NAME

mnesia − A distributed telecommunications DBMS

DESCRIPTION

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

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

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

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

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

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

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

  If a new item is inserted with the same key as an existing record, the old record is overwritten. However, if a table is of type *bag*, each key can map to several records. All records in type *bag* tables are unique, only the keys can be duplicated.

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

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

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

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

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

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

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

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

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

EXPORTS

abort(Reason) -> transaction abort

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

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

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

Args is a list of the following tuples:

* *{name,Name}*. *Name* is the checkpoint name. Each checkpoint must have a name that is unique to the associated nodes. The name can be reused only once the checkpoint has been deactivated. By default, a name that is probably unique is generated.
* *{max,MaxTabs}*. *MaxTabs* is a list of tables that are to be included in the checkpoint. Default is *[]*. For these tables, the redundancy is maximized and checkpoint information is retained together with all replicas. The checkpoint becomes more fault tolerant if the tables have several replicas. When a new replica is added by the schema manipulation function *mnesia:add_table_copy/3*, a retainer is also attached automatically.
* *{min,MinTabs}*. *MinTabs* is a list of tables that are to be included in the checkpoint. Default is []. For these tables, the redundancy is minimized and the checkpoint information is only retained with one replica, preferably on the local node.
* *{allow_remote,Bool}*. *false* means that all retainers must be local. The checkpoint cannot be activated if a table does not reside locally. *true* allows retainers to be allocated on any node. Default is *true*.
* *{ram_overrides_dump,Bool}*. Only applicable for *ram_copies*. *Bool* allows you to choose to back up the table state as it is in RAM, or as it is on disc. *true* means that the latest committed records in RAM are to be included in the checkpoint. These are the records that the application accesses. *false* means that the records dumped to *DAT* files are to be included in the checkpoint. These records are loaded at startup. Default is *false*.

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

name. *Nodes* are the nodes that are involved in the checkpoint. Only nodes that keep a checkpoint retainer know about the checkpoint.

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

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

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

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

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

transaction:
Short for *{transaction, infinity}*

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

sync_transaction:
Short for *{sync_transaction, infinity}*

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

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

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

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

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

Mnesia forwards calls to the following functions:

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

* mnesia:index_read/3
* mnesia:table_info/2

to the corresponding:

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

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

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

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

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

mnesia:add_table_copy(person, Node, disc_copies)

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

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

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

mnesia:add_table_index(person, age)

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

all_keys(Tab) -> KeyList | transaction abort

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

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

Calls the *Fun* in a context that is not protected by a transaction. The Mnesia function calls

performed in the *Fun* are mapped to the corresponding dirty functions. This still involves logging, replication, and subscriptions, but there is no locking, local transaction storage, or commit protocols involved. Checkpoint retainers and indexes are updated, but they are updated dirty. As for normal *mnesia:dirty_** operations, the operations are performed semi-asynchronously. For details, see *mnesia:activity/4* and the User's Guide.

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

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

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

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

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

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

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

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

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

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

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

extra_db_nodes:
Value is a list of nodes that Mnesia is to try to connect to. *ReturnValue* is those nodes in *Value* that Mnesia is connected to.

Notice that this function must only be used to connect to newly started RAM nodes (N.D.R.S.N.) with an empty schema. If, for example, this function is used after the network has been partitioned, it can lead to inconsistent tables.

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

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

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

AcccessMode is by default the atom *read_write* but it can also be set to the atom *read_only*. If

AccessMode is set to *read_only*, updates to the table cannot be performed. At startup, Mnesia always loads *read_only* tables locally regardless of when and if Mnesia is terminated on other nodes.

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

For example:

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

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

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

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

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

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

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

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

Deletes all entries in the table *Tab*.

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

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

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

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

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

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

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

  At startup, Mnesia always loads *read_only* table locally regardless of when and if Mnesia is terminated on other nodes. This argument returns the access mode of the table. The access mode can be *read_only* or *read_write*.

* *{attributes, AtomList}* is a list of the attribute names for the records that are supposed to populate the table. Default is *[key, val]*. The table must at least have one extra attribute in addition to the key.

  When accessing single attributes in a record, it is not necessary, or even recommended, to hard code any attribute names as atoms. Use construct *record_info(fields, RecordName)* instead. It can be used for records of type *RecordName*.

* *{disc_copies, Nodelist}*, where *Nodelist* is a list of the nodes where this table is supposed to have disc copies. If a table replica is of type *disc_copies*, all write operations on this particular replica of the table are written to disc and to the RAM copy of the table.

  It is possible to have a replicated table of type *disc_copies* on one node and another type on another node. Default is *[]*.

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

  For example:

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

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

  Notice that currently *ordered_set* is not supported for *disc_only_copies*.

* *{local_content, Bool}*, where *Bool* is *true* or *false*. Default is *false*.

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

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

If it is required that Mnesia must build and maintain an extra index table on attribute *address*

of all the *person* records that are inserted in the table, the following code would be issued:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

delete_object(Record) -> transaction abort | ok

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

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

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

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

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

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

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

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

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

Permanently deletes all replicas of table *Tab*.

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

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

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

Calls *mnesia:dirty_delete(Tab, Key)*.

