

Mnesia

Copyright © 1997-2017 Ericsson AB. All Rights Reserved.
Mnesia 4.12.5
February 22, 2017

Copyright © 1997-2017 Ericsson AB. All Rights Reserved.
The contents of this file are subject to the Erlang Public License, Version 1.1, (the "License"); you may not use this file except in compliance with the License. You should have received a copy of the Erlang Public License along with this software. If not, it can be retrieved online at http://www.erlang.org/. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. Ericsson AB. All Rights Reserved..

February 22, 2017

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.

1.1.1 About Mnesia

The management of data in telecommunications system has many aspects whereof some, but not all, are addressed by traditional commercial DBMSs (Data Base Management Systems). In particular the very high level of fault tolerance which is required in many nonstop systems, combined with requirements on the DBMS to run in the same address space as the application, have led us to implement a brand new DBMS. called Mnesia. Mnesia is implemented in, and very tightly connected to, the programming language Erlang and it provides the functionality that is necessary for the implementation of fault tolerant telecommunications systems. Mnesia is a multiuser Distributed DBMS specially made for industrial telecommunications applications written in the symbolic programming language Erlang, which is also the intended target language. Mnesia tries to address all of the data management issues required for typical telecommunications systems and it has a number of features that are not normally found in traditional databases.
In telecommunications applications there are different needs from the features provided by traditional DBMSs. The applications now implemented in the Erlang language need a mixture of a broad range of features, which generally are not satisfied by traditional DBMSs. Mnesia is designed with requirements like the following in mind:

- Fast real-time key/value lookup
- Complicated non real-time queries mainly for operation and maintenance
- Distributed data due to distributed applications
- High fault tolerance
- Dynamic re-configuration
- Complex objects

What sets Mnesia apart from most other DBMSs is that it is designed with the typical data management problems of telecommunications applications in mind. Hence Mnesia combines many concepts found in traditional databases, such as transactions and queries with concepts found in data management systems for telecommunications applications, such as very fast real-time operations, configurable degree of fault tolerance (by means of replication) and the ability to reconfigure the system without stopping or suspending it. Mnesia is also interesting due to its tight coupling to the programming language Erlang, thus almost turning Erlang into a database programming language. This has many benefits, the foremost is that the impedance mismatch between data format used by the DBMS and data format used by the programming language, which is used to manipulate the data, completely disappears.

1.1.2 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

QLC 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 QLC have their own documentation as part of the OTP documentation set. Below are the main features of Mnesia Session and QLC when used in conjunction with Mnesia:

- *QLC* has the ability to optimize the query compiler for the Mnesia Database Management System, essentially making the DBMS more efficient.
- *QLC*, can be used as a database programming language for Mnesia. It includes a notation called "list comprehensions" and can be used to make complex database queries over a set of tables.
- *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 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 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 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 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 6, "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 7, "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.

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.

1.2.1 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 (BEAM) emulator version 4.9

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

In 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.

1.2.2 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.

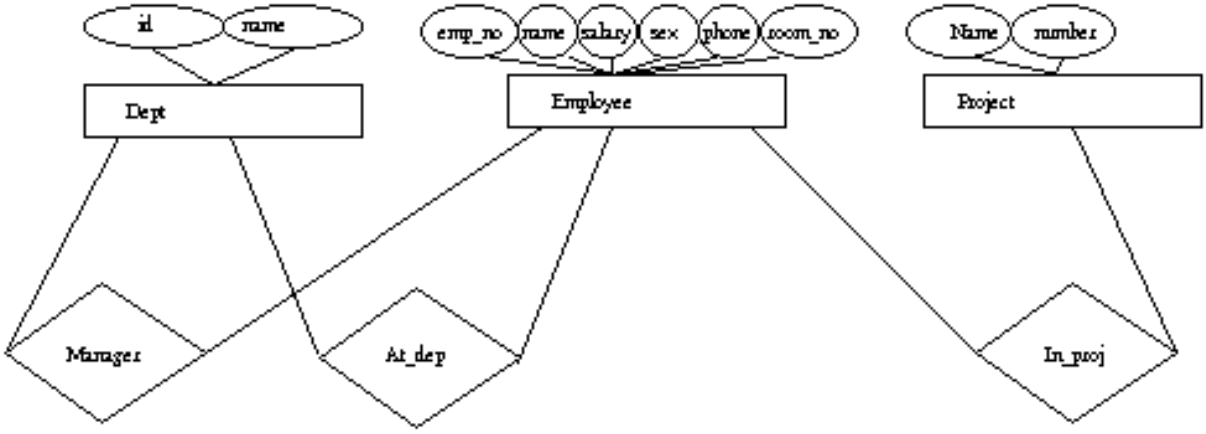


Figure 2.1: Company Entity-Relation Diagram

The database model looks as follows:

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

Defining Structure and Content

We first enter our record definitions into a text file named `company.hrl`. 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.*

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 (BEAM) emulator version 4.9

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

The following program module creates and populates previously defined tables:

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

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

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*.
- `mnesia:start().` This function starts Mnesia. This function is fully explained in Chapter 3: *Starting Mnesia*.

Continuing the dialogue with the Erlang shell will produce the following:

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

A set of tables is created:

- `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*.

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:
  - `Emp` is an employee record.
  - `DeptId` is the identity of the department where the employee is working.
  - `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 the Erlang Reference Manual, "Fun Expressions".

