Mnesia 用户手册

4.4.10 版

Mnesia 是一个分布式数据库管理系统（DBMS），适合于电信和其它需要持续运行和具备软实时特性的 Erlang 应用。

翻译：法无定法

mnesia 4.4.10

目　录

1、介绍 4
1.1 关于 Mnesia 4
1.2 Mnesia 数据库管理系统（DBMS） 4
2、开始 Mnesia....7
2.1 首次启动 Mnesia 7
2.2 一个示例....8
3、构建 Mnesia 数据库....22
3.1 定义模式....22
3.2 数据模型 23
3.3 启动 Mnesia 23
3.4 创建新表....26
4、事务和其他上下文存取....29
4.1 事务属性....29
4.2 锁....30
4.3 脏操作....33
4.4 记录名与表....34
4.5 作业（Activity）概念和多种存取上下文....37
4.6 嵌套事务....38
4.7 模式匹配 39
4.8 迭代....41
5、其它 Mnesia 特性....45
5.1 索引 45
5.2 分布和容错....45
5.3 表分片 46
5.4 本地内容表....53
5.5 无盘节点....54
5.6 更多的模式管理....55
5.7 Mnesia 事件处理....55
5.8 调试 Mnesia 应用....57
5.9 Mnesia 里的并发进程 58
5.10 原型....58
5.11 Mnesia 基于对象的编程....61
6 Mnesia 系统信息....64
6.1 数据库配置数据....64
6.2 内核转储(Core Dumps)....64
6.3 转储表....64

6.4 检查点.....64
6.5 文件.....65
6.6 在启动时加载表.....67
6.7 从通信失败中恢复.....68
6.8 事务的恢复.....68
6.9 备份、回滚以及灾难恢复.....69
7 Mnsia 与 SNMP 的结合.....74
7.1 结合 Mnesia 与 SNMP.....74
8 附录 A：Mnesia 错误信息.....75
8.1 Mnesia 中的错误.....75
9 附录 B：备份回调函数接口.....76
9.1 Mnesia 备份回调行为.....76
10 附录 C：作业存取回调接口.....83
10.1 Mnnesia 存取回调行为.....83
11 附录 D：分片表哈希回调接口.....92
11.1 mnesia_frag_hash 回调行为.....92

1、介绍

本手册介绍了 Mnesia 数据库管理系统。Mnesia 是一个分布式数据库管理系统(DBMS)，适合于电信和其它需要持续运行和具备软实时特性的 Erlang 应用，是构建电信应用的控制系统平台——开放式电信平台（OTP）的一部分。

1.1 关于 Mnesia

电信系统的数据管理与传统的商业数据库管理系统（DBMS）有很多相似之处，但并不完全相同。特别是许多不间断系统所要求的高等级容错、需要将数据库管理系统与应用程序结合运行在相同地址空间等导致我们实现了一个全新的数据库管理系统——Mnesia。Mnesia 与 Erlang 编程语言密切联系并且用 Erlang 实现，提供了实现容错电信系统所需要的功能。Mnesia 是专为工业级电信应用打造、使用符号编程语言 Erlang 写成的多用户、分布式数据库管理系统。Mnesia 试图解决典型电信系统的数据管理问题，具备一些在传统数据库中通常找不到的特性。

电信应用有许多不同于传统数据库管理系统的需求。用 Erlang 语言实现的应用程序需要具备宽广的特性，这是传统数据库管理系统无法满足的。Mnesia 的设计要求如下：

1、快速实时的键（key）/值（value）查找
2、主要用于运营和维护的非实时复杂查询
3、由于分布式应用导致的分布式数据
4、高容错
5、动态重配置
6、复杂对象

Mnesia 与其它大部分数据库管理系统的区别在于其是被设计用于解决电信应用中的典型数据管理问题。因此，Mnesia 有许多传统数据库的概念，如事务和查询，也有许多电信应用数据管理系统的概念，如高速实时操作，可配置的容错等级（在复制的意义上）以及不停机进行重新配置的能力等。Mnesia 与 Erlang 编程语言是紧耦合的，使得 Erlang 几乎成为数据库编程语言。其最大的好处是在操作数据时由于数据库与编程语言所用的数据格式不同而带来的阻抗失配问题完全消失。

1.2 Mnesia 数据库管理系统（DBMS）

1.2.1 特性

Mnesia 包含的下列特性组合后可以产生一个可容错的分布式数据库管理系统：

- 数据库模式（schema）可在运行时动态重配置；

- 表能被声明为有诸如位置（location）、复制（replication）和持久（persistence）等属性；
- 表能被迁移或复制到多个节点来改进容错性，系统的其它部分仍然可以对表进行存取来读、写和删除记录；
- 表的位置对程序员是透明的。程序通过表名寻址，由系统自身保持对表位置的追踪；
- 数据库事务能够被分布并且在一个事务中能够调用大量函数；
- 多个事务能够并发运行，由数据库管理系统完全同步其执行。Mnesia 保证不会有两个进程同时操作数据；
- 事务能被赋予要么在系统的所有节点上执行，要么不在任何一个节点上执行的特性。 能够用"脏操作（dirty operations）"绕过事务以减少开销并且大大加快运行速度。

这些特性的细节在下面章节中描述。

1.2.2 附加的应用程序

QLC 和 Mnesia Session 可以与 Mnesia 协同产生一些增强 Mnesia 操作能力的函数。Mnesia Session 和 QLC 都有其自己的文档作为 OTP 文档集的一部分。下面是 Mnesia Session 和 QLC 与 Mnesia 协同使用时的主要特性：

- QLC 具有为 Mnesia 数据库管理系统优化查询编译器的能力，从根本上让 DBMS 更高效；
- QLC 能用作 Mnesia 的数据库编程语言，它包括被称为"列表表达式（list comprehensions）"的标记方法并且能用来对表的集合做复杂的数据库查询；
- Mnesia Session 是 Mnesia 数据库管理系统的一个接口；
- Mnesia Session 允许外部编程语言（即除了 Erlang 以外的语言）访问 Mnesia DBMS。

1.2.2.1 何时使用 Mnesia

在下列类型的应用中使用 Mnesia：

- 需要复制数据的应用；
- 需要对数据执行复杂搜索的应用；
- 需要使用原子事务来同步更新多条记录的应用；
- 需要用到软实时特性的应用。

另一方面，Mnesia 可能不适合以下类型的应用：

- 处理纯文本或二进制文件数据的程序；
- 仅需查询字典的应用可使用标准库模块 `dets`，这是 `ets` 模块的磁盘版；
- 需要磁盘日志设施的应用应优先使用 `disc_log` 模块；
- 不适合硬实时系统。

1.2.3 范围和目的

本手册是 OTP 文档的一部分，它描述了怎样构建 Mnesia 数据库应用以及怎样集成和使用 Mnesia 数据库管理系统和 OTP。本手册讲述了编程结构并且包含了大量编程实例来解释 Mnesia 的使用。

1.2.4 前提

本手册的读者应熟悉系统开发原则和数据库管理系统，读者熟悉 Erlang 编程语言。

1.2.5 关于本书

本书包括下列各章：

- 第 2 章 开始 Mnesia：用一个数据库实例来介绍 Mnesia。实例演示了怎样启动一个 Erlang 会话，指定 Mnesia 数据库目录，初始化数据库模式，启动 Mnesia 并且创建表。还讨论了纪录初始原型的定义。
- 第 3 章 构建一个 Mnesia 数据库：对第 2 章介绍的步骤做更正式地描述， 即定义数据库模式、启动 Mnesia 以及创建表的 Mnesia 函数。
- 第 4 章 事务和其它存取上下文：描述使得 Mnesia 成为容错实时分布式数据库管理系统的事务属性。本章还描述了锁的概念，用来保证表的一致性。脏操作可绕过事务系统来改进速度并且减少开销。
- 第 5 章 多种 Mnesia 特性： 描述能用于构建复杂数据库应用的特性。这些特性包括索引、检查点、分布与容错、 无盘节点、复制操作、本地内容表、并发以及 Mnesia 基于对象的编程。
- 第 6 章 Mnesia 系统信息： 描述包含在 Mnesia 数据库目录里的文件，数据库配置数据，内核及表的转储， 此外，还包括了备份、回滚以及灾难恢复的原理。
- 第 7 章 Mnesia 与 SNMP 的结合：简短勾画 Mnesia 与 SNMP 的结合。
- 附录 A Mnesia 错误信息：列出了 Mnesia 的错误信息及其含义。
- 附录 B 备份回调函数接口： 此设施默认实现的程序代码列表。
- 附录 C 作业存取回调函数接口：一种可能实现此设施的的程序框架。

2、开始 Mnesia

本章介绍 Mnesia。在简要的讨论关于首次初始化设置后，会演示一个 Mnesia 数据库的实例。这个数据库示例将在后面的章节中被引用和修改以说明不同的程序结构。在本章中将举例说明下列强制过程：