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

Dirty equivalent of the function *mnesia:delete/3*.

dirty_delete_object(Record)

Calls *mnesia:dirty_delete_object(Tab, Record)*, where *Tab* is *element(1, Record)*.

dirty_delete_object(Tab, Record)

Dirty equivalent of the function *mnesia:delete_object/3*.

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

Records in *set* or *bag* tables are not ordered. However, there is an ordering of the records that is unknown to the user. Therefore, a table can be traversed by this function with the function *mnesia:dirty_next/2*.

If there are no records in the table, this function returns the atom *'$end_of_table'*. It is therefore highly undesirable, but not disallowed, to use this atom as the key for any user records.

dirty_index_match_object(Pattern, Pos)

Starts *mnesia:dirty_index_match_object(Tab, Pattern, Pos)*, where *Tab* is *element(1, Pattern)*.

dirty_index_match_object(Tab, Pattern, Pos)

Dirty equivalent of the function *mnesia:index_match_object/4*.

dirty_index_read(Tab, SecondaryKey, Pos)

Dirty equivalent of the function *mnesia:index_read/3*.

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

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

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

Calls *mnesia:dirty_match_object(Tab, Pattern)*, where *Tab* is *element(1, Pattern)*.

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

Dirty equivalent of the function *mnesia:match_object/3*.

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

Traverses a table and performs operations on all records in the table. When the end of the table is reached, the special key *'$end_of_table'* is returned. Otherwise, the function returns a key that can be used to read the actual record. The behavior is undefined if another Erlang process performs write operations on the table while it is being traversed with the function *mnesia:dirty_next/2*.

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

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

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

Calls *mnesia:dirty_read(Tab, Key)*.

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

Dirty equivalent of the function *mnesia:read/3*.

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

Dirty equivalent of the function *mnesia:select/2*.

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

Traverses a table in a manner similar to the function *mnesia:dirty_next/2*. A table has a number of slots that range from 0 (zero) to an unknown upper bound. The function *mnesia:dirty_slot/2* returns the special atom *'$end_of_table'* when the end of the table is reached. The behavior of this function is undefined if a write operation is performed on the table while it is being traversed.

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

Calls *mnesia:dirty_update_counter(Tab, Key, Incr)*.

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

Mnesia has no special counter records. However, records of the form *{Tab, Key, Integer}* can be used as (possibly disc-resident) counters when *Tab* is a *set*. This function updates a counter with a positive or negative number. However, counters can never become less than zero. There are two significant differences between this function and the action of first reading the record, performing the arithmetics, and then writing the record:

* It is much more efficient.
* *mnesia:dirty_update_counter/3* is performed as an atomic operation although it is not protected by a transaction.

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

If *Key* do not exists, a new record is created with value *Incr* if it is larger than 0, otherwise it is set to 0.

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

Calls *mnesia:dirty_write(Tab, Record)*, where *Tab* is *element(1, Record)*.

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

Dirty equivalent of the function *mnesia:write/3*.

dump_log() -> dumped

Performs a user-initiated dump of the local log file. This is usually not necessary, as Mnesia by default manages this automatically. See configuration parameters **dump_log_time_threshold** and **dump_log_write_threshold**.

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

Dumps a set of *ram_copies* tables to disc. The next time the system is started, these tables are initiated with the data found in the files that are the result of this dump. None of the tables can have disc-resident replicas.

dump_to_textfile(Filename)

Dumps all local tables of a Mnesia system into a text file, which can be edited (by a normal text editor) and then be reloaded with *mnesia:load_textfile/1*. Only use this function for educational purposes. Use other functions to deal with real backups.

error_description(Error) -> String

All Mnesia transactions, including all the schema update functions, either return value *{atomic, Val}* or the tuple *{aborted, Reason}*. *Reason* can be either of the atoms in the following list. The function *error_description/1* returns a descriptive string that describes the error.

* *nested_transaction*. Nested transactions are not allowed in this context.
* *badarg*. Bad or invalid argument, possibly bad type.
* *no_transaction*. Operation not allowed outside transactions.
* *combine_error*. Table options illegally combined.
* *bad_index*. Index already exists, or was out of bounds.
* *already_exists*. Schema option to be activated is already on.
* *index_exists*. Some operations cannot be performed on tables with an index.
* *no_exists*. Tried to perform operation on non-existing (not-alive) item.
* *system_limit*. A system limit was exhausted.
* *mnesia_down*. A transaction involves records on a remote node, which became unavailable before the transaction was completed. Records are no longer available elsewhere in the network.
* *not_a_db_node*. A node was mentioned that does not exist in the schema.
* *bad_type*. Bad type specified in argument.
* *node_not_running*. Node is not running.
* *truncated_binary_file*. Truncated binary in file.
* *active*. Some delete operations require that all active records are removed.
* *illegal*. Operation not supported on this record.

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

The following examples illustrate a function that returns an error, and the method to retrieve more detailed error information:

* The function **mnesia:create_table(bar, [{attributes, 3.14}])** returns the tuple *{aborted,Reason}*, where *Reason* is the tuple *{bad_type,bar,3.14000}*.
* The function **mnesia:error_description(Reason)** returns the term *{"Bad type on some provided arguments",bar,3.14000}*, which is an error description suitable for display.

