_write`.

- `{attributes, AtomList}` a list of the attribute names for the records that are supposed to populate the table. The default value is `[key, val]`. The table must have at least 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 the 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 as well as 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. The default value 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 will 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 shall build and maintain an extra index table. The Mnemosyne query compiler may or may not utilize any additional indices while processing queries on a table.
- `{load_order, Integer}`. The load order priority is by default 0 (zero) but may be set to any integer. The tables with the highest load order priority will be loaded first at startup.
- `{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 obviously not written to disc on a per transaction basis. It is possible to dump `ram_copies` replicas to disc with the function `mnesia:dump_tables(Tabs)`. The 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 name of the table.
- `{snmp, SnmpStruct}`. See `mnesia:snmp_open_table/2` for a description of `SnmpStruct`. If this attribute is present in the `ArgList` to `mnesia:create_table/2`, the table is immediately accessible by means of the Simple Network Management Protocol (SNMP). This means that applications which 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 which makes up a Mnesia table.
- `{type, Type}`, where `Type` must be either of the atoms `set` or `bag`. The default value is `set`. In a `set` all records have unique keys and in a `bag` several records may have the same key, but the record content is unique. If a non-unique record is stored the old, conflicting record(s) will simply be overwritten.

For example, the following call creates the person table previously defined and replicates it on 2 nodes:

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

If it was required that Mnesia build and maintain an extra index table on the address attribute 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` may be hard coded as `{index, [2]}` and `{attributes, [name, age, address, salary, children]}` respectively.

`mnesia:create_table/2` writes records into the schema table. This function, as well as 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 may happen when nodes go down or when a replica is deleted. Checkpoints will also be 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 may also be used to delete a replica of the table named `schema`. Then the node will be removed from `db_nodes`.

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

This function deletes the index on attribute with name `AttrName` in a table.

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

Invokes `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. See `mnesia:activity/4` for more information. 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
```

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

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

In a table is of type bag, we may sometimes want to delete only some of the records with a certain key. This can be done with the `delete_object/2` function. A complete record must be supplied to this function.

The semantics of this function is context sensitive. See `mnesia:activity/4` for more information. 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` is not alive, or if Mnesia is running on any of the nodes.

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

**Warning:**

This function must be used with extreme caution since 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})`.

This is the dirty equivalent of the `mnesia:all_keys/1` function.

`dirty_delete({Tab, Key}) -> ok | exit({aborted, Reason})`

Invokes `mnesia:dirty_delete(Tab, Key)`.

`dirty_delete(Tab, Key) -> ok | exit({aborted, Reason})`

This is the dirty equivalent of the `mnesia:delete/3` function.

`dirty_delete_object(Record)`

Invokes `mnesia:dirty_delete_object(Tab, Record)` where `Tab` is `element(1, Record)`.

`dirty_delete_object(Tab, Record)`

This is the dirty equivalent of the `mnesia:delete_object/3` function.

`dirty_first(Tab) -> Key | exit({aborted, Reason})`

Records in Mnesia tables are not ordered. However, there is an ordering of the records which is not known to the user. Accordingly, it is possible to traverse a table by means of this function in conjunction with the `mnesia:dirty_next/2` function.

If there are no records at all in the table, this function returns the atom `'$end_of_table'`. For this reason, it is highly undesirable, but not disallowed, to use this atom as the key for any user records.

`dirty_index_match_object(Pattern, Pos)`

Invokes `mnesia:dirty_index_match_object(Tab, Pattern, Pos)` where `Tab` is `element(1, Pattern)`.

`dirty_index_match_object(Tab, Pattern, Pos)`

This is the dirty equivalent of the `mnesia:index_match_object/4` function.

`dirty_index_read(Tab, SecondaryKey, Pos)`

This is the dirty equivalent of the `mnesia:index_read/3` function.

`dirty_match_object(Pattern) -> RecordList | exit({aborted, Reason})`.

Invokes `mnesia:dirty_match_object(Tab, Pattern)` where `Tab` is `element(1, Pattern)`.

`dirty_match_object(Tab, Pattern) -> RecordList | exit({aborted, Reason})`.

This is the dirty equivalent of the `mnesia:match_object/3` function.

`dirty_next(Tab, Key) -> Key | exit({aborted, Reason})`

This function makes it possible to traverse a table and perform 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 which 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 `mnesia:dirty_next/2` function.

`dirty_read({Tab, Key}) -> ValueList | exit({aborted, Reason})`

Invokes `mnesia:dirty_read(Tab, Key)`.

`dirty_read(Tab, Key) -> ValueList | exit({aborted, Reason})`

This is the dirty equivalent of the `mnesia:read/3` function.

`dirty_slot(Tab, Slot) -> RecordList | exit({aborted, Reason})`

This function can be used to traverse a table in a manner similar to the `mnesia:dirty_next/2` function. A table has a number of slots which range from 0 (zero) to some 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})`

Invokes `mnesia:dirty_update_counter(Tab, Key, Incr)`.

`dirty_update_counter(Tab, Key, Incr) -> NewVal | exit({aborted, Reason})`

There are no special counter records in Mnesia. 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 despite the fact that it is not protected by a transaction.

If two processes perform `mnesia:dirty_update_counter/3` simultaneously, both updates will take effect without the risk of losing one of the updates. The new value `NewVal` of the counter is returned.

`dirty_write(Record) -> ok | exit({aborted, Reason})`

Invokes `mnesia:dirty_write(Tab, Record)` where `Tab` is `element(1, Record)`.

`dirty_write(Tab, Record) -> ok | exit({aborted, Reason})`

This is the dirty equivalent of `mnesia:write/3`.

`dump_log() -> dumped`

Performs a user initiated dump of the local log file. This is usually not necessary since Mnesia, by default, manages this automatically.

`dump_tables(TabList) -> {atomic, ok} | {aborted, Reason}`

This function 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 may have disc resident replicas.

`dump_to_textfile(Filename)`

Dumps all local tables of a mnesia system into a text file which can then be edited (by means of a normal text editor) and then later 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 the value `{atomic, Val}` or the tuple `{aborted, Reason}`. The `Reason` can be either of the following atoms. The `error_description/1` function returns a descriptive string which 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 were illegally combined.
- `bad_index`. Index already exists or was out of bounds.
- `already_exists`. Schema option is already set.
- `index_exists`. Some operations can not be performed on tabs with index.
- `no_exists`. Tried to perform operation on non-existing, or not alive, item.
- `system_limit`. Some system\_limit was exhausted.
- `mnesia_down`. A transaction involving records at some remote node which died while transaction was executing. Record(s) are no longer available elsewhere in the network.
- `not_a_db_node`. A node which does not exist in the schema was mentioned.
- `bad_type`. Bad type on some arguments.
- `node_not_running`. Node 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 record.

The Error may be Reason, {error, Reason}, or {aborted, Reason}. The Reason may be an atom or a tuple with Reason as an atom in the first field.

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

Call the Fun in a raw context which is not protected by a transaction. The Mnesia function call is performed in the Fun are 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. See `mnesia:activity/4` and the Mnesia User's Guide for more details.