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 2.1: 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 2.2: 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 2.3: 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 a 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

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

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

Since we want to update the record using `mnesia:write/1` after we have increased the salary we acquire a write lock (third argument to read) when we read the record from the table.

It is not always the case that we can directly read the values from the table, we might need to search the table or several tables to get the data we want, this is done by writing database queries. Queries are always more expensive operations than direct lookups done with `mnesia:read` and should be avoided in performance critical code.

There are two methods for writing database queries:

- Mnesia functions
- QLC

Mnesia functions

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

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

Select must always run within an activity such as a transaction. To be able to call from the shell we might construct a function as:

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

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

This function can be called from the shell as follows:

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

See also the *Pattern Matching* chapter for a description of select and its syntax.

Using QLC

This section contains simple introductory examples only. Refer to *QLC reference manual* for a full description of the QLC query language. Using QLC might be more expensive than using Mnesia functions directly but offers a nice syntax.

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

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

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

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

This function can be called from the shell as follows:

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

In traditional relational database terminology, 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#employee.name` such that `E` is taken from the table of employees and the `sex` attribute of each records is equal with the atom `female`.

The whole list comprehension must be given to the `qlc:q/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 = qlc:q([E || E <- mnesia:table(employee),
                                E#employee.sex == female]),
  Fs = qlc:e(Q),
                over_write(Fs, Amount)
        end,
    mnesia:transaction(F).

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

The function `raise_females/1` returns the tuple `{atomic, Number}`, where `Number` is the number of female employees who received a salary increase. 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.

1.3.1 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:clear_table(Tab)`. This function permanently deletes all entries in 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.
- `mnesia:transform_table(Tab, Fun, NewAttributeList, NewRecordName)`. 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 a record of the old type, and returns the record of the new type. The table key may not be changed.

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

Transformer =
```

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

The `Fun` argument can also be the atom `ignore`, it indicates that only the meta data about the table will be updated. Usage of `ignore` is not recommended (since it creates inconsistencies between the meta data and the actual data) but included as a possibility for the user to do his own (off-line) transform.

- `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`.

1.3.2 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*, 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.

1.3.3 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 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:

- On the node called gin:

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

- On the node called skeppet:

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

- On one of the two nodes:

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

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

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:

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

- Tables that reside on remote nodes are available to other nodes as soon as they are loaded.

Table initialization is asynchronous, the function call `mnesia:start()` returns the atom `ok` and then starts to initialize the different tables. Depending on the size of the database, this 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 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.

1.3.4 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`, `ordered_set` or `bag`. The default value is `set`. Note: currently 'ordered_set' is not supported for 'disc_only_copies' tables. A table of type `set` or `ordered_set` has either zero or one record per key. Whereas a table of type `bag` can have an arbitrary number of records per key. The key for each record is always the first attribute of the record.

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

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

    This transaction 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:

    - read operations must be very fast and performed in RAM
    - 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*.

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
- Iteration

1.4.1 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, write),
                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 *A*tomicity, *C*onsistency,*I*solation, and *D*urability, 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.

1.4.2 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.

1.4.3 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. Even if the following functions are executed inside a transaction no locks will be acquired. 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 `set` or `bag` 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 any process perform a write operation 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:dirty_last(Tab)` This function works exactly like `mnesia:dirty_first/1` but returns the last object in Erlang term order for the `ordered_set` table type. For all other table types, `mnesia:dirty_first/1` and `mnesia:dirty_last/1` are synonyms.
- `mnesia:dirty_prev(Tab, Key)` This function works exactly like `mnesia:dirty_next/2` but returns the previous object in Erlang term order for the ordered_set table type. For all other table types, `mnesia:dirty_next/2` and `mnesia:dirty_prev/2` are synonyms.
- `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:

  - it is much more efficient

  - 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_select(Tab, Pat)`. This function is the dirty equivalent of `mnesia:select/2`.
- `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`.

1.4.4 Record Names versus Table Names

In Mnesia, all records in a table must have the same name. All the records must be instances of the same record type. The record name 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 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).
```

1.4.5 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:select/3 (select/2)
- mnesia:foldl/3 (foldl/4, foldr/3, foldr/4)
- 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
- sync_transaction
- async_dirty
- sync_dirty
- ets

By passing the same "fun" as argument to the function `mnesia:sync_transaction(Fun [, Args])` it will be performed in synced transaction context. Synced transactions waits until all active replicas has committed the transaction (to disc) before returning from the mnesia:sync_transaction call. Using sync_transaction 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, and also when combining transaction writes with dirty_reads. This is also useful in situations where an application performs frequent or voluminous updates which may overload Mnesia on other nodes.

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.

You can check if your code is executed within a transaction with `mnesia:is_transaction/0`, it returns `true` when called inside a transaction context and false otherwise.

Mnesia tables with storage type RAM_copies and disc_copies are implemented internally as "ets-tables" and it is possible for applications to access the these tables directly. This is only recommended if all options have been weighed and the possible outcomes are understood. By passing the earlier mentioned "fun" to the function `mnesia:ets(Fun [, Args])` it 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 `QLC` queries within an activity context with a customized callback module. By providing table information about table indices and other `QLC` requirements, `QLC` may be used as a generic query language to access virtual tables.

QLC queries may be performed in all these activity contexts (transaction, sync_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.