- 启动一个 Erlang 对话并指定 Mnesia 数据库目录；
- 初始化数据库模式（schema）；
- 启动 Mnesia 并创建所要求的表。

2.1 首次启动 Mnesia

以下是 Mnesia 系统启动的一个简单演示。对话来自于 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
```

在上面的例子里，下列动作被执行：

- 启动 Erlang 时，参数-mnesia dir '"/tmp/funky"'指定了 Mnesia 存储数据的目录；
- mnesia:create_schema([node()])在本地节点上初始化一个空的 schema ；
- DBMS 通过 mnesia:start()启动；
- 通过 mnesia:create_table(funky, [])来创建表 funky；
- mnesia:info()根据数据库的状态来显示信息。

2.2 一个示例

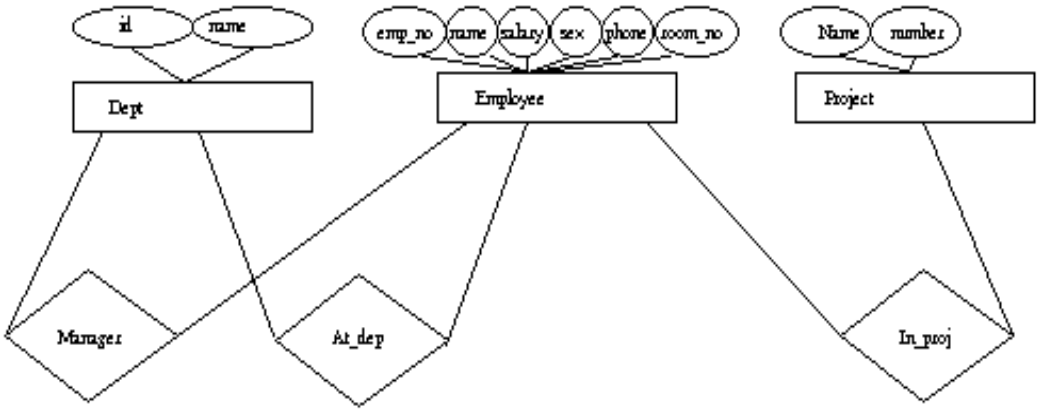
Mnesia 数据库被组织为一个表的集合，每个表由实例（Erlang 纪录）构成，表也包含一些属性，如位置（location）和持久性（persistence）。

在这个例子中：

- 启动一个 Erlang 系统，指定数据库位置目录；
- 初始化一个新的模式（schema），使用一个属性来指定数据库在那些节点上操作；
- 启动 Mnesia 本身；
- 创建并且驻留数据库表。

2.2.1 示例数据库

在这个数据库例子里，我们将创建如下数据库和关系，称为公司（Company）数据库。



公司实体关系图

数据库模型如下：

- 有三个实体：雇员（employee）、项目（project）、部门（department）。
- 这些实体间有三个关系：

  1、一个部门由一个雇员管理，因此有管理者（manager）的关系；

  2、一个雇员在一个部门工作，因此有在部门（at_dept）的关系；

  3、每个雇员为一些项目工作，因此有在项目中（in_proj）的关系。

2.2.2 定义结构和内容

我们首先将 record 定义输入到一个 company.hrl 文件，该文件定义了如下结构：

```
-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}).
```

该结构在我们的数据库中定义了 6 个表。在 Mnesia 里，函数 mnesia:create_table(Name, ArgList)用来创建表。 Name 是表名。注意：当前版本的 Mnesia 不要求表（table）名与记录（record）名一致，看第 4 章：记录名与表名（Record Names Versus Table Names）。

例如，通过函数 mnesia:create_table(employee, [{attributes, record_info(fields, employee)}]) 创建 employee 表，表名 emplpyee 匹配在 ArgList 中指定的记录名。Erlang 预处理器对表达式 record_info(fields, RecordName)进行处理，将由记录名标识的包含有不同域的记录解析为列表。

2.2.3 程序

以下在 shell 里的交互启动 Mnesia 并为我们的公司数据库初始化 schema。

```
% 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
```

以下程序模块创建前面定义的表：

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

2.2.4 程序的解释

以下命令和函数用来初始化 Company 数据库：

- % erl -mnesia dir '"/ldisc/scratch/Mnesia.Company"'。这是一个用来启动 Erlang 系统的 UNIX 命令行输入，旗标 -mnesia dir Dir 指定了数据库目录的位置，系统用 1>来响应并等待后面的输入；
- mnesia:create_schema([node()])。这个函数有一个格式；mnesia:create_schema(DiscNodeList) 用来初始化一个新的模式（schema）。在这个示例中，我们仅在一个节点上创建了一个非分布式的系统。模式的完整解释见第 3 章：定义模式。
- mnesia:start()。这个函数启动 Mnesia。完整解释见第 3 章：启动 Mnesia。

在 Erlang shell 里继续对话过程如下：

```
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
```

创建了一个表的集合：

- mnesia:create_table(Name,ArgList). 此函数用以创建要求的数据库表。ArgList 可用选项的解释见第 3 章：创建新表。

company:init/0 函数创建了我们的表。其中，manager 关系与 in_proj 关系一样，这两个表的类型都是 bag。因为一个雇员可以是多个部门的经理，一个雇员也能够参与多个项目。 at_dep 关系的类型是 set，因为一个雇员仅能在一个部门内工作。在我们这个所示例关系的数据模型中，set 表示一对一，bag 表示一对多。

mnesia:info()现在显示数据库有 7 个本地表，其中 6 个是我们定义的表，另一个是模式表。6 个事务被提交，在创建表的时候，6 个事务被成功运行。

写一个插入一条雇员记录到数据库的函数，必须同时插入一条 at_dep 记录和若干条 in_proj 记录。仔细研究下列用来完成这个动作的代码：

```
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) ->, insert_emp/3` 的参数是：
    1. `Emp` 是雇员记录；
    2. `DeptId` 是雇员工作部门的标识；
    3. `ProjNames` 是雇员参加的项目列表（list）。

insert_emp(Emp, DeptId, ProjNames) -> 函数创建一个函数对象，函数对象由 Fun 来标识。Fun 被当做一个单独的参数传递给函数 mnesia:transaction(Fun)，这意味着被当做一个事务来运行的 Fun 具备下列性质：

- Fun 要么完全成功要么完全失败；
- 操作同样数据记录的代码可以并行运行在不同的进程中而不会相互干扰。

该函数可以这样使用：

```
      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]).
```

注释：函数对象（Funs）在 Erlang 参考手册中的“Fun 表达式”中描述。

2.2.5 初始化数据库内容

在插入名为 **klacke** 的雇员后，我们在数据库中有下列记录：

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

Employee

一条雇员数据对应的元组表现为：{employee, 104732, klacke, 7, male, 98108, {221, 015}}

emp	dept_name
klacke	B/SFR

At_dep

At_dep 表现的元组为：{at_dep, klacke, 'B/SFR'}。

emp	proj_name

klacke	Erlang
klacke	otp
klacke	mnesia

In_proj

In_proj 表现的元组为：{in_proj, klacke, 'Erlang', klacke, 'otp', klacke, 'mnesia'}

表的行与 Mnesia 记录二者之间没有什么不同，两个概念是一样的。在本书中将交替使用。

一个 Mnesia 表由 Mnesia 记录构成。例如，元组{boss, klacke, bjarne}是一条记录。这个元组的第二个元素是键（key），为了唯一的标识出表中的行需要有表名和键这两个元素。术语对象标识符（object identifier）即 oid 有时是指{Tab, Key}这个二元元组。记录{boss, klacke, bjarne}的 oid 是二元元组{boss, klacke}。元组的第一个元素是记录的类型，第二个元素是键。取决于表的类型是 set 或 bag，一个 oid 可以对应零、一或多条记录。

我们插入{boss, klacke, bjarne}记录到数据库时可能隐含一个至今在数据库中还不存在的雇员，Mnesia 没有强制要求不能这样做。

2.2.6 添加记录和关系到数据库

在增加附加的纪录到公司数据库之后，我们以下列记录作为终结。

employee

```
{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"}.
```

project

```
{project, erlang, 1}.
{project, otp, 2}.
{project, beam, 3}.
{project, mnesia, 5}.
```

```
{project, wolf, 6}.
{project, documentation, 7}.
{project, www, 8}.
```

上述三个标题为 employees、dept 和 projects 的表是有实体记录构成的。下面在数据库中存储的表表现的是关系。这些表的标题为：manager，at_dep 和 in_proj。

manager

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

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}.
```

房间号是雇员记录的一个属性，这个属性由一个元组构成。元组的第一个元素标识过道（corridor），第二个元素标识在这个过道里的实际房间号。我们也可以选择用记录-record(room, {corr, no})来代替匿名的元组来表现。

2.2.7 写查询语句

从 DBMS 里获取数据的函数为 mnesia:read/3 或 mnesia:read/1。下列函数增加工资：

```
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).
```

由于我们希望在增加工资后使用 mnesia:write/1 来更新记录，所以我们在从表里读数据时要获得一个写锁(第三个参数)。

我们不会总是能够从一个表直接读出值，有时候我们需要搜索一个或多个表才能获取我们想要的数据，要做到这一点需要写数据库查询。查询总是会比直接用 mnesia:read 查找的开销要大很多。

有两种方式来写数据库查询:

- Mnesia 函数
- QLC

2.2.7.1 Mnesia 函数

下列函数从数据库获取女性雇员的名字：

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

select 必须运行在事务等作业里，为了能从 shell 里调用我们需要构造如下函数：

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

select 表达式匹配 employee 表里所有的记录中域 sex 为 female 的记录。

这个函数可以从 shell 里直接调用：

1> company:all_females().

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

也可以看 4.7 模式匹配（Pattern Matching）一节获得 select 的描述和其语法。

2.2.7.2 使用 QLC

这部分仅包含了简单的示例。对 QLC 查询语言的完整描述参考 QLC 参考手册。使用 QLC 可能比直接使用 Mnesia 函数开销更大，但是它提供了一个很好的语法。

下列函数从数据库中提取女性雇员的列表：

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

使用 QLC 列表表达式来访问 Mnesia 表时必须运行在一个事务里。参考如下函数：

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

如下所示，该函数可以从 shell 里调用：

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

在传统的关系数据库中术语中，上述操作被称为选择（selection）和映射（projection）。

上面示例的列表表达式包含的一些语法元素如下：

1）第一个方括号 [应视为“构建列表（list）”
2）|| 表示“如此这般”，箭头 <- 表示“从哪里获取”
因此，上面示例的列表表达式表示：构建列表 E#employee.name 的 E 取自于 employee 表并且每条记录的 sex 属性等于原子（atom）female。

整个列表表达式必须给到 qlc:q/1 函数中。

将列表表达式与低级的 Mnesia 函数在同一个事务中组合是可能的。如果我们要增加所有女性雇员的工资可以执行下列函数：