```
force_load_table(Tab) -> yes | ErrorDescription
```

The Mnesia algorithm for table load might 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 due to 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 could lead to a situation where some transaction effects are lost with a inconsistent database as result, but for some applications high availability is more important than consistent data.

```
index_match_object(Pattern, Pos) -> transaction abort | ObjList
```

Invokes `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 `mnesia:index_read/3` function, we can also utilize any index information when we try to match records. This function takes a pattern which obeys the same rules as the `mnesia:match_object/3` function with the exception 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` may either be an integer (`#record.Field`), or an attribute name.

The two index search functions described here are automatically invoked when searching tables with Mnemosyne list comprehensions and also when using the low level `mnesia:[dirty_]match_object` functions.

*Note:* Mnemosyne has a “search” advantage as it uses some clever heuristics in order to select the best index.

The semantics of this function is context sensitive. See `mnesia:activity/4` for more information. 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 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 the `person` table, the call `mnesia:index_read(person, 36, #person.age)` returns a list of all persons with age equal to 36. `Pos` may also be an attribute name (`atom`), but if the notation `mnesia:index_read(person, 36, age)` is used, the field position will be searched for in runtime, for each call.

The semantics of this function is context sensitive. See `mnesia:activity/4` for more information. In transaction context it acquires a read lock on the entire table.

```
info() -> ok
```

Prints some information about the system on the `tty`. This function may be used even if Mnesia is not started. However, more information will be displayed if Mnesia is started.

```
install_fallback(Opaque) -> ok | {error,Reason}
```

Invokes `mnesia:install_fallback(Opaque, Args)` where `Args` is `[{scope, global}]`.

```
install_fallback(Opaque, BackupMod) -> ok | {error,Reason}
```

Invokes `mnesia:install_fallback(Opaque, Args)` where `Args` is `[{scope, global}, {module, BackupMod}]`.

```
install_fallback(Opaque, Args) -> ok | {error,Reason}
```

This function is used to install a backup as fallback. The fallback will be used to restore the database at the next start-up. Installation of fallbacks requires Erlang to be up and running on all the involved nodes, but it does not matter if Mnesia is running or not. The installation of the fallback will fail 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 is performed via a callback module named `BackupMod`. The `Opaque` argument is forwarded to the callback module which may interpret it as it wish. The default callback module is called `mnesia_backup` and it interprets the `Opaque` argument as a local filename. The default for this module is also configurable via the `-mnesia mnesia_backup` configuration parameter.
- `{scope, Scope}` The `Scope` of a fallback may either be `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 all nodes with disc resident schema or none. Which nodes that are disc resident or not, is determined from the schema info in the backup.  
If the `Scope` of the operation is `local` the fallback will only be 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 towards the `Mnesia` directory as configured with the `-mnesia dir` configuration parameter. But by explicitly supplying an `AlternateDir` the fallback will be installed there regardless of the `Mnesia` directory configuration parameter setting. After installation of a fallback on an alternate `Mnesia` directory that directory is fully prepared for usage as an active `Mnesia` directory.  
This is a somewhat dangerous feature which must be used with care. By unintentional mixing of directories you may easily end up with a inconsistent database, if the same backup is installed on more than one directory.

`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 and using other functions to deal with real backups, is recommended.

`lock(LockItem, LockKind) -> GoodNodes | 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 invoked in a transaction context. The granularity of a lock may either be a single record or an entire table.

This function `mnesia:lock/2` is intended to support explicit locking on tables but also intended for situations when locks need to be acquired regardless of how tables are replicated. Currently, two `LockKind`'s are supported:

`write` Write locks are exclusive, which means that if one transaction manages to acquire a write lock on an item, no other transaction may acquire any kind of lock on the same item.

`read` Read locks may be shared, which means that if one transaction manages to acquire a read lock on an item, other transactions may 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 some one has a write lock no one can acquire a read lock nor 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 aborted and executed again. Mnesia does this automatically as long as the upper limit of maximum retries is not reached. See `mnesia:transaction/3` for the details.

For the sake of completeness sticky write locks will also be described here even if a sticky write lock is not supported by this particular function:

`stick_write` Sticky write locks are a mechanism which 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 may be the best option available.

When a sticky write lock is acquired, all nodes will be informed which node is locked. Subsequently, sticky lock requests from the same node will be performed as a local operation without any communication with other nodes. The sticky lock lingers on the node even after the transaction has ended. See the Mnesia User's Guide for more information.

Currently, two kinds of `LockItem`'s are supported by this function:

`{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. See `mnesia:activity/4` for more information. In transaction context it acquires locks otherwise it just ignores the request.