1.4.6 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.

It is also possible to mix different activity access contexts while nesting, but the dirty ones (async_dirty,sync_dirty and ets) will inherit the transaction semantics if they are called inside a transaction and thus it will grab locks and use two or three phase commit.

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

1.4.7 Pattern Matching

When it is not possible to use `mnesia:read/3` Mnesia provides the programmer with several functions for matching records against a pattern. The most useful functions of these are:

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

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

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

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

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

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:

```
Pat = #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 is equal to their room number:

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

The function `mnesia:match_object/3` lacks some important features that `mnesia:select/3` have. For example `mnesia:match_object/3` can only return the matching records, and it can not express constraints other then equality. If we want to find the names of the male employees on the second floor we could write:

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

Select can be used to add additional constraints and create output which can not be done with `mnesia:match_object/3`.

The second argument to select is a `MatchSpecification`. A `MatchSpecification` is list of `MatchFunctions`, where each `MatchFunction` consists of a tuple containing `{MatchHead, MatchCondition, MatchBody}`. `MatchHead` is the same pattern used in `mnesia:match_object/3` described above. `MatchCondition` is a list of additional constraints applied to each record, and `MatchBody` is used to construct the return values.

A detailed explanation of match specifications can be found in the *Erts users guide: Match specifications in Erlang* , and the ets/dets documentations may provide some additional information.

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

Warning:

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

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. Hence if the table is large, this can become a time consuming operation, but it can be remedied with indices (refer to Chapter 5: *Indexing*) if `mnesia:match_object` is used.

QLC queries can also be used to search Mnesia tables. By using `mnesia:table/[1|2]` as the generator inside a QLC query you let the query operate on a mnesia table. Mnesia specific options to `mnesia:table/2` are {lock, Lock}, {n_objects,Integer} and {traverse, SelMethod}. The `lock` option specifies whether mnesia should acquire a read or write lock on the table, and `n_objects` specifies how many results should be returned in each chunk to QLC. The last option is `traverse` and it specifies which function mnesia should use to traverse the table. Default `select` is used, but by using `{traverse, {select, MatchSpecification}}` as an option to `mnesia:table/2` the user can specify it's own view of the table.

If no options are specified a read lock will acquired and 100 results will be returned in each chunk, and select will be used to traverse the table, i.e.:

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

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

1.4.8 Iteration

Mnesia provides a couple of functions which iterates over all the records in a table.

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

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

The first time the `Fun` is applied `Acc0` will be the second argument. The next time the `Fun` is called the return value from the previous call, will be used as the second argument. The term the last call to the Fun returns will be the return value of the `fold[lr]` function.

The difference between `foldl` and `foldr` is the order the table is accessed for `ordered_set` tables, for every other table type the functions are equivalent.

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

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

For example finding all the employees who has a salary below 10 could look like:

```
find_low_salaries() ->
  Constraint =
       fun(Emp, Acc) when Emp#employee.salary < 10 ->
              [Emp | Acc];
```

```
              (_, Acc) ->
                  Acc
          end,
    Find = fun() -> mnesia:foldl(Constraint, [], employee) end,
    mnesia:transaction(Find).
```

Raising the salary to 10 for everyone with a salary below 10 and return the sum of all raises:

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

A lot of nice things can be done with the iterator functions but some caution should be taken about performance and memory utilization for large tables.

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

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

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

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

If records are written and deleted during the traversal, use `mnesia:fold[lr]/4` with a `write` lock. Or `mnesia:write_lock_table/1` when using first and next.

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

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

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.

1.5.1 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:

- `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*.

1.5.2 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:
  - The `mnesia:dump_tables/1` function can be used to dump RAM table replicas to disc.
  - The table replicas can be backed up; either from RAM, or from disc if dumped there with 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* 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.

1.5.3 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. Note: In `ordered_set` tables the records will be ordered per fragment, and the the order is undefined in results returned by select and match_object.

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.

`{hash_module, Atom}`

Enables definition of an alternate hashing scheme. The module must implement the `mnesia_frag_hash` callback behaviour (see the reference manual). This property may explicitly be set at table creation. The default is `mnesia_frag_hash`.

Older tables that was created before the concept of user defined hash modules was introduced, uses the `mnesia_frag_old_hash` module in order to be backwards compatible. The `mnesia_frag_old_hash` is still using the poor deprecated `erlang:hash/1` function.

`{hash_state, Term}`

Enables a table specific parameterization of a generic hash module. This property may explicitly be set at table creation. The default is `undefined`.

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