```
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.
```

函数 raise_females/1 返回元组{atomic, Number}，其中 Number 是获得增加工资的女性雇员的数量。当出现错误时，值{aborted, Reason}被返回。在出错的情况下，Mnesia 保证任何雇员的工资都不会被增加。

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

3、构建 Mnesia 数据库

本章详细介绍了设计 Mnesia 数据库和编程结构的基本步骤。本章包括下列部分：

- 定义模式
- 数据模型
- 启动 Mnesia
- 创建新表

3.1 定义模式

Mnesia 系统的配置在模式（schema）里描述。模式是一种特殊的表，它包含了诸如表名、每个表的存储类型（例如，表应该存储到 RAM、硬盘或者可能是两者以及表的位置）等信息。

不像数据表，模式表里包含的信息只能通过与模式相关的函数来访问和修改。Mnesia 提供多种方法来定义数据库模式，可以移动、删除表或者重新配置表的布局。这些方法的一个重要特性是当表在重配置的过程中可以被访问。例如，可以在移动一个表的同时执行写操作。该特性对需要连续服务的应用非常好。

下面的部分描述模式管理所要用到的函数，它们全部都返回一个元组（tuple）：

- `{atomic, ok}`或
- `{aborted, Reason}` 如果成功。

3.1.1 模式函数

- `mnesia:create_schema(NodeList)`. 该函数用来初始化一个新的空模式，在 Mnesia 启动之前这是一个强制性的必要步骤。Mnesia 是一个真正分布式的数据库管理系统，而模式是一个系统表，它被复制到 Mnesia 系统的所有节点上。如果 NodeList 中某一个节点已经有模式，则该函数会失败。该函数需要 NodeList 中所有节点上的 Mnesia 都停止之后才执行。应用程序只需调用该函数一次，因为通常只需要初始化数据库模式一次。
- `mnesia:delete_schema(DiscNodeList)`. 该函数在 DiscNodeList 节点上删除旧的模式，它也删除所有旧的表和数据。该函数需要在所有数据库节点（db_nodes）上的 Mnesia 都停止后才能执行。
- `mnesia:delete_table(Tab)`. 该函数永久删除表 Tab 的所有副本。
- `mnesia:clear_table(Tab)`. 该函数永久删除表 Tab 的全部记录。
- `mnesia:move_table_copy(Tab, From, To)`. 该函数将表 Tab 的拷贝从 From 节点移动到 To 节点。表的存储类型{type}被保留，这样当移动一个 RAM 表到另一个节点时，在新节点上也维持一个 RAM 表。在表移动的过程中仍然可以有事务执行读和写操作。
- `mnesia:add_table_copy(Tab, Node, Type)`. 该函数在 Node 节点上创建 Tab 表的备份。Type 参数必须是 ram_copies、 disc_copies 或者是 disc_only_copies。如果我们加一

个系统表 schema 的拷贝到某个节点上，这意味着我们要 Mnesia 模式也驻留在那里。这个动作扩展了组成特定 Mnesia 系统节点的集合。

- `mnesia:del_table_copy(Tab, Node)`. 该函数在 Node 节点上删除 Tab 表的备份，当最后一个备份被删除后，表本身也被删除。
- `mnesia:transform_table(Tab, Fun, NewAttributeList, NewRecordName)`. 该函数改变表 Tab 中所有记录的格式。它对表里所有记录调用参数 Fun 指明的函数进行处理，从表中取得旧的记录类型处理后返回新的纪录类型，表的键（key）可以不被改变。

```
-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),
```

Fun 的参数也可以是原子 ignore，它表示只更新表的元（meta）数据，不推荐使用（因为它将在元数据和实际数据之间产生矛盾）。但有可能用户需要用其在离线时做自己的转换。

- `change_table_copy_type(Tab, Node, ToType)`. 该函数改变表的存储类型。例如，将在 Node 节点上指定的内存类型的表 Tab 改为磁盘类型的表.

3.2 数据模型

Mnesia 采用的数据模型是一个扩展的关系数据模型。数据按照表的集合来组织，不同数据记录之间的关系通过描述实际关系的附加表来建模。每个表包含记录的实例，记录由元组表示。

对象标识，即 oid，由表名和键组成。例如，如果我们有一个雇员记录用元组表示为{employee, 104732, klacke, 7, male, 98108, {221, 015}}。这个纪录的对象标示（Oid）是元组{employee, 104732}。

因此，每个表由纪录（record）组成，第一个元素是记录名，表的第二个元素是标识表中特定记录的键（key）。表名和键的组合是一个被称为 Oid 的二元元组{Tab, Key}。参看第 4 章：记录名与表名，获得更多关于记录名和表名二者之间关系的信息。

Mnesia 数据模型是对关系模型的扩展，因为该模型可以在域属性里存储任意的 Erlang 项（term）。例如，可以在一个属性里存储指向在其它表中不同项的 oids 树，而这种类型的记录在传统的关系型 DBMS 里很难建模。

3.3 启动 Mnesia

在启动 Mnesia 之前我们必须在全部相关的节点上初始化一个空的模式。

- Erlang 系统必须启动。
- 设置为 disc 数据库模式的节点必须用函数 `create_schema(NodeList)` 来定义和执行。

当运行一个有两个或更多个节点参与的分布式系统时，必须在参与的每一个节点上分别执行函数 mnesia:start( )。典型的，在一个嵌入式环境里 mnesia:start( )应该成为启动脚本的一部分。

在一个测试环境或交互式环境里，mnesia:start( )也能够从 Erlang shell 或其他程序里调用。

3.3.1 初始化模式并启动 Mnesia

用一个已知的实例，我们说明怎样在被称为 a@gin 和 b@skeppet 这样两个分开的节点上运行在第 2 章描述的公司数据库。要想在这两个节点上运行公司数据库，每个节点在 Mnesia 启动前必须有一个在初始化模式时建立的 Mnesia 目录。 有两种方式指定 Mnesia 目录：

启动 Erlang shell 的时候或在应用程序脚本中通过一个应用程序参数来指定 Mnesia 目录。先前已用过下面的实例为我们的公司数据库创建目录：

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

如果没有输入命令行旗标（flag），则 Mnesia 使用当前节点 Erlang shell 启动时的工作目录作为 Mnesia 目录。

启动我们的公司数据库并使得其运行在两个分开的节点上，我们键入下列命令：

1. 在节点 gin 上调用：

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

2. 在节点 skeppet 上调用：

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

3. 在这两个节点之一上：

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

4. 在两个节点上调用函数 `mnesia:start()`。
5. 在两个节点之一执行以下代码来初始化数据库：

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