```
match_object(Pattern) ->transaction abort | RecList
```

Invokes `mnesia:match_object(Tab, Pattern, read)` where `Tab` is `element(1, Pattern)`.

```
match_object(Tab, Pattern, LockKind) ->transaction abort | RecList
```

This function takes a pattern with 'don't care' variables denoted as a '\_' parameter. This function returns a list of records which 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 thirty-six (36).

The function `mnesia:match_object/3` automatically uses indices if these exist. However, no heuristics are performed in order to select the best index. Use Mnemosyne if this is an issue.

The semantics of this function is context sensitive. See `mnesia:activity/4` for more information. 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. It is still possible for other transactions to read and write in the table while it is being moved.

This function cannot be used on `local_content` tables.

```
read({Tab, Key}) -> transaction abort | RecordList
```

Invokes `mnesia:read(Tab, Key, read)`.

```
read(Tab, Key, LockKind) -> transaction abort | RecordList
```

This function 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 `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. See `mnesia:activity/4` for more information. In transaction context it acquires a lock of type `LockKind`. Currently, the lock types `read` and `write` are supported.

If the user wants to update the record it is more efficient to use `write` as the `LockKind`.

```
read_lock_table(Tab) -> ok | transaction abort
```

Invokes `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's 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 is entering a new interesting phase in its execution, it may be a good idea to use `mnesia:report_event/1`. The `Event` may 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 may 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` will be used to access the backup media. If omitted, the default backup module will be used.
- `{skip_tables, TabList}` Where `TabList` is a list of tables which should not be read from the backup.
- `{clear_tables, TabList}` Where `TabList` is a list of tables which should be cleared, before the records from the backup are inserted, ie. all records in the tables are deleted before the tables are restored. Schema information about the tables is not cleared or read from backup.

- `{keep_tables, TabList}` Where `TabList` is a list of tables which should be not be cleared, before the records from the backup are inserted, i.e. the records in the backup will be added to the records in the table. Schema information about the tables is not cleared or read from backup.
- `{recreate_tables, TabList}` Where `TabList` is a list of tables which should be re-created, 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 needs to be up and running.
- `{default_op, Operation}` Where `Operation` is one of the following operations `skip_tables`, `clear_tables`, `keep_tables` or `recreate_tables`. The default operation specifies which operation should be used on tables from the backup which are not specified in any of the lists above. If omitted, the operation `clear_tables` will be used.

The affected tables are write locked during the restoration, but 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 may not be possible to restore it online. In such cases, the old database must be restored by installing a fallback and then restart.

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

Invokes `mnesia:delete(Tab, Key, sticky_write)`

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

Invokes `mnesia:delete_object(Tab, Record, sticky_write)` where `Tab` is `element(1, Record)`.

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

Invokes `mnesia:write(Tab, Record, sticky_write)` where `Tab` is `element(1, Record)`.

`schema() -> ok`

Prints information about all table definitions on the tty.

`schema(Tab) -> ok`

Prints information about one table definition on the tty.

`set_debug_level(Level) -> OldLevel`

Changes the internal debug level of Mnesia. See the chapter about configuration parameters for details.

`set_master_nodes(MasterNodes) -> ok | {error, Reason}`

For each table Mnesia will determine its replica nodes (`TabNodes`) and invoke `mnesia:set_master_nodes(Tab, TabMasterNodes)` where `TabMasterNodes` is the intersection of `MasterNodes` and `TabNodes`. See `mnesia:set_master_nodes/2` about the semantics.