```
 {sec_dict_frag6,0},
 {sec_dict_frag7,0},
 {sec_dict_frag8,0}]
(a@sam)17>
```

Management of Fragmented Tables

The function `mnesia:change_table_frag(Tab, Change)` is intended to be used for reconfiguration of fragmented tables. 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 lose 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/1` 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 pro-type 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.

1.5.4 Local Content Tables

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

If 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.

1.5.5 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. It is also possible to use `mnesia:change_config/2` to assign a value to 'extra_db_nodes' and force a connection after mnesia have been started, i.e. mnesia:change_config(extra_db_nodes, NodeList).

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.

1.5.6 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.

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

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.

1.5.7 Mnesia Event Handling

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

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

`mnesia:subscribe(Event-Category)`

Ensures that a copy of all events of type `Event-Category` are sent to the calling process.

`mnesia:unsubscribe(Event-Category)`

Removes the subscription on events of type `Event-Category`

`Event-Category` may either be the atom `system`, the atom `activity`, or one of the tuples `{table, Tab, simple}`, `{table, Tab, detailed}`. The old event-category `{table, Tab}` is the same event-category as `{table, Tab, simple}`. The subscribe functions activate a subscription of events. The events are delivered as messages to the process evaluating the `mnesia:subscribe/1` function. The syntax of system events is `{mnesia_system_event, Event}`, `{mnesia_activity_event, Event}` for activity events, and `{mnesia_table_event, Event}` for table events. What the various event types mean is described below.

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.

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 i `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`.

Activity Events

Currently, there is only one type of activity event:

`{complete, ActivityID}`

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

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

Table Events

The final category of events are table events, which are events related to table updates. There are two types of table events simple and detailed.

The simple table events are tuples 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.

The detailed table events are tuples looking like this: `{Oper, Table, Data, [OldRecs], ActivityId}`. Where `Oper` is the operation performed. `Table` is the table involved in the operation, `Data` is the record/oid written/deleted. `OldRecs` is the contents before the operation. and `ActivityId` is the identity of the transaction performing the operation. The various table related events that may occur are:

`{write, Table, NewRecord, [OldRecords], ActivityId}`

a new record has been written. NewRecord contains the new value of the record and OldRecords contains the records before the operation is performed. Note that the new content is dependent on the type of the table.

`{delete, Table, What, [OldRecords], ActivityId}`

records has possibly been deleted `What` is either {Table, Key} or a record {RecordName, Key, ...} that was deleted. Note that the new content is dependent on the type of the table.

1.5.8 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
```

1.5.9 Concurrent Processes in Mnesia

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

A group of functions or processes can be called within a transaction. A transaction 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* for more information.

1.5.10 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, Atrribute2 ....}.
    {Typename, Attribute1, Atrribute2 ....}.
```

`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 (BEAM) emulator version 4.9

    Eshell V4.9  (abort with ^G)
    1> mnesia:load_textfile("FRUITS").
    New table fruit
    New table vegetable
    {atomic,ok}
    2> mnesia:info().
    ---> Processes holding locks <---
    ---> Processes waiting for locks <---
    ---> Pending (remote) transactions <---
    ---> Active (local) transactions <---
    ---> Uncertain transactions <---
```

```
     ---> Active tables <---
     vegetable      : with 2 records occuping 299 words of mem
     fruit          : with 2 records occuping 291 words of mem
     schema         : with 3 records occuping 401 words of mem
     ===> System info in version "1.1", debug level = none <===
     opt_disc. Directory "/var/tmp/Mnesia.nonode@nohost" is used.
     use fallback at restart = false
     running db nodes = [nonode@nohost]
     stopped db nodes = []
     remote           = []
     ram_copies       = [fruit,vegetable]
     disc_copies      = [schema]
     disc_only_copies = []
     [{nonode@nohost,disc_copies}] = [schema]
     [{nonode@nohost,ram_copies}] = [fruit,vegetable]
     3 transactions committed, 0 aborted, 0 restarted, 2 logged to disc
     0 held locks, 0 in queue; 0 local transactions, 0 remote
     0 transactions waits for other nodes: []
     ok
     3>
```

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

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

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

In reality, data models are seldom fully normalized. A realistic alternative to a normalized database model would be a data model 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) ->
    Q = qlc:q(
          [E || E <- mnesia:table(employee),
                E#employee.salary > Salary,
                E#employee.dept == Dep]
  ),
    F = fun() -> qlc:e(Q) end,
    transaction(F).
```

This code is 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

1.6.1 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.

1.6.2 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.

1.6.3 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 .DCD file on the disc. When the Mnesia system is started, the RAM table will initially be loaded with data from its .DCD file.

1.6.4 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 defaults 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.

1.6.5 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*:

- 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:

- ```
  % erl  -sname klacke -mnesia dir '"/ldisc/scratch/klacke"'
  ```
- ```
  Erlang (BEAM) emulator version 4.9

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

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

  The schema in the backup file FALLBACK.BUP has been used to generate the file `schema.DAT`. Since 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.
- 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    86 May 26 19:07 LATEST.LOG
  -rw-rw-r-- 1 klacke staff    94 May 26 19:07 foo.DCD
  ```

```
-rw-rw-r-- 1 klacke staff  6679 May 26 19:07 schema.DAT
```

Where a file `foo.DCD` has been created. This file will eventually store 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, .DCD and .DCL data files. For example, if the operation "write record `{foo, 4, elvis, 6}`" is listed in the log, Mnesia inserts the operation into the file `foo.DCL`, later when Mnesia thinks the .DCL has become to large the data is moved to the .DCD file. 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 one .DAT file. This contain the schema itself, contained in the `schema.DAT` file. The DAT files are indexed files, and it is efficient to insert and search for records in these files with a specific key. The .DAT files are used for the schema and for `disc_only_copies` tables. The Mnesia data files are currently implemented 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.

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

1.6.6 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 affect 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.

1.6.7 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.

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

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

1.6.8 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).

1.6.9 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)*, `{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 the section *Checkpoints* 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* 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, [TargeMod,] 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 `{ValGoodBackupItems, 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

1.7.1 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)`

1.8.1 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

1.9.1 mnesia_backup callback behavior

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

-module(mnesia_backup).

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

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

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

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

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

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

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

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

```
    end.