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

Calls the *Fun* in a raw context that is not protected by a transaction. The Mnesia function call is performed in the *Fun* and performed directly on the local ETS tables on the assumption that the local storage type is *ram_copies* and the tables are not replicated to other nodes. Subscriptions are not triggered and checkpoints are not updated, but it is extremely fast. This function can also be applied to *disc_copies* tables if all operations are read only. For details, see *mnesia:activity/4* and the User's Guide.

Notice that calling (nesting) a *mnesia:ets* inside a transaction-context inherits the transaction semantics.

first(Tab) -> Key | transaction abort

Records in *set* or *bag* tables are not ordered. However, there is an ordering of the records that is unknown to the user. A table can therefore be traversed by this function with the function *mnesia:next/2*.

If there are no records in the table, this function returns the atom *'$end_of_table'*. It is therefore highly undesirable, but not disallowed, to use this atom as the key for any user records.

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

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

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

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

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

force_load_table(Tab) -> yes | ErrorDescription

The Mnesia algorithm for table load can lead to a situation where a table cannot be loaded. This situation occurs when a node is started and Mnesia concludes, or suspects, that another copy of the table was active after this local copy became inactive because of a system crash.

If this situation is not acceptable, this function can be used to override the strategy of the Mnesia table load algorithm. This can lead to a situation where some transaction effects are lost with an inconsistent database as result, but for some applications high availability is more important than consistent data.

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

Starts *mnesia:index_match_object(Tab, Pattern, Pos, read)*, where *Tab* is *element(1, Pattern)*.

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

In a manner similar to the function *mnesia:index_read/3*, any index information can be used when trying to match records. This function takes a pattern that obeys the same rules as the function *mnesia:match_object/3*, except that this function requires the following conditions:

* The table *Tab* must have an index on position *Pos*.
* The element in position *Pos* in *Pattern* must be bound. *Pos* is an integer (*#record.Field*) or an attribute name.

The two index search functions described here are automatically started when searching tables

with *qlc* list comprehensions and also when using the low-level *mnesia:[dirty_]match_object* functions.

The semantics of this function is context-sensitive. For details, see *mnesia:activity/4*. In transaction-context, it acquires a lock of type *LockKind* on the entire table or on a single record. Currently, the lock type *read* is supported.

index_read(Tab, SecondaryKey, Pos) -> transaction abort | RecordList

Assume that there is an index on position *Pos* for a certain record type. This function can be used to read the records without knowing the actual key for the record. For example, with an index in position 1 of table *person*, the call *mnesia:index_read(person, 36, #person.age)* returns a list of all persons with age 36. *Pos* can also be an attribute name (atom), but if the notation *mnesia:index_read(person, 36, age)* is used, the field position is searched for in runtime, for each call.

The semantics of this function is context-sensitive. For details, see *mnesia:activity/4*. In transaction-context, it acquires a read lock on the entire table.

info() -> ok

Prints system information on the terminal. This function can be used even if Mnesia is not started. However, more information is displayed if Mnesia is started.

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

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

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

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

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

Installs a backup as fallback. The fallback is used to restore the database at the next startup. Installation of fallbacks requires Erlang to be operational on all the involved nodes, but it does not matter if Mnesia is running or not. The installation of the fallback fails if the local node is not one of the disc-resident nodes in the backup.

Args is a list of the following tuples:

* *{module, BackupMod}*. All accesses of the backup media are performed through a callback module named *BackupMod*. Argument *Opaque* is forwarded to the callback module, which can interpret it as it wishes. The default callback module is called *mnesia_backup* and it interprets argument *Opaque* as a local filename. The default for this module is also configurable through configuration parameter *-mnesia mnesia_backup*.

* *{scope, Scope}*. The *Scope* of a fallback is either *global* for the entire database or *local* for one node. By default, the installation of a fallback is a global operation, which either is performed on all nodes with a disc-resident schema or none. Which nodes that are disc-resident is determined from the schema information in the backup.

  If *Scope* of the operation is *local*, the fallback is only installed on the local node.

* *{mnesia_dir, AlternateDir}*. This argument is only valid if the scope of the installation is *local*. Normally the installation of a fallback is targeted to the Mnesia directory, as configured with configuration parameter *-mnesia dir*. But by explicitly supplying an *AlternateDir*, the fallback is installed there regardless of the Mnesia directory configuration parameter setting. After installation of a fallback on an alternative Mnesia directory, that directory is fully prepared for use as an active Mnesia directory.

  This is a dangerous feature that must be used with care. By unintentional mixing of

directories, you can easily end up with an inconsistent database, if the same backup is installed on more than one directory.

is_transaction() -> boolean

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

last(Tab) -> Key | transaction abort

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

load_textfile(Filename)

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

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

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

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

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

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

read:
Read locks can be shared. This means that if one transaction manages to acquire a read lock on an item, other transactions can also acquire a read lock on the same item. However, if someone has a read lock, no one can acquire a write lock at the same item. If someone has a write lock, no one can acquire either a read lock or a write lock at the same item.

Conflicting lock requests are automatically queued if there is no risk of a deadlock. Otherwise the transaction must be terminated and executed again. Mnesia does this automatically as long as the upper limit of the maximum *retries* is not reached. For details, see *mnesia:transaction/3*.