```
set_master_nodes(Tab, MasterNodes) -> ok | {error, Reason}
```

If the application detects that there has been a communication failure (in a potentially partitioned network) which may have caused an inconsistent database, it may use the function `mnesia:set_master_nodes(Tab, MasterNodes)` to define from which nodes each table will be loaded. At startup Mnesia's normal table load algorithm will be bypassed and the table will be loaded from one of the master nodes defined for the table, regardless of when and if Mnesia was terminated on other nodes. The `MasterNodes` may only contain nodes where the table has a replica and if the `MasterNodes` list is empty, the master node recovery mechanism for the particular table will be reset and the normal load mechanism will be used at next restart.

The master node setting is always local and it may be changed regardless of whether Mnesia is started or not.

The database may also become inconsistent if the `max_wait_for_decision` configuration parameter 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()]`

The `RowIndex` may specify a non-existing row. Specifically, it might be the empty list. Returns the index of the next lexicographical row. If `RowIndex` is the empty list, this function will return 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)`

Makes it possible to read 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`

It is possible to establish a direct one to one mapping 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.

The `SnmpStruct` argument is a list of SNMP information. Currently, the only information needed is information about the key types in the table. It is not possible to handle multiple keys 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 fixed size. For example:

```
mnesia:snmp_open_table(person, [{key, string}])
```

causes the `person` table to be ordered as an SNMP table.

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`.

It is possible 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 the `telno` column 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,  $O(\log N)$ . And more memory is used.

*Note:* Only the lexicographical SNMP ordering is implemented in Mnesia, not the actual SNMP monitoring.

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

The start-up 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 will be referred to as the Mnesia directory. Mnesia may also be started on disc-less nodes. See `mnesia:create_schema/1` and the Mnesia User's Guide for more information about disc-less nodes.

The set of nodes which makes up a Mnesia system is kept in a schema and it is possible to add and remove Mnesia nodes 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 start-up procedure, Mnesia will exchange schema information between the nodes in order to verify that the table definitions are compatible.

Each schema has a unique cookie which may be regarded as a unique schema identifier. The cookie must be the same on all nodes where Mnesia is supposed to run. See the Mnesia User's Guide for more information about these details.

The schema file, as well as all other files which 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)` may also be used.

`stop()` -> `stopped`

Stops Mnesia locally on the current node.

`application:stop(mnesia)` may also be used.

`subscribe(EventCategory)`

Ensures that a copy of all events of type `EventCategory` are sent to the caller. The event types available are described in the Mnesia User's Guide.

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

Call the `Fun` in a context which 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. See `mnesia:activity/4` and the Mnesia User's Guide for more details.

`system_info(InfoKey)` -> `Info` | `exit({aborted, Reason})`

Returns information about the Mnesia system, such as transaction statistics, `db_nodes`, and configuration parameters. Valid keys are:

- `all`. This argument returns a list of all local system information. Each element is a `{InfoKey, InfoVal}` tuples. *Note:* New `InfoKey`'s may be added and old undocumented `InfoKey`'s may be removed without notice.
- `access_module`. This argument returns the name of the module which is configured to be the activity access callback module.
- `auto_repair`. This argument returns `true` or `false` to indicate if Mnesia is configured to invoke the auto repair facility on corrupted disc files.
- `backup_module`. This argument returns the name of the module which is configured to be the backup callback module.
- `checkpoints`. This argument returns a list of the names of the checkpoints currently active on this node.
- `event_module`. This argument returns the name of the module which is the event handler callback module.

- `db_nodes`. This argument returns the nodes which make up the persistent database. Disc less nodes will only be included in the list of nodes if they explicitly has been added to the schema, e.g. with `mnesia:add_table_copy/3`. The function can be invoked even if Mnesia is not yet running.
- `debug`. This argument returns the current debug level of Mnesia.
- `directory`. This argument returns the name of the Mnesia directory. It can be invoked even if Mnesia is not yet running.
- `dump_log_load_regulation`. This argument returns a boolean which tells whether Mnesia is configured to load regulate the dumper process or not. This feature is temporary and will disappear in future releases.
- `dump_log_time_threshold`. This argument returns the time threshold for transaction log dumps in milliseconds.
- `dump_log_update_in_place`. This argument returns a boolean which tells whether Mnesia is configured to perform the updates in the dets files directly or if the updates should be performed in a copy of the dets files.
- `dump_log_write_threshold`. This argument returns the write threshold for transaction log dumps as the number of writes to the transaction log.
- `embedded_mnemosyne`. This argument returns a boolean which tells whether or not Mnesia is configured to start Mnemosyne as a supervised part of the Mnesia application as it did in previous releases. This feature is temporary and will disappear in a future release.
- `extra_db_nodes`. This argument returns a list of extra `db_nodes` to be contacted at start-up.
- `fallback_activated`. This argument returns true if a fallback is activated, otherwise false.
- `held_locks`. This argument returns a list of all locks held by the local Mnesia lock manager.
- `is_running`. This argument returns yes or no to indicate if Mnesia is running. It may also return `starting` or `stopping`. Can be invoked even if Mnesia is not yet running.
- `local_tables`. This argument returns a list of all tables which are configured to reside locally.
- `lock_queue`. This argument returns a list of all transactions that are queued for execution by the local lock manager.
- `log_version`. This argument returns the version number of the Mnesia transaction log format.
- `master_node_tables`. This argument returns a list of all tables with at least one master node.
- `protocol_version`. This argument returns the version number of the Mnesia inter-process communication protocol.
- `running_db_nodes`. This argument returns a list of nodes where Mnesia currently is running. This function can be invoked even if Mnesia is not yet running, but it will then have slightly different semantics. If Mnesia is down on the local node, the function will return those other `db_nodes` and `extra_db_nodes` that for the moment are up and running. If Mnesia is started, the function will return 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 may perform access of remote replicas. Before the schema merge Mnesia will only operate locally. Sometimes there may be more nodes included in the `running_db_nodes` list than all `db_nodes` and `extra_db_nodes` together.

- `schema_location`. This argument returns the initial schema location.
- `subscribers`. This argument returns a list of local processes currently subscribing to system events.
- `tables`. This argument returns a list of all locally known tables.
- `transactions`. This argument returns a list of all currently active local transactions.
- `transaction_failures`. This argument returns a number which indicates how many transactions have failed since Mnesia was started.
- `transaction_commits`. This argument returns a number which indicates how many transactions have terminated successfully since Mnesia was started.
- `transaction_restarts`. This argument returns a number which indicates how many transactions have been restarted since Mnesia was started.
- `transaction_log_writes`. This argument returns a number which indicates the number of write operation that have been performed to the transaction log since start-up.
- `use_dir`. This argument returns a boolean which indicates whether the Mnesia directory is used or not. Can be invoked even if Mnesia is not yet running.
- `version`. This argument returns the current version number of Mnesia.