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

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

%% Opens backup media for read
%%
%% Returns {ok, OpaqueData} or {error, Reason}
open_read(OpaqueData) ->
    File = OpaqueData,
    case file:read_file_info(File) of
 {error, Reason} ->
     {error, Reason};
 _FileInfo -> %% file exists
     case disk_log:open([{file, File},
    {name, make_ref()},
    {repair, false},
    {mode, read_only},
    {linkto, self()}]) of
  {ok, Fd} ->
      {ok, #restore{file = File, file_desc = Fd, cont = start}};
  {repaired, Fd, _, {badbytes, 0}} ->
      {ok, #restore{file = File, file_desc = Fd, cont = start}};
  {repaired, Fd, _, _} ->
      {ok, #restore{file = File, file_desc = Fd, cont = start}};
  {error, Reason} ->
      {error, Reason}
     end
    end.

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

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

1.10 Appendix C: The Activity Access Call Back Interface

1.10.1 mnesia_access callback behavior

```
-module(mnesia_frag).

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

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

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

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

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

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

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

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

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

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


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


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

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

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

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

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

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

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

table_info2(ActivityId, Opaque, Tab, Frag, Item) ->
    case Item of
 size ->
     SumFun = fun({_, Size}, Acc) -> Acc + Size end,
     lists:foldl(SumFun, 0, frag_size(ActivityId, Opaque, Tab));
 memory ->
     SumFun = fun({_, Size}, Acc) -> Acc + Size end,
     lists:foldl(SumFun, 0, frag_memory(ActivityId, Opaque, Tab));
 base_table ->
     lookup_prop(Tab, base_table);
 node_pool ->
     lookup_prop(Tab, node_pool);
 n_fragments ->
     FH = lookup_frag_hash(Tab),
     FH#frag_state.n_fragments;
 foreign_key ->
     FH = lookup_frag_hash(Tab),
```

```
    FH#frag_state.foreign_key;
 foreigners ->
    lookup_foreigners(Tab);
 n_ram_copies ->
    length(val({Tab, ram_copies}));
 n_disc_copies ->
    length(val({Tab, disc_copies}));
 n_disc_only_copies ->
    length(val({Tab, disc_only_copies}));

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

first(ActivityId, Opaque, Tab) ->
   case ?catch_val({Tab, frag_hash}) of
 {'EXIT', _} ->
    mnesia:first(ActivityId, Opaque, Tab);
 FH ->
    FirstFrag = Tab,
    case mnesia:first(ActivityId, Opaque, FirstFrag) of
  '$end_of_table' ->
     search_first(ActivityId, Opaque, Tab, 1, FH);
  Next ->
     Next
    end
   end.

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

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

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

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

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

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

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

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

1.11 Appendix D: The Fragmented Table Hashing Call Back Interface

1.11.1 mnesia_frag_hash callback behavior

1.11 Appendix D: The Fragmented Table Hashing Call Back Interface

```
-module(mnesia_frag_hash).