```
mnesia:create_table(at_dep,
                    [{ram_copies, [a@gin, b@skeppet]},
                     {attributes, record_info(fields, at_dep)}]),
mnesia:create_table(in_proj,
                   [{type, bag},
                    {ram_copies, [a@gin, b@skeppet]},
                    {attributes, record_info(fields, in_proj)}]).
```

如上所示，两个目录驻留在不同的节点上。因为/ldisc/scratch (本地磁盘)存在于这两个不同的节点上。

通过执行这些命令，我们已经配置了两个 Erlang 节点并初始化以运行公司数据库。这些操作仅需要在设置时做一次，以后这两个节点上的系统将通过调用 mnesia:start()来启动系统。

在 Mnesia 节点的系统中，每个节点都知道所有表的当前位置。在这个例子中，数据被复制到两个节点上，对我们表里的数据进行操作的函数可以在这两个节点中的任何一个上执行。无论数据是否驻留在那里，操作 Mnesia 数据的代码行为都是一样的。

mnesia:stop()函数在当前节点上停止 Mnesia，start/0 和 stop/0 二者都只在本地 Mnesia 系统上起作用，没有启动和停止节点集合的函数。

3.3.2 启动过程

Mnesia 通过调用下列函数来启动：

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

该函数在本地初始化 DBMS。

配置选择会更改表的位置和加载顺序。可替换的选择如下所列：

1、表只存储在本地，从本地 Mnesia 目录初始化；
2、根据本地或其它节点上的表哪个是最新的来决定是从本地硬盘还是其它节点复制表到本地来初始化，Mnesia 会检测哪些表是最新的；
3、一旦表被加载就可以被其他节点访问。

表初始化是同步的。函数调用 mnesia:start()返回原子 ok 并且开始初始化不同的表。如果数据库比较大，将花费一些时间，应用程序员必须等待，直到应用程序要用到的表可用时为止。这可以使用下列函数来实现：

- `mnesia:wait_for_tables(TabList, Timeout)`

此函数暂停调用程序直到在 Tablist 中指定的全部表都正确的初始化完成。

如果 Mnesia 推断另一个节点（远程）的拷贝比本地节点的拷贝更新时，初始化时在节点上复制表可能会导致问题，初始化进程无法处理。在这种情况下，对 mnesia:wait_for_tables/2 的调用将暂停调用进程，直到远程节点从其本地磁盘初始化表后通过网络将表复制到本地节点上。

这个过程可能相当耗时，下面的函数将以较快的速度从磁盘载入全部的表。

- `mnesia:force_load_table(Tab)`，此函数将不管网络情况如何强制从磁盘上加载表。

因此，我们假定如果应用程序希望使用表 a 和 b，在表能够被使用之前，应用程序必须执行一些类似下面代码完成的动作。

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

警告

当从本地磁盘强制加载表，本地节点宕机而远程节点的复制被激活时，所有执行复制表的操作将丢失。这将使得数据库变得不稳定。

如果启动过程失败，mnesia:start() 函数返回加密（cryptic）元组{error,{shutdown,{mnesia_sup,start,[normal,[]]}}。在 erl 脚本里使用命令行参数-boot start_sasl 可获得更多有关启动失败的信息。

3.4 创建新表

Mnesia 提供一个函数创建新表，这个函数是：

mnesia:create_table(Name, ArgList).

当执行这个函数时，其返回下列回应之一：

- `{atomic, ok}`如果函数执行成功
- `{aborted, Reason}`如果函数失败

此函数的参数为：

- Name 是原子类型（atomic）的表名，其通常与构成表的记录名同名（参看 record_name 获得更多的细节）。
- ArgList 是一个{Key,Value}元组的列表。下列参数是有效的：
  - {type, Type}这里 Type 必须是 set，ordered_set 或 bag 这三个原子之一，默认值为 set。注意：目前'ordered_set'不支持'disc_only_copies'表。set 或 ordered_set 类型的表每个键只能有零或一条记录，而 bag 类型的表每个键可以有任意数量的记录。每条记录的键总是记录的第一个属性。
    下面的例子示例了类型 set 和 bag 之间的不同：

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

如果 foo 表的类型是 set，事务将返回列表[{foo,1,3}]。但如果表的类型是 bag，将返回列表[{foo,1,2},{1,3}]。注意 bag 和 set 表类型的用途。
在同一个 Mnesia 表中不能有相同的记录，即不能有键和属性的内容都相同的记录。

- {disc_copies, NodeList}，这里 NodeList 是表要存储在磁盘上的节点列表。默认值是 []。
  对 disc_copies 类型表副本的写操作会将数据写到磁盘和内存的表中。
  将表以 disc_copies 类型放在一个节点上， 同一个表的副本以不同存储类型存放在其它节点上是可能的。如此安排在有如下需求时是我们所希望的:
  1. 读操作必须很快并且需要在内存中执行
  2. 全部写操作必须写到可持久的存储中。

  对 disc_copies 表的写操作会分两步执行，首先将写操作添加到日志文件中，然后在内存中里执行实际的操作。
- {ram_copies, NodeList}，这里 NodeList 是表要存储在内存中的节点列表，默认值是 [node()]。如果采用默认值创建新表，将仅仅位于本地节点。
  ram_copies 类型表的副本可以用 mnesia:dump_tables(TabList)函数来转储到磁盘上。
- {disc_only_copies, NodeList}，这种类型的表副本只存储在硬盘上，因此访问比较慢，但是这种类型的表比其它两种类型的表消耗的内存要少。
- {index, AttributeNameList}，这里 AttributeNameList 是一个原子类型的属性名列表，Mnesia 将对其指定的属性建立和维护索引，列表中的每个元素都有一个索引表。
  Mnesia 记录的第一个域是键，所以不需要建立额外的索引。
  记录的第一个字段是元组的第 2 个元素，其被用于表现记录。
- {snmp, SnmpStruct}. SnmpStruct 在 SNMP 用户指南中描述。如果在函数 mnesia:create_table/2 的参数 ArgList 中出现这个属性，表示该表可以立即通过简单网络管理协议(SNMP)来访问。
  可以很容易设计出使用 SNMP 操作和控制系统的应用程序。Mnesia 提供由逻辑表构成的 SNMP 控制应用程序与由物理数据构成的 Mnesia 表之间的直接映射。默认值为[]。
- {local_content, true}，当应用需要一个其内容对每个节点来说在本地都是唯一的表时，可使用 local_content 表。表名对所有 Mnesia 节点可见，但是内容对每个节点都是唯一的。这种类型的表只能在本地进行存取。
- {attributes, AtomList}， AtomList 是一个准备用于构成表的记录的属性名列表。 默认值是列表[key, val]。表必须至少有一个除了键之外的属性。当存取记录的单个属性时，不建议将属性名作为原子硬编码，可用结构 record_info(fields,record_name)来代替。表达式 record_info(fields,record_name)被 Erlang 宏预处理程序处理后返回记录的域名列表。定义记录-record(foo, {x,y,z})，表达式 record_info(fields,foo)被扩展为列表[x,y,z]。所以，或者是你自己提供属性名，或者用 record_info/2 标记。
  建议采用 record_info/2 标记，以使得程序更容易维护，具有更好的健壮性，也便于将来修改记录。

- {record_name, Atom}指定表中所有记录的通用名，全部存储在表中的记录都必须用这个名字作为其第一个元素。默认的记录名是表的名字。更多信息参见第 4 章：记录名与表名。

作为一个示例，我们定义记录：

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

下列调用将创建一个复制到两个节点，在 y 属性上有一个附加的索引，类型 bag 的表：

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

而下列调用采用默认值：

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

将返回一个在本地节点上用内存拷贝，没有附加的索引，并且属性默认为列表[key,val]的表。

4、事务和其他上下文存取

本章描述 Mnesia 事务系统和事务属性，其使得 Mnesia 成为一个容错的、分布式的数据库管理系统。

本章的内容也涉及到锁函数，包括表锁（table lock）和粘锁（sticky lock）以及绕开事务机制的替换函数，这些函数被称为“脏操作（dirty operation）”。我们还将描述嵌套事务（nested transaction）。本章包含下列部分：

- 事务属性，包括原子性，一致性，隔离性，持久性
- 锁
- 脏操作
- 记录名与表
-
- 作业（Activity）概念和多种上下文存取
- 嵌套事务
- 模式匹配
- 迭代

4.1 事务属性

在设计容错和分布式系统时事务是一个重要的工具。Mnesia 事务是一种可以将一系列数据库操作作为一个函数块来执行的机制。作为一个事务来运行的函数块被称为函数对象(Fun)，它可以对 Mnesia 记录进行读、写和删除。 Fun 作为一个事务，要么提交要么终止，如果事务成功执行，它将在所有相关的节点上复制这个动作，如果出错则终止事务。

```
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).
```

事务 raise(Eno, Raise) - >包括一个 Fun，这个 Fun 被语句 mnesia:transaction(F)调用并返回一个值。

Mnesia 事务系统通过提供以下重要特性来方便构建可信任的、分布式的系统：

- 事务处理器保证在一个事务中的 Fun 在对一些表执行一系列的操作时不会干预在其他事物中的操作。
- 事务处理器保证事务中的操作要么在所有节点上完全原子化的成功执行，要么失败并且对所有节点没有任何影响。

- Mnesia 事务有四大特性，即原子性（A），一致性（C），隔离性(I)和持久性（D），简写为 ACID，这些特性描述如下。

4.1.1 原子性

原子性意味着在通过事务对数据库实施的改变或者在全部节点上都被执行，或者没有一个节点执行。换言之，事务要么完全成功，要么完全失败。

当我们希望在同一个事务中写多条记录时，原子性特别重要。前述示例中的 raise/2 函数仅写入一条记录。第 2 章里的 insert_emp/3 函数在写记录 employee 的时候也将与其关联的 at_dept 和 in_proj 记录写到数据库里。如果我们在一个事务中运行这些代码，事务处理机制确保事物要么成功完成，要么根本就不执行。

Mnesia 是一个可以将数据复制到多个服务节点上的分布式数据库管理系统。原子性保证一个事务要么在所有的节点都有效，要么没有一个节点有效。

4.1.2 一致性

一致性保证了事务总是让数据库管理系统保持一致的状态。 例如，当 Erlang、Mnesia 或者计算机崩溃时，如果有一个写操作正在运行，Mnesia 确保不出现数据不一致的情况。

4.1.3 隔离性

隔离性保证当事务在网络的不同节点上执行时，对相同数据记录的存取操作不会互相干扰。

这使得并发执行 raise/2 函数成为可能。在并发控制理论里的一个经典问题是“更新丢失问题（lost update problem）”。

在出现下列情况时隔离性特别有用：有一个编号为 123 的雇员和两个进程（P1 和 P2），这两个并发进程试图给这个雇员加薪，雇员起初的薪水比如说是 5。进程 P1 开始执行，读取该雇员的记录并对其薪水加 2 。在这个时点，进程 P1 由于某种原因暂停而进程 P2 获得机会运行，P2 读取该雇员的记录并对其薪水加 3，最终写入一条薪水为 8 的员工记录。 现在 P1 开始再次运行，写入一条薪水为 7 的员工记录，于是有效的覆盖和取消了进程 P2 所执行的工作，P2 所作的更新被丢弃。

事务系统使得并发执行的两个或多个进程操作相同记录成为可能。程序员不需要检查更新是否同步，事务处理机制会监督这一点。所有通过事务系统访问数据库的程序都可以认为自己对数据有唯一的访问权限。

4.1.4 持久性

持久性。事务对数据库管理系统所做的改变是永久的。 一旦事务提交，对数据库所做的任何更改都是持久的——它们被安全的写入磁盘，不会消失或者被损坏。

注意

如果将 Mnesia 配置为纯内存数据库则持久性不起作用。

4.2 锁

不同的事务管理器使用不同的策略来满足隔离属性。Mnesia 使用两阶段锁（two-phase locking）的标准技术，这意味着记录在读写之前被加锁，Mnesia 使用 5 种不同的锁。

- 读锁。在记录的副本能被读取之前设置读锁。.
- 写锁。当事务写一条记录时，首先在这条记录的所有副本上设置写锁。
- 读表锁。如果事务要扫描整张表来搜索一条记录，那么，对表里的记录一条一条的加锁效率很低也很耗内存（如果表很大，读锁本身会消耗很多空间）。因此，Mnesia 可以对表设置读锁。
- 写表锁。如果事务要写大量的记录到表里，则可以对整张表设置写锁。
- 粘（**Sticky**）锁。即使设置锁的事务终止后，这些写锁也会一直保留在节点上。

当事务执行时，Mnesia 采取的策略是借助诸如 mnesia:read/1 这样的函数来获得需要的动态锁。Mnesia 会自动加锁和解锁，程序员不必对这些操作编码。

当并发进程对相同的记录进行加锁或解锁时可能出现死锁。Mnesia 使用“等待-死亡（wait-die）”策略来解决这个问题。当某个事务尝试加锁时，如果 Mnesia 怀疑可能出现死锁，就强制该事务释放所有的锁并休眠一段时间。包含在事务中的函数（Fun）将被求值一次或多次。

由于上述理由，保持传递给 mnesia:transaction/1 的函数（Fun）中的代码干净很重要，否则会引发一些奇怪的结果，例如，通过事务函数（Fun）发送消息。下面的实例演示了这种情况：