```
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`. This argument returns a list of all local table information. Each element is a `{InfoKey, ItemVal}` tuples. *Note:* New `InfoItem`'s may be added and old undocumented `InfoItem`'s may be removed without notice.
- `access_mode`. This argument returns the access mode of the table. The access mode may either be `read_only` or `read_write`.
- `arity`. This argument returns the arity of records in the table as specified in the schema.
- `attributes`. This argument returns the table attribute names which are specified in the schema.
- `checkpoints`. This argument returns the names of the currently active checkpoints which involves this table on this node.
- `cookie`. This argument 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 initially created.
- `disc_copies`. This argument returns the nodes where a `disc_copy` of the table resides according to the schema.
- `disc_only_copies`. This argument returns the nodes where a `disc_only_copy` of the table resides according to the schema.
- `index`. This argument returns the list of index position integers for the table.

- `load_node`. This argument returns the name of the node that Mnesia loaded the table from. The structure of the returned value is unspecified but may be useful for debugging purposes.
- `load_order`. This argument returns the load order priority of the table. It is an integer and defaults to 0 (zero).
- `load_reason`. This argument returns the reason of why Mnesia decided to load the table. The structure of the returned value is unspecified but may be useful for debugging purposes.
- `local_content`. This argument returns `true` or `false` to indicate whether the table is configured to have locally unique content on each node.
- `master_nodes`. This argument returns the master nodes of a table.
- `memory`. This argument returns the number of words allocated to the table on this node.
- `ram_copies`. This argument returns the nodes where a `ram_copy` of the table resides according to the schema.
- `record_name`. This argument returns the record name, common for all records in the table
- `size`. This argument returns the number of records inserted in the table.
- `snmp`. This argument returns the SNMP struct. `[]` meaning that the table currently has no SNMP properties.
- `storage_type`. This argument 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 which only reside remotely.
- `subscribers`. This argument returns a list of local processes currently subscribing to local table events which involve this table on this node.
- `type`. This argument returns the table type, which is either `bag` or `set`.
- `user_properties`. This argument returns the user associated table properties of the table. It is a list of the stored property records.
- `version`. This argument returns the current version of the table definition. The table version is incremented when the table definition is changed. The table definition may be incremented directly when the table definition has been changed in a schema transaction, or when a committed table definition is merged with table definitions from other nodes during start-up.
- `where_to_read`. This argument returns the node where the table can be read. If the value `nowhere` is returned, the table is not loaded, or it resides at a remote node which is not running.
- `where_to_write`. This argument returns a list of the nodes that currently hold an active replica of the table.
- `wild_pattern`. This argument returns a structure which can be given to the various match functions for a certain table. A record tuple is where all record fields have the value `'_'`.

```
transaction(Fun [[, Args], Retries]) -> {aborted, Reason} | {atomic, ResultOfFun}
```

This function executes the functional object `Fun` with arguments `Args` as a transaction. The code which 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 aborted and the function `transaction/1` returns the tuple `{aborted, Reason}`.

If all is well, `{atomic, ResultOfFun}` is returned where `ResultOfFun` is the value of the last expression in `Fun`.

A function which adds a family to the database can be written as follows if we have 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 will ensure that either the whole family is added to the database, or the whole transaction aborts. For example, if the last child is badly formatted, or the executing process terminates due to an 'EXIT' signal while executing the family code, the transaction aborts. Accordingly, 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, should 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 `raise/2` function without interfering with each other.

Since Mnesia detects deadlocks, a transaction can be restarted any number of times. This function will attempt 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) -> {aborted, R} | {atomic, ok}
```

This function applies the argument `Fun` to all records in the table. `Fun` is a function which takes a record of the old type and returns a transformed record of the new type.

```
traverse_backup(Source, [SourceMod,] Target, [TargetMod,] Fun, Acc) -> {ok, LastAcc}
| {error, Reason}
```

With this function it is possible to iterate over a backup, either for the purpose of transforming it into a new backup, or just reading it. The arguments are explained briefly below. See the Mnesia User's Guide for additional details.

- `SourceMod` and `TargetMod` are the names of the modules which actually access the backup media.
- `Source` and `Target` are opaque data used exclusively by the modules `SourceMod` and `TargetMod` for the purpose of initializing 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}
```

Invokes `mnesia:uninstall_fallback([scope, global])`.

```
uninstall_fallback(Args) -> ok | {error, Reason}
```

This function is used to de-installed a fallback before it has been used to restore the database. This is normally a distributed operation that is either is performed on all nodes with disc resident schema or none. Uninstallation of fallbacks requires Erlang to be up and running 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 info in the local fallback.