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

sticky_write:
Sticky write locks are a mechanism that can be used to optimize write lock acquisition. If your application uses replicated tables mainly for fault tolerance (as opposed to read access optimization purpose), sticky locks can be the best option available.

When a sticky write lock is acquired, all nodes are informed which node is locked. Then, sticky lock requests from the same node are performed as a local operation without any communication with other nodes. The sticky lock lingers on the node even after the transaction ends. For details, see the User's Guide.

Currently, this function supports two kinds of *LockItem*:

{table, Tab}:
This acquires a lock of type *LockKind* on the entire table *Tab*.

{global, GlobalKey, Nodes}:
This acquires a lock of type *LockKind* on the global resource *GlobalKey*. The lock is acquired on all active nodes in the *Nodes* list.

Locks are released when the outermost transaction ends.

The semantics of this function is context-sensitive. For details, see *mnesia:activity/4*. In transaction-context, it acquires locks, otherwise it ignores the request.

match_object(Pattern) -> transaction abort | RecList

Calls *mnesia:match_object(Tab, Pattern, read)*, where *Tab* is *element(1, Pattern)*.

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

Takes a pattern with "don't care" variables denoted as a '_' parameter. This function returns a list of records that matched the pattern. Since the second element of a record in a table is considered to be the key for the record, the performance of this function depends on whether this key is bound or not.

For example, the call *mnesia:match_object(person, {person, '_', 36, '_', '_'}, read)* returns a list of all person records with an *age* field of 36.

The function *mnesia:match_object/3* automatically uses indexes if these exist. However, no heuristics are performed to select the best index.

The semantics of this function is context-sensitive. For details, see *mnesia:activity/4*. In transaction-context, it acquires a lock of type *LockKind* on the entire table or a single record. Currently, the lock type *read* is supported.

move_table_copy(Tab, From, To) -> {aborted, Reason} | {atomic, ok}

Moves the copy of table *Tab* from node *From* to node *To*.

The storage type is preserved. For example, a RAM table moved from one node remains a RAM on the new node. Other transactions can still read and write in the table while it is being moved.

This function cannot be used on *local_content* tables.

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

Traverses a table and performs operations on all records in the table. When the end of the table is reached, the special key *'$end_of_table'* is returned. Otherwise the function returns a key that can be used to read the actual record.

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

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

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

Calls function *mnesia:read(Tab, Key, read)*.

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

Calls function *mnesia:read(Tab, Key, read)*.

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

Reads all records from table *Tab* with key *Key*. This function has the same semantics regardless of the location of *Tab*. If the table is of type *bag*, the function *mnesia:read(Tab, Key)* can return an arbitrarily long list. If the table is of type *set*, the list is either of length 1, or *[]*.

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

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

read_lock_table(Tab) -> ok | transaction abort

Calls the function *mnesia:lock({table, Tab}, read).*

report_event(Event) -> ok

When tracing a system of Mnesia applications it is useful to be able to interleave Mnesia own events with application-related events that give information about the application context.

Whenever the application begins a new and demanding Mnesia task, or if it enters a new interesting phase in its execution, it can be a good idea to use *mnesia:report_event/1*. *Event* can be any term and generates a *{mnesia_user, Event}* event for any processes that subscribe to Mnesia system events.

restore(Opaque, Args) -> {atomic, RestoredTabs} |{aborted, Reason}

With this function, tables can be restored online from a backup without restarting Mnesia. *Opaque* is forwarded to the backup module. *Args* is a list of the following tuples:

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

The affected tables are write-locked during the restoration. However, regardless of the lock conflicts caused by this, the applications can continue to do their work while the restoration is being performed. The restoration is performed as one single transaction.

If the database is huge, it it not always possible to restore it online. In such cases, restore the old database by installing a fallback and then restart.

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

Calls the function *mnesia:delete(Tab, Key, sticky_write)*

s_delete_object(Record) -> ok | transaction abort

Calls the function *mnesia:delete_object(Tab, Record, sticky_write)*, where *Tab* is *element(1, Record)*.

s_write(Record) -> ok | transaction abort

Calls the function *mnesia:write(Tab, Record, sticky_write)*, where *Tab* is *element(1, Record)*.

schema() -> ok

Prints information about all table definitions on the terminal.

schema(Tab) -> ok

Prints information about one table definition on the terminal.

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

Matches the objects in table *Tab* using a *match_spec* as described in the **ets:select/3**. Optionally a lock *read* or *write* can be given as the third argument. Default is *read*. The return value depends on *MatchSpec*.

Notice that for best performance, *select* is to be used before any modifying operations are done on that table in the same transaction. That is, do not use *write* or *delete* before a *select*.

In its simplest forms, the *match_spec* look as follows:

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

For a complete description of *select*, see the **ERTS** User's Guide and the **ets** manual page in STDLIB.

For example, to find the names of all male persons older than 30 in table *Tab*:

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

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

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

Notice that for best performance, *select* is to be used before any modifying operations are done on that table in the same transaction. That is, do not use *mnesia:write* or *mnesia:delete* before a *mnesia:select*. For efficiency, *NObjects* is a recommendation only and the result can contain anything from an empty list to all available results.

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

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

Notice that any modifying operations, that is, *mnesia:write* or *mnesia:delete*, that are done between the *mnesia:select/4* and *mnesia:select/1* calls are not visible in the result.

set_debug_level(Level) -> OldLevel

Changes the internal debug level of Mnesia. For details, see **Section Configuration Parameters**.