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

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

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

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

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

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

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

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

del_frag(#hash_state{next_n_to_split = SplitN, n_doubles = L, n_fragments = N} = State) ->
    P = SplitN - 1,
    if
 P < 1 ->
     L2 = L - 1,
     MergeN = power2(L2),
     State2 = State#hash_state{n_fragments     = N - 1,
```

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

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

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

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

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

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

2 Reference Manual

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.

mnesia

Erlang 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, QLC (as an add-on library).
- 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 QLC's man page.

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', 'ordered_set' or 'bag' semantics. Note: currently 'ordered_set' is not supported for 'disc_only_copies'. 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 are local to each node, `local_content` tables may be used. The name of the table is known to all Mnesia nodes, but its contents are 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`.

- `majority` This attribute can be either `true` or `false` (default is `false`). When `true`, a majority of the table replicas must be available for an update to succeed. Majority checking can be enabled on tables with mission-critical data, where it is vital to avoid inconsistencies due to network splits.
- `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.

Writing or deleting in transaction context creates a local copy of each modified record during the transaction. During iteration, i.e. `mnesia:fold[lr]/4 mnesia:next/2 mnesia:prev/2 mnesia:snmp_get_next_index/2`, mnesia will compensate for every written or deleted record, which may reduce the performance. If possible avoid writing or deleting records in the same transaction before iterating over the table.

Exports

`abort(Reason) -> transaction abort`

Makes the transaction silently return the tuple `{aborted, Reason}`. 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.

`sync_transaction`

Short for `{sync_transaction, infinity}`

`{sync_transaction, Retries}`

Invokes `mnesia:sync_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:sync_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:lock/2 (read_lock_table/1, write_lock_table/1)
- mnesia:write/3 (write/1, s_write/1)
- mnesia:delete/3 (delete/1, s_delete/1)
- mnesia:delete_object/3 (delete_object/1, s_delete_object/1)
- mnesia:read/3 (read/1, wread/1)
- mnesia:match_object/3 (match_object/1)
- mnesia:all_keys/1
- mnesia:first/1
- mnesia:last/1
- mnesia:prev/2
- mnesia:next/2
- mnesia:index_match_object/4 (index_match_object/2)
- mnesia:index_read/3
- mnesia:table_info/2

to the corresponding:

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

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.

Note: Calling (nesting) a `mnesia:[a]sync_dirty` inside a transaction context will inherit the transaction semantics.

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

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

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

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

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

The `Config` should be an atom of the following configuration parameters:

`extra_db_nodes`

> `Value` is a list of nodes which Mnesia should try to connect to. The `ReturnValue` will be those nodes in `Value` that Mnesia are connected to.
> Note: This function shall only be used to connect to newly started ram nodes (N.D.R.S.N.) with an empty schema. If for example it is used after the network have been partitioned it may lead to inconsistent tables.
> Note: Mnesia may be connected to other nodes than those returned in `ReturnValue`.

`dc_dump_limit`

> `Value` is a number. See description in `Configuration Parameters` below. The `ReturnValue` is the new value. Note this configuration parameter is not persistent, it will be lost when mnesia stopped.

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

The `AcccessMode` 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.

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

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

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

Deletes all entries in the table `Tab`.

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

Creates a new database on disc. Various files are created in the local Mnesia directory of each node. 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 `qlc` 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.
- `{majority, Flag}`, where `Flag` must be a boolean. If `true`, any (non-dirty) update to the table will abort unless a majority of the table's replicas are available for the commit. When used on a fragmented table, all fragments will be given the same majority setting.
- `{ram_copies, Nodelist}`, where `Nodelist` is a list of the nodes where this table is supposed to have RAM copies. A table replica of type `ram_copies` is 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.
- `{storage_properties, [{Backend, Properties}]}`. Forwards additional properties to the backend storage. `Backend` can currently be `ets` or `dets` and `Properties` is a list of options sent to the backend storage during table creation. `Properties` may not contain properties already used by mnesia such as `type` or `named_table`.

  For example:

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

- `{type, Type}`, where `Type` must be either of the atoms `set`, `ordered_set` or `bag`. 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. Note: currently 'ordered_set' is not supported for 'disc_only_copies'.
- `{local_content, Bool}`, where `Bool` must be either `true` or `false`. The default value is `false`.

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, [4]}` 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 mnesia node will be removed. Note: Mnesia must be stopped on the node first.

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

If 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/3` 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 `set` or `bag` 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_last(Tab) -> Key | exit({aborted, Reason})

This function works exactly like `mnesia:dirty_first/1` but returns the last object in Erlang term order for the `ordered_set` table type. For all other table types, `mnesia:dirty_first/1` and `mnesia:dirty_last/1` are synonyms.

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

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

This function works exactly like `mnesia:dirty_next/2` but returns the previous object in Erlang term order for the ordered_set table type. For all other table types, `mnesia:dirty_next/2` and `mnesia:dirty_prev/2` are synonyms.

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

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_select(Tab, MatchSpec) -> ValueList | exit({aborted, Reason}

This is the dirty equivalent of the `mnesia:select/2` 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.

If `Key` don't exits, a new record is created with the value `Incr` if it is larger than 0, otherwise it is set to 0.

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. See configuration parameters *dump_log_time_threshold* and *dump_log_write_threshold*.

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 cannot 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.

Note: Calling (nesting) a mnesia:ets inside a transaction context will inherit the transaction semantics.

first(Tab) -> Key | transaction abort

Records in set or bag tables are not ordered. However, there is an ordering of the records 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: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.

foldl(Function, Acc, Table) -> NewAcc | transaction abort

Iterates over the table Table and calls Function(Record, NewAcc) for each Record in the table. The term returned from Function will be used as the second argument in the next call to the Function.

foldl returns the same term as the last call to Function returned.

foldr(Function, Acc, Table) -> NewAcc | transaction abort

This function works exactly like foldl/3 but iterates the table in the opposite order for the ordered_set table type. For all other table types, foldr/3 and foldl/3 are synonyms.

force_load_table(Tab) -> yes | ErrorDescription

The Mnesia algorithm for table load 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 `qlc` list comprehensions and also when using the low level `mnesia:[dirty_]match_object` functions.

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.

`is_transaction() -> boolean`

When this function is executed inside a transaction context it returns `true`, otherwise `false`.

`last(Tab) -> Key | transaction abort`

This function works exactly like `mnesia:first/1` but returns the last object in Erlang term order for the `ordered_set` table type. For all other table types, `mnesia:first/1` and `mnesia:last/1` are synonyms.

`load_textfile(Filename)`

Loads a series of definitions and data found in the text file (generated with `mnesia:dump_to_textfile/1`) into Mnesia. This function also starts Mnesia and possibly creates a new schema. This function is intended for educational purposes only and using other functions to deal with real backups, is recommended.

`lock(LockItem, LockKind) -> Nodes | ok | transaction abort`

Write locks are normally acquired on all nodes where a replica of the table resides (and is active). Read locks are acquired on one node (the local node if a local replica exists). Most of the context sensitive access functions acquire an implicit lock if they are invoked in a transaction context. The granularity of a lock may either be a single record or an entire table.

The normal usage is to call the function without checking the return value since it exits if it fails and the transaction is restarted by the transaction manager. It returns all the locked nodes if a write lock is acquired, and `ok` if it was a read lock.

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:

`sticky_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.

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.

next(Tab, Key) -> Key | transaction abort

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.

prev(Tab, Key) -> Key | transaction abort

This function works exactly like `mnesia:next/2` but returns the previous object in Erlang term order for the ordered_set table type. For all other table types, `mnesia:next/2` and `mnesia:prev/2` are synonyms.

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

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

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`, `write` and `sticky_write` are supported.

If the user wants to update the record it is more efficient to use `write/sticky_write` as the LockKind. If majority checking is active on the table, it will be checked as soon as a write lock is attempted. This can be used to quickly abort if the majority condition isn't met.

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.

select(Tab, MatchSpec [, Lock]) -> transaction abort | [Object]

Matches the objects in the table `Tab` using a match_spec as described in the ERTS Users Guide. Optionally a lock `read` or `write` can be given as the third argument, default is `read`. The return value depends on the `MatchSpec`.

Note: for best performance `select` should be used before any modifying operations are done on that table in the same transaction, i.e. don't use `write` or `delete` before a `select`.

In its simplest forms the match_spec's look like this:

- MatchSpec = [MatchFunction]
- MatchFunction = {MatchHead, [Guard], [Result]}
- MatchHead = tuple() | record()
- Guard = {"Guardtest name", ...}
- Result = "Term construct"

See the ERTS Users Guide and `ets` documentation for a complete description of the select.

For example to find the names of all male persons with an age over 30 in table Tab do:

```
MatchHead = #person{name='$1', sex=male, age='$2', _='_'},
Guard = {'>', '$2', 30},
Result = '$1',
mnesia:select(Tab,[{MatchHead, [Guard], [Result]}]),
```

select(Tab, MatchSpec, NObjects, Lock) -> transaction abort | {[Object],Cont} | '$end_of_table'

Matches the objects in the table `Tab` using a match_spec as described in ERTS users guide, and returns a chunk of terms and a continuation, the wanted number of returned terms is specified by the `NObjects` argument. The lock argument can be `read` or `write`. The continuation should be used as argument to `mnesia:select/1`, if more or all answers are needed.

Note: for best performance `select` should be used before any modifying operations are done on that table in the same transaction, i.e. don't use `mnesia:write` or `mnesia:delete` before a `mnesia:select`. For efficiency the `NObjects` is a recommendation only and the result may contain anything from an empty list to all available results.

select(Cont) -> transaction abort | {[Object],Cont} | '$end_of_table'

Selects more objects with the match specification initiated by `mnesia:select/4`.

Note: Any modifying operations, i.e. `mnesia:write` or `mnesia:delete`, that are done between the `mnesia:select/4` and `mnesia:select/1` calls will not be visible in the result.

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(logN). 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) -> {ok, Node} | {error, Reason}`

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 at *Mnesia Event Handling*.

`Node` is the local node. For table events to be subscribed, mnesia must have a readable local copy of the table on the node.

`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.

sync_log() -> ok | {error, Reason}

Ensures that the local transaction log file is synced to disk. On a single node system data written to disk tables, since the last dump, can be lost in case of a power outage. See *dump_log/0*.

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

This function waits until data have been committed and logged to disk (if disk is used) on every involved node before it returns, otherwise it behaves as `mnesia:transaction/[1,2,3]`.

This functionality can be used to avoid that one process may overload a database on another node.

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

Returns information about the Mnesia system, such as transaction statistics, db_nodes, and configuration parameters. 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.
- `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(Tab [,[Option]]) -> QueryHandle`

Returns a QLC (Query List Comprehension) query handle, see *qlc(3)*.The module `qlc` implements a query language, it can use mnesia tables as sources of data. Calling `mnesia:table/1,2` is the means to make the `mnesia` table `Tab` usable to QLC.

The list of Options may contain mnesia options or QLC options, the following options are recognized by Mnesia: `{traverse, SelectMethod},{lock, Lock},{n_objects,Number}`, any other option is forwarded to QLC. The `lock` option may be `read` or `write`, default is `read`. The option `n_objects` specifies (roughly) the number of objects returned from mnesia to QLC. Queries to remote tables may need a larger chunks to reduce network overhead, default `100` objects at a time are returned. The option `traverse` determines the method to traverse the whole table (if needed), the default method is `select`:

- `select`. The table is traversed by calling `mnesia:select/4` and `mnesia:select/1`. The match specification (the second argument of `select/3`) is assembled by QLC: simple filters are translated into equivalent match specifications while more complicated filters have to be applied to all objects returned by `select/3` given a match specification that matches all objects.
- `{select, MatchSpec}`. As for `select` the table is traversed by calling `mnesia:select/3` and `mnesia:select/1`. The difference is that the match specification is explicitly given. This is how to state match specifications that cannot easily be expressed within the syntax provided by QLC.

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

The `table_info/2` function takes two arguments. The first is the name of a Mnesia table, the second is one of the following keys:

- `all`. 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`, `set` or `ordered_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, NewRecordName) -> {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. The `Fun` argument can also be the atom `ignore`, it indicates that only the meta data about the table will be updated. Usage of `ignore` is not recommended but included as a possibility for the user to do his own transform. `NewAttributeList` and `NewRecordName` specifies the attributes and the new record type of converted table. Table name will always remain unchanged, if the record_name is changed only the mnesia functions which uses table identifiers will work, e.g. `mnesia:write/3` will work but `mnesia:write/1` will not.

transform_table(Tab, Fun, NewAttributeList) -> {aborted, R} | {atomic, ok}

Invokes `mnesia:transform_table(Tab, Fun, NewAttributeList, RecName)` where `RecName` is `mnesia:table_info(Tab, record_name)`.

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

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-install a fallback before it has been used to restore the database. This is normally a distributed operation that is either performed on all nodes with disc resident schema or none. Uninstallation of fallbacks requires Erlang to be 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) -> {ok, Node} | {error, Reason}

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

`Node` is the local node.

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

Some applications need to wait for certain tables to be accessible 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`

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

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

Invoke `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 core_dir Directory`. The name of the directory where Mnesia core files is stored or false. Setting it implies that also ram only nodes, will generate a core file if a crash occurs.
- `-mnesia dc_dump_limit Number`. Controls how often `disc_copies` tables are dumped from memory. Tables are dumped when `filesize(Log) > (filesize(Tab)/Dc_dump_limit)`. Lower values reduces cpu overhead but increases disk space and startup times. The default is 4.
- `-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 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 fallback_error_function {UserModule, UserFunc}` specifies a user supplied callback function which will be called if a fallback is installed and mnesia goes down on another node. Mnesia will call the function with one argument the name of the dying node, e.g. `UserModule:UserFunc(DyingNode)`. Mnesia should be restarted or else the database could be inconsistent. The default behaviour is to terminate mnesia.
- `-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 no_table_loaders NUMBER` specifies the number of parallel table loaders during start. More loaders can be good if the network latency is high or if many tables contains few records. The default value is `2`.
- `-mnesia send_compressed Level` specifies the level of compression to be used when copying a table from the local node to another one. The default level is 0.

  `Level` must be an integer in the interval [0, 9], with 0 representing no compression and 9 representing maximum compression. Before setting it to a non-zero value, make sure the remote nodes understand this configuration.
- `-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), qlc(3), dets(3), ets(3), disk_log(3), application(3)

mnesia_frag_hash

Erlang module

The module `mnesia_frag_hash` defines a callback behaviour for user defined hash functions of fragmented tables.

Which module that is selected to implement the `mnesia_frag_hash` behaviour for a particular fragmented table is specified together with the other `frag_properties`. The `hash_module` defines the module name. The `hash_state` defines the initial hash state.

It implements dynamic hashing which is a kind of hashing that grows nicely when new fragments are added. It is well suited for scalable hash tables

Exports

`init_state(Tab, State) -> NewState | abort(Reason)`

Types:

```
Tab = atom()
State = term()
NewState = term()
Reason = term()
```

This function is invoked when a fragmented table is created with `mnesia:create_table/2` or when a normal (un-fragmented) table is converted to be a fragmented table with `mnesia:change_table_frag/2`.

Note that the `add_frag/2` function will be invoked one time each for the rest of the fragments (all but number 1) as a part of the table creation procedure.

`State` is the initial value of the `hash_state` `frag_property`. The `NewState` will be stored as `hash_state` among the other `frag_properties`.

`add_frag(State) -> {NewState, IterFrags, AdditionalLockFrags} | abort(Reason)`

Types:

```
State = term()
NewState = term()
IterFrags = [integer()]
AdditionalLockFrags = [integer()]
Reason = term()
```

In order to scale well, it is a good idea ensure that the records are evenly distributed over all fragments including the new one.

The `NewState` will be stored as `hash_state` among the other `frag_properties`.

As a part of the `add_frag` procedure, Mnesia will iterate over all fragments corresponding to the `IterFrags` numbers and invoke `key_to_frag_number(NewState,RecordKey)` for each record. If the new fragment differs from the old fragment, the record will be moved to the new fragment.

As the `add_frag` procedure is a part of a schema transaction Mnesia will acquire a write locks on the affected tables. That is both the fragments corresponding to `IterFrags` and those corresponding to `AdditionalLockFrags`.

`del_frag(State) -> {NewState, IterFrags, AdditionalLockFrags} | abort(Reason)`

Types:

```
State = term()
NewState = term()
IterFrags = [integer()]
AdditionalLockFrags = [integer()]
Reason = term()
```

The `NewState` will be stored as `hash_state` among the other `frag_properties`.

As a part of the `del_frag` procedure, Mnesia will iterate over all fragments corresponding to the `IterFrags` numbers and invoke `key_to_frag_number(NewState,RecordKey)` for each record. If the new fragment differs from the old fragment, the record will be moved to the new fragment.

Note that all records in the last fragment must be moved to another fragment as the entire fragment will be deleted.

As the `del_frag` procedure is a part of a schema transaction Mnesia will acquire a write locks on the affected tables. That is both the fragments corresponding to `IterFrags` and those corresponding to `AdditionalLockFrags`.

`key_to_frag_number(State, Key) -> FragNum | abort(Reason)`

Types:

```
FragNum = integer()()
Reason = term()
```

This function is invoked whenever Mnesia needs to determine which fragment a certain record belongs to. It is typically invoked at read, write and delete.

`match_spec_to_frag_numbers(State, MatchSpec) -> FragNums | abort(Reason)`

Types:

```
MatcSpec = ets_select_match_spec()
FragNums = [FragNum]
FragNum = integer()
Reason = term()
```

This function is invoked whenever Mnesia needs to determine which fragments that needs to be searched for a MatchSpec. It is typically invoked at select and match_object.

See Also

mnesia(3)

mnesia_registry

Erlang 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)