`Args` is a list of the following tuples:

- `{module, BackupMod}`. See `mnesia:install_fallback/2` about the semantics.
- `{scope, Scope}` See `mnesia:install_fallback/2` about the semantics.
- `{mnesia_dir, AlternateDir}` See `mnesia:install_fallback/2` about the semantics.

```
unsubscribe(EventCategory)
```

Stops sending events of type `EventCategory` to the caller.

```
wait_for_tables(TabList, Timeout) -> ok | {timeout, BadTabList}
```

Some applications need to wait for certain tables to be accessible in order to do useful work. `mnesia:wait_for_tables/2` hangs until all tables in the `TabList` are accessible, or until `timeout` is reached.

```
wread({Tab, Key}) -> transaction abort | RecordList
```

Invokes `mnesia:read(Tab, Key, write)`.

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

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

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

Writes the record `Record` to the table `Tab`.

The function returns `ok`, or aborts if an error occurs. For example, the transaction aborts if no person table exists.

The semantics of this function is context sensitive. See `mnesia:activity/4` for more information. In transaction context it acquires a lock of type `LockKind`. The following lock types are supported: `write` and `sticky_write`.

```
write_lock_table(Tab) -> ok | transaction abort
```

Invokes `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. The default is `mnesia`.
- `-mnesia auto_repair` `true | false`. This flag controls whether Mnesia will try to automatically repair files that have not been properly closed. The default is `true`.
- `-mnesia backup_module` `Module`. The name of the Mnesia backup callback module. The default is `mnesia_backup`.
- `-mnesia debug` `Level` Controls the debug level of Mnesia. Possible values are:
  - `none` No trace outputs at all. This is the default setting.
  - `verbose` Activates tracing of important debug events. These debug events generate `{mnesia_info, Format, Args}` system events. Processes may subscribe to these events with `mnesia:subscribe/1`. The events are always sent to Mnesia's 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 may 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 level debug. On this debug level, the Mnesia event handler starts subscribing to updates on all Mnesia tables. This level is only intended for debugging small toy systems since many large events may be generated.
  - `false` An alias for `none`.
  - `true` An alias for `debug`.

- `-mnesia dir` Directory. The name of the directory where all Mnesia data is stored. The name of the directory must be unique for the current node. Two nodes may, under no circumstances, share the same Mnesia directory. The results are totally unpredictable.
- `-mnesia dump_log_load_regulation true | false`. Controls if the log dumps should be performed as fast as possible or if the dumper should do its own load regulation. This feature is temporary and will disappear in a future release. The default is `false`.
- `-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. The default is `true`.
- `-mnesia dump_log_write_threshold Max`, where `Max` is an integer which specifies the maximum number of writes allowed to the transaction log before a new dump of the log is performed. It defaults to 100 log writes.
- `-mnesia dump_log_time_threshold Max`, where `Max` is an integer which specifies the dump log interval in milliseconds. It defaults to 3 minutes. If a dump has not been performed within `dump_log_time_threshold` milliseconds, then a new dump is performed regardless of how many writes have been performed.
- `-mnesia embedded_mnemosyne true | false`. By default this is set to `false`, but if it is set to `true` Mnesia will start Mnemosyne as a supervised part of the Mnesia application (as it did in previous releases). This feature is temporary and will disappear in a future release. The default is `false`.
- `-mnesia event_module Module`. The name of the Mnesia event handler callback module. The 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 should also establish contact. The default value is the empty list `[]`.
- `-mnesia max_wait_for_decision Timeout`. Specifies how long Mnesia will wait for other nodes to share their knowledge regarding the outcome of an unclear transaction. By default the `Timeout` is set to the atom `infinity`, which implies that if Mnesia upon startup encounters a “heavyweight transaction” whose outcome is unclear, the local Mnesia will wait until Mnesia is started on some (in worst cases all) of the other nodes that were involved in the interrupted transaction. This is a very rare situation, but when/if it happens, Mnesia does not guess if the transaction on the other nodes was committed or aborted. Mnesia will wait until it knows the outcome and then act accordingly.  
If `Timeout` is set to an integer value in milliseconds, Mnesia will force “heavyweight transactions” to be finished, even if the outcome of the transaction for the moment is unclear. After `Timeout` milliseconds, Mnesia will commit/abort the transaction and continue with the startup. This may lead to a situation where the transaction is committed on some nodes and aborted on other nodes. If the transaction was a schema transaction, the inconsistency may be fatal.
- `-mnesia schema_location Loc` controls where Mnesia will look for its schema. The parameter `Loc` may 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 start-up, a tiny new schema is generated. This default schema just contains the definition of the schema table and only resides on the local node. Since no other nodes are

found in the default schema, the configuration parameter `extra_db_nodes` must be used in order to let the node share its table definitions with other nodes. (The `extra_db_nodes` parameter may also be used on disc based nodes.)