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

For each table Mnesia determines its replica nodes (*TabNodes*) and starts *mnesia:set_master_nodes(Tab, TabMasterNodes)*. where *TabMasterNodes* is the intersection of *MasterNodes* and *TabNodes*. For semantics, see *mnesia:set_master_nodes/2*.

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

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

The master node setting is always local. It can be changed regardless if Mnesia is started or not.

The database can also become inconsistent if configuration parameter *max_wait_for_decision* is used or if *mnesia:force_load_table/1* is used.

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

Removes the possibility for SNMP to manipulate the table.

snmp_get_mnesia_key(Tab, RowIndex) -> {ok, Key} | undefined

Types:

```
Tab ::= atom()
RowIndex ::= [integer()]
Key ::= key() | {key(), key(), ...}
key() ::= integer() | string() | [integer()]
```

Transforms an SNMP index to the corresponding Mnesia key. If the SNMP table has multiple keys, the key is a tuple of the key columns.

snmp_get_next_index(Tab, RowIndex) -> {ok, NextIndex} | endOfTable

Types:

```
Tab ::= atom()
RowIndex ::= [integer()]
NextIndex ::= [integer()]
```

RowIndex can specify a non-existing row. Specifically, it can be the empty list. Returns the index of the next lexicographical row. If *RowIndex* is the empty list, this function returns the index of the first row in the table.

snmp_get_row(Tab, RowIndex) -> {ok, Row} | undefined

Types:

Tab ::= atom()
RowIndex ::= [integer()]
Row ::= record(Tab)

Reads a row by its SNMP index. This index is specified as an SNMP Object Identifier, a list of integers.

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

Types:

Tab ::= atom()
SnmpStruct ::= [{key, type()}]
type() ::= type_spec() | {type_spec(), type_spec(), ...}
type_spec() ::= fix_string | string | integer

A direct one-to-one mapping can be established between Mnesia tables and SNMP tables. Many telecommunication applications are controlled and monitored by the SNMP protocol. This connection between Mnesia and SNMP makes it simple and convenient to achieve this mapping.

Argument *SnmpStruct* is a list of SNMP information. Currently, the only information needed is information about the key types in the table. Multiple keys cannot be handled in Mnesia, but many SNMP tables have multiple keys. Therefore, the following convention is used: if a table has multiple keys, these must always be stored as a tuple of the keys. Information about the key types is specified as a tuple of atoms describing the types. The only significant type is *fix_string*. This means that a string has a fixed size.

For example, the following causes table *person* to be ordered as an SNMP table:

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

Consider the following schema for a table of company employees. Each employee is identified by department number and name. The other table column stores the telephone number:

```
mnesia:create_table(employee,
  [{snmp, [{key, {integer, string}}]},
   {attributes, record_info(fields, employees)}]),
```

The corresponding SNMP table would have three columns: *department*, *name*, and *telno*.

An option is to have table columns that are not visible through the SNMP protocol. These columns must be the last columns of the table. In the previous example, the SNMP table could have columns *department* and *name* only. The application could then use column *telno* internally, but it would not be visible to the SNMP managers.

In a table monitored by SNMP, all elements must be integers, strings, or lists of integers.

When a table is SNMP ordered, modifications are more expensive than usual, O(logN). Also, more memory is used.

Notice that only the lexicographical SNMP ordering is implemented in Mnesia, not the actual SNMP monitoring.

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

The startup procedure for a set of Mnesia nodes is a fairly complicated operation. A Mnesia system consists of a set of nodes, with Mnesia started locally on all participating nodes. Normally, each node has a directory where all the Mnesia files are written. This directory is referred to as the Mnesia directory. Mnesia can also be started on disc-less nodes. For more information about disc-less nodes, see *mnesia:create_schema/1* and the User's Guide.

The set of nodes that makes up a Mnesia system is kept in a schema. Mnesia nodes can be added to or removed from the schema. The initial schema is normally created on disc with the function *mnesia:create_schema/1*. On disc-less nodes, a tiny default schema is generated each time Mnesia is started. During the startup procedure, Mnesia exchanges schema information between the nodes to verify that the table definitions are compatible.

Each schema has a unique cookie, which can be regarded as a unique schema identifier. The cookie must be the same on all nodes where Mnesia is supposed to run. For details, see the User's Guide.

The schema file and all other files that Mnesia needs are kept in the Mnesia directory. The command-line option *-mnesia dir Dir* can be used to specify the location of this directory to the Mnesia system. If no such command-line option is found, the name of the directory defaults to *Mnesia.Node*.

application:start(mnesia) can also be used.

stop() -> stopped

Stops Mnesia locally on the current node.

application:stop(mnesia) can also be used.

subscribe(EventCategory) -> {ok, Node} | {error, Reason}

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

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

Calls the *Fun* in a context that is not protected by a transaction. The Mnesia function calls performed in the *Fun* are mapped to the corresponding dirty functions. It is performed in almost the same context as *mnesia:async_dirty/1,2*. The difference is that the operations are performed synchronously. The caller waits for the updates to be performed on all active replicas before the *Fun* returns. For details, see *mnesia:activity/4* and the User's Guide.