```
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).
```

这个事务可能会将文本“Trying to write ...”写一千次到终端上。尽管如此，Mnesia 会保证每个事务最终会运行。结果，Mnesia 不仅仅释放死锁，也释放活锁。

Mnesia 程序员不能区分事务的优先级，所以 Mnesia DBMS 事务系统不适合硬实时应用。但是 Mnesia 包括其他软实时特性。

当事务执行时 Mnesia 动态加锁和解锁，所以执行有事务副作用的代码是很危险的。特别是在事务中含有 receive 语句时会让事务一直挂着而无法返回，这会导致锁不被释放。这种情况会使整个系统停顿，因为其他节点、其他事务的进程会被强制等待这个有问题的事务。

如果事务异常终止，Mnesia 将自动释放该事务的锁。

我们在前面已经示范了一些可以用于事务中的函数。下面会列出可在事务中工作的最简单的 Mnesia 函数。重要的是要知道这些函数必须被嵌入到事务中。如果没有封装事务（或其他可用来封装的 Mnesia 作业）存在，这些函数将失败。

- `mnesia:transaction(Fun) -> {aborted, Reason} |{atomic, Value}`，这个函数用函数对象 Fun 作为单独的参数执行一个事务。
- `mnesia:read({Tab, Key}) -> transaction abort | RecordList`，这个函数从表 Tab 中读取全部键值为 Key 的记录。无论表在哪里，此函数具有同样的语义。如果表的类型是 bag，则 read({Tab, Key})可能返回一个任意长的列表。如果表的类型是 set，则列表的长度等于一或者为`[]`。
- `mnesia:wread({Tab, Key}) -> transaction abort | RecordList`，该函数与上面列出的 read/1 函数行为一样，只是其会获取一个写锁而不是一个读锁。如果我们执行一个读取一条记录，修改这条记录，然后再写此记录的事务，那么立即设置一个写锁会更有效。 如果我们先调用 mnesia:read/1，然后调用 mnesia:write/1，那么执行写操作时必须将读锁升级为写锁。
- `mnesia:write(Record) -> transaction abort | ok`， 这个函数将一条记录写入数据库中。 Record 参数是记录的一个实例，此函数返回 ok 或者在错误发生时中止事务。

- `mnesia:delete({Tab, Key}) -> transaction abort | ok`， 这个函数删除 Key 对应的所有记录。
- `mnesia:delete_object(Record) -> transaction abort | ok`， 这个函数删除对象标识为 Record 的记录。该函数仅用来删除表的类型为 bag 的记录。

4.2.1 粘（Sticky）锁

如上所述，Mnesia 使用的锁策略是在读一条记录时锁住该条记录，写一条记录时锁住该条记录的所有副本。但有一些应用使用 Mnesia 主要是看中了其容错的特点，这些应用可能配置为一个节点承担所有繁重的任务，而另一个备用节点在主节点失败时来接替它。这样的应用使用粘锁来代替普通的锁会更有利。

粘锁是这样一种锁，在第一次设置这个锁的事务终止后锁依然留在节点的适当位置上。为了演示这一点，假设我们执行下列事务：

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

foo 表被复制到 N1 和 N2 这两个节点上。
普通的锁要求：

- 一个网络远程调用（2 条消息）来获取写锁；
- 三条网络消息来执行两阶段提交协议。

如果我们使用粘锁，必须首先将代码修改如下：

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

这段代码使用 s_write/1 函数来代替 write/1 函数。 s_write/1 函数用粘锁来代替普通的锁。如果表没有被复制，粘锁没有任何特殊效果。 如果表被复制，并且我们在 N1 节点加一个粘锁，于是这个锁将会粘到 N1 节点上。下次我们试着在 N1 节点的同一条记录加锁时，Mnesia 将会发现纪录已经加锁，不必再通过网络操作来加锁。

本地加锁比通过网络加锁更高效，因此粘锁对需要对表备份且大部分工作仅在一个节点上处理的应用更有利。

如果 N1 节点上的一条记录被粘着（stuck），当我们试着在 N2 节点上对同一条记录加粘锁时，该记录必须是未被粘着（unstuck）的。这种操作的开销很大并且会降低性能。如果我们在 N2 节点上发布 s_write/1 请求，解除粘着（unsticking）会自动完成。

4.2.2 表锁

Mnesia 支持对整个表的读和写锁作为只针对单条记录的普通锁的补充。如前所述，Mnesia 会自动设置和释放锁，无须程序员对这些操作编码。但是，如果在一个事务中存在对某个表中的大量记录进行读写操作的情况下，我们在开始这个事务时对该表加表锁来阻塞来自于这个表的其它并发进程将更有效率。下面的两个函数被用来对读写操作显式的加表锁：

- `mnesia:read_lock_table(Tab)` 在表 Tab 上加读锁
- `mnesia:write_lock_table(Tab)` 在表 Tab 上加写锁

获得表锁可替换的语法如下：

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

Mnesia 内的匹配操作既可能锁定整个表，也可能只锁定单条记录（当键 Key 在模式中被绑定时）。

4.2.3 全局锁

写锁一般会要求在所有存放有表的副本并且是活动的节点上设置。读锁只在一个节点（如果存在本地副本的话就是本地节点）上设置。

函数 mnesia:lock/2 被用来支持表锁（如上所述），但也可用于无论表是怎样被复制都需要加锁的情况：

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

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

锁被加在节点列表中指定的所有节点的加锁项（LockItem）上。

4.3 脏操作

在许多应用里，事务的过度开销可能导致性能损失。脏操作是绕开很多事务处理、增加事务处理速度的捷径。

脏操作在很多情况下是非常有用的，例如，在数据报路由应用里，Mnesia 存储路由表，每接收一个包就启动整个事务是非常耗时的。因此，Mnesia 有无须事务即可对表进行操作的函数。这种替代的处理被称为脏操作。但是，懂得权衡避免事务处理的开销非常重要：

- 会失去 Mnesia 的原子性和隔离性；
- 隔离性受到拖累，因为在其它 Erlang 进程使用事务操作数据时，如果我们同时正在使用脏操作从同一个表中读写记录，那么，使用事务的进程无法获得隔离性带来的好处。

脏操作的主要优点是它们比对应的事务处理快的多。

如果脏操作在 disc_copies 或 disc_only_copies 类型的表上执行，会写到磁盘上。Mnesia 也能保证如果执行对表的脏写操作，那么这个表的所有副本都会被更新。

脏操作将保证某种程度的一致性，例如脏操作不可能返回混乱的记录。因此，每一个单独读或写操作是以原子行为的方式执行。

所有的脏操作在失败时调用 exit({aborted, Reason})，下列函数是可用的，这些函数即使在事务中执行也不会加锁：

- `mnesia:dirty_read({Tab, Key})`， 此函数从 Mnesia 读取记录；
- `mnesia:dirty_write(Record)`， 此函数写记录 `Record`；
- `mnesia:dirty_delete({Tab, Key})`， 此函数删除以键值 `Key` 标识的记录；
- `mnesia:dirty_delete_object(Record)`， 此函数是替换 `delete_object/1` 的脏操作；
- `mnesia:dirty_first(Tab)`，此函数返回表 `Tab` 中的“第一个” 键
  在 `set` 或 `bag` 表中的 记录是 不被排序的。然而， 存在着一个不 为用 户所知的 记录顺序。 这表明用这个函数与 `dirty_next/2` 函数协同来遍历表是可能的。

如果表中根本就没有记录， 此函数将返回原子'$end_of_table'。 不建议任何用户记录使用这个原子作为键。

- mnesia:dirty_next(Tab, Key)， 此函数返回表 Tab 中的“下一个”键。 这个函数使得遍历表并且对表中的全部记录执行一些操作是可能的。当达到表的终点时，函数返回'$end_of_table'；否则，函数返回能被用于读取实际记录的键。
  当我们用 dirty_next/2 函数遍历表的时候，如果有任何进程在表上执行写操作，其行为都是不确定的。这是因为在 Mnesia 表上的写操作会导致表内部的重新组织。 这是一个实现细节，但 请记住 脏函数是一些低 级别的函数。
- mnesia:dirty_last(Tab)此函数与 mnesia:dirty_first/1 工作方式是完全一样的， 但对 ordered_set 类型的表返回按 Erlang 项序中排列的最后一个对象。对其它类型的表， mnesia:dirty_first/1 与 mnesia:dirty_last/1 的语义是相同的。
- mnesia:dirty_prev(Tab, Key) 此函数与 mnesia:dirty_next/2 工作方式是完全一样的，但对 ordered_set 类型的表返回按 Erlang 项序中排列的前一个对象。对其它类型的表， mnesia:dirty_next/2 与 mnesia:dirty_prev/2 的语义是相同的。
- mnesia:dirty_slot(Tab, Slot)
  返回表中与 Slot 有关联的记录列表。此函数可用于与 dirty_next/2 函数类似的方式遍历表。表有一些范围从 0 到某些未知上界的 slots。当达到表的终点时，函数 dirty_slot/2 返回特殊原子'$end_of_table'。
  在遍历表的时候，如果对表写入，此函数的行为是不确定的。可用 mnesia:read_lock_table(Tab)来确保迭代期 间事 务保 护的写操作没有被 执行。
- mnesia:dirty_update_counter({Tab,Key}, Val).
  计数器是一个取值大于或等于 0 的正整数。 更新计数器将加 Val 到计数器，Val 是一个正或负的整数。
  在 Mnesia 中，不存在特殊的计数器记录。 不过，{TabName, Key, Integer}形式的记录能够被用作计数器并且是能持久的。
  计数器记录没有事务保护更新。
  当使用这个函数代替读记录、执行计算以及写记录时有两个重要的区别:
  1. 会非常高效；
  2. 虽然 dirty_update_counter/2 函数不被事务保护，但其被作为一个原子操作执行。因此，如果有两个进程同时执行 dirty_update_counter/2 函数，对表做的更新也不会 丢失。
- mnesia:dirty_match_object(Pat)，此函数是等同于 mnesia:match_object/1 的脏函数。
- mnesia:dirty_select(Tab, Pat)，此函数是等同于 mnesia:select/2 的脏函数。
- mnesia:dirty_index_match_object(Pat, Pos)，此函数是等同于 mnesia:index_match_object/2 的脏函数。

- `mnesia:dirty_index_read(Tab, SecondaryKey, Pos)`，此函数是等同于 `mnesia:index_read/3` 的脏函数。
- `mnesia:dirty_all_keys(Tab)`，此函数是等同于 `mnesia:all_keys/1` 的脏函数。

4.4 记录名与表

在 Mnesia 里，表内所有记录必须有同样的名字，所有记录必须是同一记录类型的实例。记录名可以是但不必须是表名，尽管本文大多数示例中的表名和记录名都是一样的。如果创建表的时候没有指定记录名（record_name）属性，下面的代码将确保表中所有的记录会有和表名相同的字：

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

如下所示，如果创建表的时候显示指定一个记录名(subscriber)作为参数，那么，在这两个表（my_subscriber 和 your_subscriber）中存储 subscriber 记录而不管表名是可能的：

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

存取上述指定了记录名的表不能用本文前面介绍过的简单存取函数，比如，写一条 subscriber 记录到表中需要用函数 mnesia:write/3 来代替简单函数 mnesia:write/1 和 mnesia:s_write/1：

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

下列简单的代码片断说明了在大多数示例中使用的简单存取函数与其更灵活的对应者之间的关系：