`opt_disc` Optional disc. The schema may reside either 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`. The 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

`mnesia_registry(3)`, `mnesia_session(3)`, `mnemosyne(3)`, `dets(3)`, `ets(3)`, `disc_log(3)`, `application(3)`

# mnesia\_registry (Module)

The module `mnesia_registry` is usually part of `erl_interface`, but for the time being, it is a part of the Mnesia application.

`mnesia_registry` is mainly an module intended for internal usage within OTP, but it has two functions that are exported for public use.

On C-nodes `erl_interface` has support for registry tables. These reside in RAM on the C-node but they may also be dumped into Mnesia tables. By default, the dumping of registry tables via `erl_interface` causes a corresponding Mnesia table to be created with `mnesia_registry:create_table/1` if necessary.

The tables that are created with these functions can be administered as all other Mnesia tables. They may be included in backups or replicas may be added etc. The tables are in fact normal Mnesia tables owned by the user of the corresponding `erl_interface` registries.

## Exports

```
create_table(Tab) -> ok | exit(Reason)
```

This is 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 will only reside on the local node and its storage type will be the same as the schema table on the local node, ie. `{ram_copies, [node()]}` or `{disc_copies, [node()]}`.

It is this function that is used by `erl_interface` to create the Mnesia table if it did not already exist.

```
create_table(Tab, TabDef) -> ok | exit(Reason)
```

This is 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 should reside as `disc_only_copies` on all nodes a call would look like:

```
TabDef = [{disc_only_copies, node()|nodes()}],
mnesia_registry:create_table(my_reg, TabDef)
```

## See Also

`mnesia(3)`, `erl_interface(3)`

# List of Figures

## Chapter 1: MNESIA User's Guide

1.1 Company Entity-Relation Diagram . . . . . 7



# List of Tables

## Chapter 1: MNESIA User's Guide

- 1.1 Employee . . . . . 11
- 1.2 At\_dep . . . . . 11
- 1.3 In\_proj . . . . . 11
- 1.4 Mnemosyne configuration parameters . . . . . 89



# Index

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

abort/1  
    *mnesia* , 116

activate\_checkpoint/1  
    *mnesia* , 116

activity/3  
    *mnesia* , 117

activity/4  
    *mnesia* , 117

add\_table\_copy/3  
    *mnesia* , 118

add\_table\_index/2  
    *mnesia* , 119

all\_keys/1  
    *mnesia* , 119

async\_dirty/3  
    *mnesia* , 119

backup/2  
    *mnesia* , 119

backup\_checkpoint/3  
    *mnesia* , 119

change\_table\_access\_mode/2  
    *mnesia* , 120

change\_table\_copy\_type/3  
    *mnesia* , 120

change\_table\_load\_order/2  
    *mnesia* , 120

create\_schema/1  
    *mnesia* , 120

create\_table/1  
    *mnesia\_registry* , 143

create\_table/2  
    *mnesia* , 120

*mnesia\_registry* , 143

deactivate\_checkpoint/1  
    *mnesia* , 122

del\_table\_copy/2  
    *mnesia* , 122

del\_table\_index/2  
    *mnesia* , 122

delete/2  
    *mnesia* , 122

delete/3  
    *mnesia* , 122

delete\_object/1  
    *mnesia* , 122

delete\_object/3  
    *mnesia* , 122

delete\_schema/1  
    *mnesia* , 123

delete\_table/1  
    *mnesia* , 123

dirty\_all\_keys/1  
    *mnesia* , 123

dirty\_delete/2  
    *mnesia* , 123

dirty\_delete\_object/1  
    *mnesia* , 123

dirty\_delete\_object/2  
    *mnesia* , 123

dirty\_first/1  
    *mnesia* , 123

dirty\_index\_match\_object/2  
    *mnesia* , 124

dirty\_index\_match\_object/3  
    *mnesia* , 124

- dirty\_index\_read/3
  - mnesia* , 124
- dirty\_match\_object/1
  - mnesia* , 124
- dirty\_match\_object/2
  - mnesia* , 124
- dirty\_next/2
  - mnesia* , 124
- dirty\_read/2
  - mnesia* , 124
- dirty\_slot/2
  - mnesia* , 124
- dirty\_update\_counter/3
  - mnesia* , 124, 125
- dirty\_write/1
  - mnesia* , 125
- dirty\_write/2
  - mnesia* , 125
- dump\_log/0
  - mnesia* , 125
- dump\_tables/1
  - mnesia* , 125
- dump\_to\_textfile/1
  - mnesia* , 125
- error\_description/1
  - mnesia* , 125
- ets/3
  - mnesia* , 126
- force\_load\_table/1
  - mnesia* , 126
- index\_match\_object/2
  - mnesia* , 126
- index\_match\_object/4
  - mnesia* , 126
- index\_read/3
  - mnesia* , 127
- info/0
  - mnesia* , 127
- install\_fallback/1
  - mnesia* , 127
- install\_fallback/2
  - mnesia* , 127
- load\_textfile/1
  - mnesia* , 128
- lock/2
  - mnesia* , 128
- match\_object/1
  - mnesia* , 129
- match\_object/3
  - mnesia* , 129
- mnesia*
  - abort/1, 116
  - activate\_checkpoint/1, 116
  - activity/3, 117
  - activity/4, 117
  - add\_table\_copy/3, 118
  - add\_table\_index/2, 119
  - all\_keys/1, 119
  - async\_dirty/3, 119
  - backup/2, 119
  - backup\_checkpoint/3, 119
  - change\_table\_access\_mode/2, 120
  - change\_table\_copy\_type/3, 120
  - change\_table\_load\_order/2, 120
  - create\_schema/1, 120
  - create\_table/2, 120
  - deactivate\_checkpoint/1, 122
  - del\_table\_copy/2, 122
  - del\_table\_index/2, 122
  - delete/2, 122
  - delete/3, 122
  - delete\_object/1, 122
  - delete\_object/3, 122
  - delete\_schema/1, 123
  - delete\_table/1, 123
  - dirty\_all\_keys/1, 123
  - dirty\_delete/2, 123
  - dirty\_delete\_object/1, 123
  - dirty\_delete\_object/2, 123
  - dirty\_first/1, 123
  - dirty\_index\_match\_object/2, 124
  - dirty\_index\_match\_object/3, 124
  - dirty\_index\_read/3, 124
  - dirty\_match\_object/1, 124
  - dirty\_match\_object/2, 124
  - dirty\_next/2, 124
  - dirty\_read/2, 124
  - dirty\_slot/2, 124
  - dirty\_update\_counter/3, 124, 125
  - dirty\_write/1, 125