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

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

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

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

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

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

Returns information about the Mnesia system, such as transaction statistics, *db_nodes*, and configuration parameters. The valid keys are as follows:

* *all*. Returns a list of all local system information. Each element is a *{InfoKey, InfoVal}* tuple.

  New *InfoKey*s can be added and old undocumented *InfoKey*s can be removed without notice.

* *access_module*. Returns the name of module that is configured to be the activity access callback module.
* *auto_repair*. Returns *true* or *false* to indicate if Mnesia is configured to start the autorepair facility on corrupted disc files.
* *backup_module*. Returns the name of the module that is configured to be the backup callback module.
* *checkpoints*. Returns a list of the names of the checkpoints currently active on this node.

* *event_module*. Returns the name of the module that is the event handler callback module.
* *db_nodes*. Returns the nodes that make up the persistent database. Disc-less nodes are only included in the list of nodes if they explicitly have been added to the schema, for example, with *mnesia:add_table_copy/3*. The function can be started even if Mnesia is not yet running.
* *debug*. Returns the current debug level of Mnesia.
* *directory*. Returns the name of the Mnesia directory. It can be called even if Mnesia is not yet running.
* *dump_log_load_regulation*. Returns a boolean that tells if Mnesia is configured to regulate the dumper process load.

  This feature is temporary and will be removed in future releases.

* *dump_log_time_threshold*. Returns the time threshold for transaction log dumps in milliseconds.
* *dump_log_update_in_place*. Returns a boolean that tells if Mnesia is configured to perform the updates in the Dets files directly, or if the updates are to be performed in a copy of the Dets files.
* *dump_log_write_threshold*. Returns the write threshold for transaction log dumps as the number of writes to the transaction log.
* *extra_db_nodes*. Returns a list of extra *db_nodes* to be contacted at startup.
* *fallback_activated*. Returns *true* if a fallback is activated, otherwise *false*.
* *held_locks*. Returns a list of all locks held by the local Mnesia lock manager.
* *is_running*. Returns *yes* or *no* to indicate if Mnesia is running. It can also return *starting* or *stopping*. Can be called even if Mnesia is not yet running.
* *local_tables*. Returns a list of all tables that are configured to reside locally.
* *lock_queue*. Returns a list of all transactions that are queued for execution by the local lock manager.
* *log_version*. Returns the version number of the Mnesia transaction log format.
* *master_node_tables*. Returns a list of all tables with at least one master node.
* *protocol_version*. Returns the version number of the Mnesia inter-process communication protocol.
* *running_db_nodes*. Returns a list of nodes where Mnesia currently is running. This function can be called even if Mnesia is not yet running, but it then has slightly different semantics.

  If Mnesia is down on the local node, the function returns those other *db_nodes* and *extra_db_nodes* that for the moment are operational.

  If Mnesia is started, the function returns those nodes that Mnesia on the local node is fully connected to. Only those nodes that Mnesia has exchanged schema information with are included as *running_db_nodes*. After the merge of schemas, the local Mnesia system is fully operable and applications can perform access of remote replicas. Before the schema merge, Mnesia only operates locally. Sometimes there are more nodes included in the *running_db_nodes* list than all *db_nodes* and *extra_db_nodes* together.

* *schema_location*. Returns the initial schema location.
* *subscribers*. Returns a list of local processes currently subscribing to system events.
* *tables*. Returns a list of all locally known tables.
* *transactions*. Returns a list of all currently active local transactions.

* *transaction_failures*. Returns a number that indicates how many transactions have failed since Mnesia was started.
* *transaction_commits*. Returns a number that indicates how many transactions have terminated successfully since Mnesia was started.
* *transaction_restarts*. Returns a number that indicates how many transactions have been restarted since Mnesia was started.
* *transaction_log_writes*. Returns a number that indicates how many write operations that have been performed to the transaction log since startup.
* *use_dir*. Returns a boolean that indicates if the Mnesia directory is used or not. Can be started even if Mnesia is not yet running.
* *version*. Returns the current version number of Mnesia.

table(Tab [,[Option]]) -> QueryHandle

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

Option can contain Mnesia options or QLC options. Mnesia recognizes the following options (any other option is forwarded to QLC).

* *{lock, Lock}*, where *lock* can be *read* or *write*. Default is *read*.
* *{n_objects,Number}*, where *n_objects* specifies (roughly) the number of objects returned from Mnesia to QLC. Queries to remote tables can need a larger chunk to reduce network overhead. By default, *100* objects at a time are returned.
* *{traverse, SelectMethod}*, where *traverse* determines the method to traverse the whole table (if needed). The default method is *select*.

There are two alternatives for *select*:

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

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

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

* *all*. Returns a list of all local table information. Each element is a *{InfoKey, ItemVal}* tuple.

  New *InfoItem*s can be added and old undocumented *InfoItem*s can be removed without notice.

* *access_mode*. Returns the access mode of the table. The access mode can be *read_only* or *read_write*.
* *arity*. Returns the arity of records in the table as specified in the schema.
* *attributes*. Returns the table attribute names that are specified in the schema.
* *checkpoints*. Returns the names of the currently active checkpoints, which involve this table on this node.
* *cookie*. Returns a table cookie, which is a unique system-generated identifier for the table. The cookie is used internally to ensure that two different table definitions using the

same table name cannot accidentally be intermixed. The cookie is generated when the table is created initially.