```
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).
```

4.5 作业（Activity）概念和多种存取上下文

如前面所述，可以作为对象 (Fun)参数传递给函数 mnesia:transaction/1,2,3、执行对表存取操作的函数如下：

- 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

这些函数将在涉及到锁、日志、复制、检查点、订阅、提交协议等上下文相关机制的事务中执行。此外，这些函数还可在其它上下文相关的作业中执行。 目前支持的上下文相 关作 业如 下：

- transaction （事务）
- sync_transaction （同步事务）
- async_dirty （异步脏操作）
- sync_dirty （同步脏操作）
- ets

作为参数“fun”传递给函数 mnesia:sync_transaction(Fun [, Args])的函数将在同步事务的上下文里执行。在从 mnesia:sync_transction 调用返回以前，同步事务将一直等待直到全部激活的副本提交给事务（写到磁盘）。同步事务对需要在多个节点上执行并且需要在派生远端进程或发送消息给远端进程之前确认远程节点的更新已被执行的应用以及在组合了事务写和脏读（dirty_reads）的时候特别有用。对由于频繁执行和大量更新可能导致其它节点的 Mnesia 过载的应用也很有用。

作为参数“fun”传递给函数 mnesia:async_dirty(Fun [, Args])的函数将在脏（dirty）上下文里执行。这个函数调用将被映射到对应的脏函数。仍将涉及到日志、复制和预订但不涉及锁、本地事务存储或提交协议。检查点保持器（Checkpoint retainers）将被更新，但将是“脏”的，即更新是异步的。此函数将在一个节点上等待操作被执行而不管其它节点，如果表在本地则不会出现等待。

作为参数“fun”传递给函数 mnesia:sync_dirty(Fun [, Args])的函数将在与函数 mnesia:async_dirty(Fun [, Args])几乎相同的上下文里执行。不同的是操作被同步执行，调用者将等待全部激活的副本更新完成。同步脏操作（sync_dirty）对需要在多个节点上执行并且需要在派生远端进程或发送消息给远端进程之前确认远程节点的更新已被执行的应用以及由于频繁执行和大量更新可能导致其它节点的 Mnesia 过载的应用非常有用。

你能用 mnesia:is_transaction/0 函数检查你的代码是否在一个事务中被执行，当调用在一个事务的上下文里，函数返回 true，否则返回 flase。

存储类型为 RAM_copies 和 disc_copies 的 Mnesia 表内部是用 ets 表来实现的，因此，应用程序直接存取这些 ets 表是可能的，这仅仅在所有的选择都已经被权衡过并且知道可能后果的情况下才能考虑。将前面提到过的函数作为参数传递给函数 mnesia:ets(Fun [, Args])后将在一个很原始的上下文里执行。如果本地的存储类型是 RAM_copies 并且表没有被复制到其它节点，操作将直接在本地的 ets 表上执行。既不会触发订阅，也不会更新检查点，但这些操作快如闪电。使用 etc 函数无法更新驻留在磁盘上的表，因为无法对磁盘做更新。

函数也可以作为参数传递给函数 mnesia:activity/2,3,4 从而激活可定制的作业存取回调模块。可以直接用模块名作为参数，或者使用隐含在存取模块（access_module）里的配置参数。可定制的回调模块可用于几个目的，如触发器、完整性限制、运行时统计或者虚表等。回调模块不必存取实际的 Mnesia 表，只要回调接口能够实现，就能够做任何事情。附录 C“作业存取回调接口”里提供了一个可替换实现的源代码（mnesia_frag.erl）。上下文敏感的函数 mnesia:table_info/2 也被用来提供关于表的虚信息，用在可定制回调模块作业上下文里执行 QLC 查询。通过提供关于表索引和其它 QLC 要求的表信息，QLC 可作为存取虚表的通用查询语言。

QLC 查询可以在所有这些作业上下文（transaction, sync_transaction, async_dirty, sync_dirty and ets）里执行。ets 作业仅在表没有索引的情况下工作。

> **注意**
>
> mnesia:dirty_*类的函数总是以异步脏（async_dirty）的语法执行而不管作业存取上下文是如何请求的。其甚至可以不需要封装到任何作业存取上下文中即可调用。

4.6 嵌套事务

事务可以任意嵌套。一个子事务必须与其父事务运行在同一个进程里。当子事务中断时，子事务的调用者会得到返回值{aborted, Reason}并且子事务执行的任何工作都会被删除。 如果子事务提交，则子事务写入的记录会传播到父事务。

当子事务终止时不会释放锁，一个嵌套事务序列创建的锁会一直保持到最顶层的事务终止时为止。除此之外，嵌套事务所执行的任何更新按照只有被嵌套事务的父事务能够看到此更新的方式传播。最顶层的事务没有完成就不会有任何承诺。因此，即使嵌套的事务返回{atomic, Val}，如果封装的父事务失败，那么整个嵌套的操作也都失败。

嵌套事务与顶层事务具有相同语义的能力使得编写操作 Mnesia 表的库函数更容易。

下面的例子中的函数是加一个新用户到电话系统中：

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

该函数需要作为一个事务来调用。 现在假设我们希望写一个函数，该函数调用 add_subscriber/1 函数并且其自身是在一个事务上下文的保护中。 通过在另一个事务里简单的调用 add_subscriber/1 就创建了一个嵌套事务。

有可能在嵌套时混用不同的作业存取上下文，但如果在一个事务内部调用脏的那一类（async_dirty,sync_dirty 和 ets）将会继承事务语义，因此，其将抢占锁并且使用两或三段提交。

```
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).
```

4.7 模式匹配

当不能使用 mnesia:read/3 时，Mnesia 提供了若干函数来对记录进行模式匹配：