---

dirty\_write/2, 125  
 dump\_log/0, 125  
 dump\_tables/1, 125  
 dump\_to\_textfile/1, 125  
 error\_description/1, 125  
 ets/3, 126  
 force\_load\_table/1, 126  
 index\_match\_object/2, 126  
 index\_match\_object/4, 126  
 index\_read/3, 127  
 info/0, 127  
 install\_fallback/1, 127  
 install\_fallback/2, 127  
 load\_textfile/1, 128  
 lock/2, 128  
 match\_object/1, 129  
 match\_object/3, 129  
 move\_table\_copy/3, 129  
 read/2, 130  
 read/3, 130  
 read\_lock\_table/1, 130  
 report\_event/1, 130  
 restore/2, 130  
 s\_delete/2, 131  
 s\_delete\_object/1, 131  
 s\_write/1, 131  
 schema/0, 131  
 schema/1, 131  
 set\_debug\_level/1, 131  
 set\_master\_nodes/1, 131  
 set\_master\_nodes/2, 132  
 snmp\_close\_table/1, 132  
 snmp\_get\_mnesia\_key/2, 132  
 snmp\_get\_next\_index/2, 132  
 snmp\_get\_row/2, 132  
 snmp\_open\_table/2, 132  
 start/0, 133  
 stop/0, 134  
 subscribe/1, 134  
 sync\_dirty/3, 134  
 system\_info/1, 134  
 table\_info/2, 136  
 transaction/3, 137  
 transform\_table/3, 138  
 traverse\_backup/6, 139  
 uninstall\_fallback/0, 139  
 uninstall\_fallback/1, 139  
 unsubscribe/1, 139  
 wait\_for\_tables/2, 139  
 wread/2, 139  
 write/1, 140  
 write/3, 140  
 write\_lock\_table/1, 140  
  
*mnesia\_registry*  
   create\_table/1, 143  
   create\_table/2, 143  
 move\_table\_copy/3  
   *mnesia* , 129  
  
 read/2  
   *mnesia* , 130  
 read/3  
   *mnesia* , 130  
 read\_lock\_table/1  
   *mnesia* , 130  
 report\_event/1  
   *mnesia* , 130  
 restore/2  
   *mnesia* , 130  
  
 s\_delete/2  
   *mnesia* , 131  
 s\_delete\_object/1  
   *mnesia* , 131  
 s\_write/1  
   *mnesia* , 131  
 schema/0  
   *mnesia* , 131  
 schema/1  
   *mnesia* , 131  
 set\_debug\_level/1  
   *mnesia* , 131  
 set\_master\_nodes/1  
   *mnesia* , 131  
 set\_master\_nodes/2  
   *mnesia* , 132  
 snmp\_close\_table/1  
   *mnesia* , 132  
 snmp\_get\_mnesia\_key/2  
   *mnesia* , 132  
 snmp\_get\_next\_index/2  
   *mnesia* , 132  
 snmp\_get\_row/2  
   *mnesia* , 132  
 snmp\_open\_table/2  
   *mnesia* , 132  
 start/0

*mnesia* , 133

stop/0  
    *mnesia* , 134

subscribe/1  
    *mnesia* , 134

sync\_dirty/3  
    *mnesia* , 134

system\_info/1  
    *mnesia* , 134

table\_info/2  
    *mnesia* , 136

transaction/3  
    *mnesia* , 137

transform\_table/3  
    *mnesia* , 138

traverse\_backup/6  
    *mnesia* , 139

uninstall\_fallback/0  
    *mnesia* , 139

uninstall\_fallback/1  
    *mnesia* , 139

unsubscribe/1  
    *mnesia* , 139

wait\_for\_tables/2  
    *mnesia* , 139

wread/2  
    *mnesia* , 139

write/1  
    *mnesia* , 140

write/3  
    *mnesia* , 140

write\_lock\_table/1  
    *mnesia* , 140