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

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

Executes the functional object *Fun* with arguments *Args* as a transaction.

The code that executes inside the transaction can consist of a series of table manipulation functions. If something goes wrong inside the transaction as a result of a user error or a certain table not being available, the entire transaction is terminated and the function *transaction/1* returns the tuple *{aborted, Reason}*.

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

A function that adds a family to the database can be written as follows if there is a structure *{family, Father, Mother, ChildrenList}*:

```
add_family({family, F, M, Children}) ->
  ChildOids = lists:map(fun oid/1, Children),
  Trans = fun() ->
    mnesia:write(F#person{children = ChildOids},
    mnesia:write(M#person{children = ChildOids},
    Write = fun(Child) -> mnesia:write(Child) end,
    lists:foreach(Write, Children)
  end,
  mnesia:transaction(Trans).

oid(Rec) -> {element(1, Rec), element(2, Rec)}.
```

This code adds a set of people to the database. Running this code within one transaction ensures that either the whole family is added to the database, or the whole transaction terminates. For example, if the last child is badly formatted, or the executing process terminates because of an *'EXIT'* signal while executing the family code, the transaction terminates. Thus, the situation where half a family is added can never occur.

It is also useful to update the database within a transaction if several processes concurrently update the same records. For example, the function *raise(Name, Amount)*, which adds *Amount* to the salary field of a person, is to be implemented as follows:

```
raise(Name, Amount) ->
  mnesia:transaction(fun() ->
    case mnesia:wread({person, Name}) of
      [P] ->
        Salary = Amount + P#person.salary,
        P2 = P#person{salary = Salary},
        mnesia:write(P2);
      _ ->
        mnesia:abort("No such person")
    end
  end).
```

When this function executes within a transaction, several processes running on different nodes can concurrently execute the function *raise/2* without interfering with each other.

Since Mnesia detects deadlocks, a transaction can be restarted any number of times. This function attempts a restart as specified in *Retries*. *Retries* must be an integer greater than 0 or the atom *infinity*. Default is *infinity*.

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

Applies argument *Fun* to all records in the table. *Fun* is a function that takes a record of the old type and returns a transformed record of the new type. Argument *Fun* can also be the atom *ignore*, which indicates that only the metadata about the table is updated. Use of *ignore* is not recommended, but included as a possibility for the user do to an own transformation.

NewAttributeList and *NewRecordName* specify the attributes and the new record type of the converted table. Table name always remains unchanged. If *record_name* is changed, only the Mnesia functions that use table identifiers work, for example, *mnesia:write/3* works, but not *mnesia:write/1*.

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

Calls *mnesia:transform_table(Tab, Fun, NewAttributeList, RecName)*, where *RecName* is *mnesia:table_info(Tab, record_name)*.

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

Iterates over a backup, either to transform it into a new backup, or read it. The arguments are explained briefly here. For details, see the User's Guide.

* *SourceMod* and *TargetMod* are the names of the modules that actually access the backup media.
* *Source* and *Target* are opaque data used exclusively by modules *SourceMod* and *TargetMod* to initialize the backup media.
* *Acc* is an initial accumulator value.
* *Fun(BackupItems, Acc)* is applied to each item in the backup. The *Fun* must return a tuple *{BackupItems,NewAcc}*, where *BackupItems* is a list of valid backup items, and *NewAcc* is a new accumulator value. The returned backup items are written in the target backup.
* *LastAcc* is the last accumulator value. This is the last *NewAcc* value that was returned by *Fun*.

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

Calls the function *mnesia:uninstall_fallback([{scope, global}])*.

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

Deinstalls a fallback before it has been used to restore the database. This is normally a distributed operation that is either performed on all nodes with disc resident schema, or none. Uninstallation of fallbacks requires Erlang to be operational on all involved nodes, but it does not matter if Mnesia is running or not. Which nodes that are considered as disc-resident nodes is determined from the schema information in the local fallback.

Args is a list of the following tuples:

* *{module, BackupMod}*. For semantics, see *mnesia:install_fallback/2*.
* *{scope, Scope}*. For semantics, see *mnesia:install_fallback/2*.
* *{mnesia_dir, AlternateDir}*. For semantics, see *mnesia:install_fallback/2*.

unsubscribe(EventCategory) -> {ok, Node} | {error, Reason}

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

Node is the local node.

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

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

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

Calls the function *mnesia:read(Tab, Key, write)*.

write(Record) -> transaction abort | ok

Calls the function *mnesia:write(Tab, Record, write)*, where *Tab* is *element(1, Record)*.

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

Writes record *Record* to table *Tab*.

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

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

write_lock_table(Tab) -> ok | transaction abort

Calls the function *mnesia:lock({table, Tab}, write)*.

CONFIGURATION PARAMETERS

Mnesia reads the following application configuration parameters:

* *-mnesia access_module Module*. The name of the Mnesia activity access callback module. Default is *mnesia*.
* *-mnesia auto_repair true | false*. This flag controls if Mnesia automatically tries to repair files that have not been properly closed. Default is *true*.
* *-mnesia backup_module Module*. The name of the Mnesia backup callback module. Default is *mnesia_backup*.
* *-mnesia debug Level*. Controls the debug level of Mnesia. The possible values are as follows:

  none:
  No trace outputs. This is the default.