```
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
```

这些函数用表 Tab 中的所有记录匹配一个模式（ Pattern）。在 mnesia:select 调用中模式是下面描述的 MatchSpecification 的一部份。没有必要对全表执行穷举搜索，通过使用索引和模式键的绑定

值，函数实际要做的工作可能压缩为几条哈希查询。如果键被部分绑定，可使用 ordered_set 表来能减少搜索空间。

提交给函数的模式必须是合法记录并且元组的第一个元素必须是表的记录名。特殊元素‘_’匹配 Erlang 的任何数据结构。特殊元素'$<number>'的行为与 Erlang 变量一样，比如，匹配和绑定第一次出现的任何东西并且随后匹配对此变量的绑定值。

用函数 mnesia:table_info(Tab, wild_pattern)来获得匹配表内所有记录的基本模式或者使用创建记录时的默认值。不要用硬编码来制定模式，因为这将使得你的代码在将来记录定义改变时变得很脆弱。

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

雇员表的通配模式看起来像下面这样：

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

为了限制匹配你必须放置一些‘_’元素，匹配出雇员表中所有女性雇员的代码看起来像下面这样：

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

使用匹配函数来检查不同属性是否相等也是可能的。假如我们要找出所有雇员号与房间号相等的雇员：

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

函数 mnesia:match_object/3 缺乏一些函数 mnesia:select/3 具备的重要特征。例如，mnesia:match_object/3 仅能返回匹配的记录而不能表达其它相等的约束。如果我们要找出二楼男性雇员的名字，能够这样写：

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

Select 函数被用于增加约束和获得 mnesia:match_object/3 无法实现的输出。

Select 函数的第二个参数是一个匹配说明（MatchSpecificatio）。匹配说明是匹配函数（MatchFunctions）的列表，每个匹配函数由一个包含{MatchHead, MatchCondition, MatchBody}的元组构成。MatchHead 模式与上述 mnesia:match_object/3 所用的一样。MatchCondition 是一个应用于每条记录的附加约束的列表，MatchBody 用来构造返回值。

匹配说明的详细解释可在《Erts 用户指南：Erlang 中的匹配说明》中找到，ets/dets 的文档中也有一些附加信息。

函数 select/4 和 select/1 用于获得有限数量的结果，Continuation 用于取得结果的下一部分。Mnesia 用 NObjects 仅仅是建议，因此，即使在还存在更多收集到的结果的情况下，在结果列表中指定有 NObjects 的结果或多或少的会被返回，甚至空列表也可能被返回。

> **警告**
>
> 在同一个事务中对表做任何修改后使用 mnesia:select/[1|2|3|4]存在严重的性能损失。即在同一个事务内，在 mnesia:select 之前避免使用 mnesia:write/1 或 mnesia:delete/1。

如果键属性被绑定在模式内，匹配操作将非常高效。但是，如果键属性在模式中作为'_'或'$1'给定，那么整个雇员表都必须被搜索来匹配记录。因此如果表很大，将成为很耗时的操作。不过，这个问题在使用 mnesia:match_object 的情况下可用索引来补救。

QLC 查询也能用于搜索 Mnesia 表。通过使用 mnesia:table/[1|2]作为在 QLC 查询内部的生成器你就让这个查询作用于 Mnesia 表上。Mnesia 对 mnesia:table/[1|2]指定的选项为{lock, Lock}、{n_objects,Integer} 和{traverse, SelMethod}。lock 选项指定 Mnesia 是否应该请求一个读或写锁，n_objects 指定在每个部分（chunk）应该返回多少结果给 QLC。最后一个选项是 traverse，其指定哪一个函数 mnesia 应该用来遍历表，默认用 select。但对 mnesia:table/2 使用{traverse, {select, MatchSpecification}}作为选项用户能够指定属于自己的表视图。

如果没有指定选项，将会请求一个读锁，每部分返回 100 个结果，select 被用于遍历表，即：

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

函数 mnesia:all_keys(Tab)表中的全部键。

4.8 迭代

Mnesia 提供一组函数来迭代表里的所有记录：

```
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
```

这些函数以迭代方式将函数 Fun 应用于 Mnesia 表 Tab 中的每一条记录。Fun 有两个参数，第一个参数是来自于表中的一条记录，第二个参数是累加器，Fun 返回一个新的累加器。

首次调用 Fun 时 Acc0 是其第二个参数，下次调用 Fun 时，上次调用的返回值作为第二个参数，最后一次调用 Fun 所返回的项是函数 fold[lr]的返回值。

Foldl 与 foldr 之间的区别在于对 ordered_set 类型的表存取顺序不同，对其它类型的表这些函数是等价的。

LockType 指定迭代将请求何种类型的锁，默认是读锁。如果在迭代时写入或删除记录，那么就应该请求写锁。

这些函数可用于在无法为 mnesia:match_object/3 写出约束条件时在表中查找记录，或者是在你要对特定的纪录执行一些动作的时候。

查找所有工资低于 10 的雇员的示例如下：

```
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).
```

对工资低于 10 的所有雇员增加工资到 10 并且返回所加工资的合计数：

```
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).
```

迭代函数能做很多好事，但将其应用于大表时需要注意其对性能和内存的影响。

只在有表副本的节点上调用这些迭代函数。如果表是在其它节点上，每次调用 Fun 函数对表进行存取都会产生大量的、不必要的网络流量。

Mnesia 提供了另外一些函数给用户用于对表做迭代处理。如果表不是 ordered_set 类型，就不会指定迭代的顺序。

```
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
```

first/last 和 next/prev 的顺序仅适用于 ordered_set 表，对其它类型的表这只是一些同义词。当在指定的键上到达表尾时，返回'$end_of_table'。

如果在遍历时写入或删除记录，使用 mnesia:fold[lr]/4 并设置写锁。如使用 first 或 next 则用 mnesia:write_lock_table/1。

在事务上下文里写入或删除记录时每次都会创建一个本地副本，因此，在一个大表中修改每条记录时会占用大量内存。Mnesia 会对在事务上下文中迭代时的写或删除记录做补偿，这可能降低性能。如果可能的话应尽量避免在同一个事务中在对表做迭代前写入或删除记录。

在脏上下文里，如 sync_dirty 或 async_dirty，修改记录
存储本地副本，而是分别更新每条记录。如果表在其它节点有副本将产生大量网络流量并且还会带来脏操作全部的固有缺陷。特别是对 mnesia:first/1 和 mnesia:next/2 命令，上述 dirty_first 和 dirty_next 也有同样的问题，所以，不应该在迭代时写入表。

5、其它 Mnesia 特性

本用户指南的前几章描述了如何启动 Mnesia 以及怎样构建 Mnesia 数据库。本章我们将描述构建分布式、容错的 Mnesia 数据库相关的高级特性。本章包括下列部分：

- 索引
- 分布和容错
- 表分片
- 本地内容表
- 无盘节点
- 更多的模式管理
- Mnesia 应用调试
- Mnesia 应用调试
- 原型
- 基于对象的 Mnesia 编程

5.1 索引

如果我们知道记录的键，那么数据检索和匹配能够非常高效。相反，如果不知道键，那么表里必须搜索表里的全部记录。表越大，耗用的时间越多。Mnesia 的索引能力就是解决这个问题的。

下面两个函数用于对现有的表做索引：

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

这两个函数对 AttributeName 定义的字段增加或删除索引。以公司数据库的雇员表(employee, {emp_no, name, salary, sex, phone, room_no}为例，来看一下如何为表增加索引。如下所示，函数加一个索引到字段 salary：

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

Mnesia 使用下列 3 个基于索引项的函数在数据库中检索和匹配记录。

- `mnesia:index_read(Tab, SecondaryKey, AttributeName) -> transaction abort | RecordList`，通过在索引里查询次键（SecondaryKey）发现主键，以此来避免穷举搜索全表。
- `mnesia:index_match_object(Pattern, AttributeName) -> transaction abort | RecordList`，通过模式 Pattern 的 AttributeName 字段查找加了索引的次键，通过次键找到主键，以避免穷举搜索全表。 次键必须被绑定。

- `mnesia:match_object(Pattern) -> transaction abort | RecordList`，使用索引来避免穷举搜索全表。与上面的函数不同，只要次键被绑定此函数可使用任何索引。

这些函数的进一步描述和示例见第 4 章：模式匹配。

5.2 分布和容错

Mnesia 是一个分布式、容错的数据库管理系统，可以用多种方式在 Erlang 节点上复制表，Mnesia 程序员只需要在程序代码中指定不同的表名而不必知道这些表在哪里。 这就是“位置透明”，一个很重要的概念。特别是：

- 无论数据位于何处程序都可以工作，数据存放在远程节点与本地节点没有什么不同。
  注意：如果数据位于远程节点，程序将会运行得慢一些。
- 数据库可以重新配置，表可以在节点之间移动，这些操作不影响用户程序。

我们在前面已经看到，每张表有一定数量的系统属性，如索引（index）和类型（type）。

创建表的时候即就指定了表的属性。例如，下列函数将创建一个有两个 RAM 副本的新表：

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

表可以有如下属性，属性的值是一个 Erlang 节点列表。

- `ram_copies`，节点列表的值是需要将表的副本存放在其内存（RAM）中的 Erlang 节点列表。理解当程序对内存副本执行写操作时不会执行磁盘操作是很重要的。因此，应该对内存副本做持久化处理，可选择下列方式来完成：
  1. `mnesia:dump_tables/1` 函数可用来将表的内存副本导入到磁盘上。
  2. 可使用上述函数对表的副本（内存或磁盘）做备份。
- `disc_copies`，节点列表的值是需要将表的副本既存放在其内存中，也存放在其磁盘内的 Erlang 节点列表。对表的写操作将既写入表的内存副本中，也写到磁盘的拷贝内。
- `disc_only_copies`. 节点列表的值是需要将表的副本仅存放在其磁盘内的 Erlang 节点列表。其主要缺点是存取速度，主要优点是不占内存。

对现存的表设置或修改属性是可能的，参见第 3 章：定义模式。

使用一个以上的表副本有两个理由：容错和速度。值得注意的是，表复制对这两个系统需求都提供了解决方案。

如果我们有两个激活的表副本，其中一个失败，全部信息仍然是可用的。对许多应用来说，这是非常重要的优点。此外，如果表副本存在于两个指定的节点上，则这两个节点上的应用可以直接从表中读数据而不需要通过网络。 网络操作与本地操作相比既慢且耗费资源。

对于频繁读数据而很少写数据的分布式应用，为了在本地节点实现快速读操作，创建表副本很有利。主要的不利是复制会增加写数据的时间。如果表有两个副本，每次写操作都必须存取两个表副本。由于这两个写操作中的一个必须是网络操作，因此，执行写操作时复制表比不复制表要付出更高昂的代价。

5.3 表分片

5.3.1 概念

表分片概念的引入是为了对付非常大的表。其思想是将一个表分为若干个更容易处理的片断，每个片断都作为第一类 Mnesia 表来实现，就像其它表一样，可以被复制、加索引等等，但不能是 local_content 表和激活 snmp 连接。

为了从分片表中存取一条记录，Mnesia 必须确定这条真实的纪录属于哪一个片断。这由 mnesia_frag 模块来完成，通过 mnesia_access 的回调行为实现。 请读一读关于 mnesia:activity/4 的文档，看看 mnesia_frag 是怎样作为一个 mnesia_access 的回调模块被使用。

在每条记录存取时，mnesia_frag 首先根据此记录的键计算一个哈希（hash）值；其次，根据哈希值确定分片表的名字；最终，由同一个函数像对待非分片表一样执行实际的表存取。在事先不知道键的情况下，会搜索所有片断来匹配记录。注意：ordered_set 表的记录以每个片断排序，并且 select 和 math_object 函数返回结果的顺序是不确定的。

下列代码示例怎样将一个现存的 Mnesia 表转换为一个分片表以及怎样在后来加入更多的片断：