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

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

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

  false:
  An alias for none.

  true:
  An alias for debug.
* *-mnesia core_dir Directory*. The name of the directory where Mnesia core files is stored, or false. Setting it implies that also RAM-only nodes generate a core file if a crash occurs.
* *-mnesia dc_dump_limit Number*. Controls how often *disc_copies* tables are dumped from memory. Tables are dumped when *filesize(Log) > (filesize(Tab)/Dc_dump_limit)*. Lower values reduce CPU overhead but increase disk space and startup times. Default is 4.
* *-mnesia dir Directory*. The name of the directory where all Mnesia data is stored. The directory name must be unique for the current node. Two nodes must never share the the same Mnesia directory. The results are unpredictable.
* *-mnesia dump_disc_copies_at_startup true | false*. If set to false, this disables the dumping of *disc_copies* tables during startup while tables are being loaded. The default is true.
* *-mnesia dump_log_load_regulation true | false*. Controls if log dumps are to be performed as fast as possible, or if the dumper is to do its own load regulation. Default is *false*.

  This feature is temporary and will be removed in a future release
* *-mnesia dump_log_update_in_place true | false*. Controls if log dumps are performed on a copy of the original data file, or if the log dump is performed on the original data file. Default is *true*
*

  -mnesia dump_log_write_threshold Max. *Max* is an integer that specifies the maximum number of

writes allowed to the transaction log before a new dump of the log is performed. Default is *100* log writes.

*

-mnesia dump_log_time_threshold Max. *Max* is an integer that specifies the dump log interval in milliseconds. Default is 3 minutes. If a dump has not been performed within *dump_log_time_threshold* milliseconds, a new dump is performed regardless of the number of writes performed.

* *-mnesia event_module Module*. The name of the Mnesia event handler callback module. Default is *mnesia_event*.
* *-mnesia extra_db_nodes Nodes* specifies a list of nodes, in addition to the ones found in the schema, with which Mnesia is also to establish contact. Default is *[]* (empty list).
* *-mnesia fallback_error_function {UserModule, UserFunc}*. Specifies a user-supplied callback function, which is called if a fallback is installed and Mnesia goes down on another node. Mnesia calls the function with one argument, the name of the dying node, for example, *UserModule:UserFunc(DyingNode)*. Mnesia must be restarted, otherwise the database can be inconsistent. The default behavior is to terminate Mnesia.
* *-mnesia max_wait_for_decision Timeout*. Specifies how long Mnesia waits for other nodes to share their knowledge about the outcome of an unclear transaction. By default, *Timeout* is set to the atom *infinity*. This implies that if Mnesia upon startup detects a "heavyweight transaction" whose outcome is unclear, the local Mnesia waits until Mnesia is started on some (in the worst case all) of the other nodes that were involved in the interrupted transaction. This is a rare situation, but if it occurs, Mnesia does not guess if the transaction on the other nodes was committed or terminated. Mnesia waits until it knows the outcome and then acts accordingly.

  If *Timeout* is set to an integer value in milliseconds, Mnesia forces "heavyweight transactions" to be finished, even if the outcome of the transaction for the moment is unclear. After *Timeout* milliseconds, Mnesia commits or terminates the transaction and continues with the startup. This can lead to a situation where the transaction is committed on some nodes and terminated on other nodes. If the transaction is a schema transaction, the inconsistency can be fatal.

* *-mnesia no_table_loaders NUMBER*. Specifies the number of parallel table loaders during start. More loaders can be good if the network latency is high or if many tables contain few records. Default is *2*.
* *-mnesia send_compressed Level*. Specifies the level of compression to be used when copying a table from the local node to another one. Default is *0*.

  Level must be an integer in the interval *[0, 9]*, where *0* means no compression and *9* means maximum compression. Before setting it to a non-zero value, ensure that the remote nodes understand this configuration.

* *-mnesia schema_location Loc*. Controls where Mnesia looks for its schema. Parameter *Loc* can be one of the following atoms:

  disc:
  Mandatory disc. The schema is assumed to be located in the Mnesia directory. If the schema cannot be found, Mnesia refuses to start. This is the old behavior.

  ram:
  Mandatory RAM. The schema resides in RAM only. At startup, a tiny new schema is generated. This default schema only contains the definition of the schema table and only resides on the local node. Since no other nodes are found in the default schema, configuration parameter *extra_db_nodes* must be used to let the node share its table definitions with other nodes.

  Parameter *extra_db_nodes* can also be used on disc based nodes.

opt_disc:
Optional disc. The schema can reside on disc or in RAM. If the schema is found on disc, Mnesia starts as a disc-based node and the storage type of the schema table is *disc_copies*. If no schema is found on disc, Mnesia starts as a disc-less node and the storage type of the schema table is *ram_copies*. Default value for the application parameter is *opt_disc*.

First, the SASL application parameters are checked, then the command-line flags are checked, and finally, the default value is chosen.

SEE ALSO

application(3erl), **dets(3erl)**, **disk_log(3erl)**, **ets(3erl)**, **mnesia_registry(3erl)**, **qlc(3erl)**