```
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>
```

5.3.2 分片属性

表属性 frag_properties 可用 mnesia:table_info(Tab, frag_properties)读出。分片属性是一个二元标签元组列表。这个列表默认为空表，但如果这个列表非空，将触发 Mnesia 将表作为片断来看待。分片属性是：

`{n_fragments, Int}`

`n_fragments` 控制这个表当前有多少个片断。这个属性可在创建表的时候设置，也可以在后来用`{add_frag, NodesOrDist}` 或 `del_frag` 改变。`n_fragments` 默认为 `1`。

`{node_pool, List}`

节点池包含一个节点列表，可以在创建表的时候显式指定，也可以在后来用`{add_node, Node}`或`{del_node, Node}`来改变。在创建表的时候 Mnesia 尝试将每个片断的副本均匀地分布到节点池中的所有节点，期望所有节点都有同样数量的副本来结束。

`node_pool` 默认从 `mnesia:system_info(db_nodes)`返回值。

{n_ram_copies, Int}

控制每个片断应该有多少 ram_copies 副本。这个属性可在创建表时显式指定。 默认值是 0， 但如果 n_disc_copies 和 n_disc_only_copies 也是 0，则 n_ram_copies 将默认置为 1。

{n_disc_copies, Int}

控制每个片断应该有多少 disc_copies 副本。这个属性可在创建表时显式指定。 默认值是 0。

{n_disc_only_copies, Int}

控制每个片断应该有多少 disc_only_copies 副本。这个属性可在创建表时显式指定。默认值是 0。

{foreign_key, ForeignKey}

ForeignKey （外键）可以是原子 undefined 或是元组{ForeignTab, Attr}， 此处 Attr 是一个应该在其它分片表上说明为 ForeignTab 的属性。Mnesia 将确保在此表中以及在外部表中的的片断数量总是相同的。在增加或者删除片断时， Mnesia 将自动传播操作到所有通过外键引用此表的所有分片表。使用 Attr 字段的值来代替使用记录键来确定应存取哪一个片断。此特性使得在同一节点上的不同表中的记录能够自动地驻留。

foreign_key 默认为 undefined。但如果外键被设置为某些东西，其将引起其它分段特性的默认值与外部表的实际分段特性的默认值是一样的。

{hash_module, Atom}

使用一个替换的哈希模块。此模块必须实现 mnesia_frag_hash 回调函数的行为 (参看参考手册)。 这个属性可在创建表的时候显式设置。默认为 mnesia_frag_hash。

在用户定义哈希模块被引入之前建立的旧表使用 e mnesia_frag_old_hash 模块来保证向后兼容。 mnesia_frag_old_hash 仍然使用较弱的 erlang:hash/1 函数。

{hash_state, Term}

使用一个通用哈希模块的特定参数。 这个属性可在创建表的时候显式设置。默认为 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>
```

5.3.3 分片表的管理

函数 mnesia:change_table_frag(Tab, Change)用于重新配置分片表。Change 参数应为下列值之一：

`{activate, FragProps}`

激活一个现存表的分片属性，FragProps 应为 `{node_pool, Nodes}`或是空。

`deactivate`

解除表的分片属性， 片断的数量必须是 1。没有其它表在其外键中引用此表。

`{add_frag, NodesOrDist}`

加一个新的片断到分片表。在老的片断中的全部记录将被重新处理并且其中一半的记录将被移送到新（最后的）片断。 所有通过外键引用此表的其它分片表将自动获得新的片断， 其记录也将用与主表相同的方式动态重新处理。

NodesOrDist 参数可以是一个节点列表或者是来自于 `mnesia:table_info(Tab, frag_dist)`函数的结果。NodesOrDist 参数被看作是一个根据新副本首先进入的主机为最优来排序的有序节点列表。新片断将获得与第一个片断同样数量的副本(看 `n_ram_copies`， `n_disc_copies` 和 `n_disc_only_copies`)。NodesOrDist 列表必须至少包含一个需要为每个副本分配的单元。

`del_frag`

从分片表删除一个片断。在最后这个片断的所有记录将被移到其它片断之一。所有通过其外键引用此表的其它分片表将自动丢失其最后的片断，其记录也将用与主表相同的方式动态重新处理。

`{add_node, Node}`

增加一个新节点到节点池 `node_pool`。 新的节点池将影响从函数 `mnesia:table_info(Tab, frag_dist)`返回的列表。

`{del_node, Node}`

从节点池 `node_pool` 删除一个节点。新的节点池将影响从函数 `mnesia:table_info(Tab, frag_dist)`返回的列表。

5.3.4 现有函数的扩充

函数 mnesia:create_table/2 通过设置表属性 frag_properties 到一些合适的值创建一个全新的分片表。

函数 mnesia:delete_table/1 可用来删除一个包括其全部片断在内的分片表。条件是没有任何其它的分片表在其外键中引用这个表。

函数 mnesia:table_info/2 现在能理解 frag_properties 项。

如果在使用 mnesia_frag 模块的作业上下文中调用函数 mnesia:table_info/2，可获得一些新项目的信息：

`base_table`

分片表的名字

`n_fragments`

片断的实际数量

`node_pool`

节点池

`n_ram_copies`
`n_disc_copies`
`n_disc_only_copies`

存储类型 `ram_copies`，`disc_copies` 和 `disc_only_copies` 各自的副本数量。源自于第一个片断的实际值是动态的。实际值通过计算每种存储类型的每个副本的数量来确定，当需要计算实际值的时候（如在增加一个新片断时），第一个片断的类型将作为计算的依据。这表明当函数 `mnesia:add_table_copy/3`， `mnesia:del_table_copy/2` 和 `mnesia:change_table_copy_type/2` 被应用于第一个片断时，将会影响到 `n_ram_copies`， `n_disc_copies` 和 `n_disc_only_copies` 的设置。

`foreign_key`

外键

`foreigners`

通过其外键引用该表的所有其它表。

`frag_names`

所有片断的名字。

frag_dist

一个按 Count 增序排列的{Node, Count} 元组的有序列表。Count 是分片表副本所在主机节点 Node 的总数。这个列表至少包含了节点池 node_pool 中的全部节点。 不属于节点池 node_pool 的节点即使其 Count 值较低也将被放在列表的最后。

frag_size

{Name, Size}元组列表，Name 是片断名称，Size 是其包含了多少条记录。

frag_memory

{Name, Memory}元组列表， Name 是片断名称， Memory 是被占用内存数。

size

所有片断的总尺寸

memory

所有片断的总内存

5.3.5 负载均衡

有数个算法可用于将分片表的记录均匀地分布给节点池。但没有一个是最好的，完全取决于应用的需要。下面是一些需要注意的情况：

节点的永久改变（permanent change of nodes）。当一个新的永久性的数据库节点（db_node）加入或退出，这或许是改变节点池的节点数量以及在新的节点池重新均匀分布副本的时候。这或许也是在重新分布副本之前增加或删除片断的时候。

尺寸/内存 阈值（size/memory threshold）。当一个分片表（或单个片断）的总尺寸或对占用的总内存超过某些应用规定的阈值，或许就是动态增加一个新的片断以获得较好的记录分布的时候。

节点暂时宕机（temporary node down）。当一个节点暂时宕机，或许要为新副本补偿一些片断使得冗余保持在希望的水平。当此节点恢复后，或许需要将多余的副本移除。

过载阈值（overload threshold）。当某些节点的负载超过应用规定的阈值，可增加或移动一些片断副本到负载较轻的节点上。如果表有外键与其它表关联需要特别小心。为了防止出现性能严重下降，必须为所有这些相关的表做重新分布。

使用 mnesia:change_table_frag/2 增加新片断并且应用通常的模式操作函数（如 mnesia:add_table_copy/3，mnesia:del_table_copy/2 和 mnesia:change_table_copy_type/2）在每个片断执行实际的重分布。

5.4 本地内容表

复制到不同节点上的表在这些节点上的内容都是一样的。但有时候让不同节点上的表有不同的内容是有用的。

如果在创建表的时候指定属性{local_content, true}，这个表将留在我们指定其应该存在的节点上，且对其的写操作仅能在本地执行。

此外，但在启动时被初始化时只能在本地进行而且表的内容不能复制到其它节点。

5.5 无盘节点

Mnesia 可以在无盘节点上运行。当然，在这样的节点上不可能有 disc_copie 或 disc_only_copies 类型的表副本。特别麻烦的是模式（schema）表，因为 Mnesia 需要模式表来初始化自身。

如同其它表一样，模式表可以驻留在一个或多个节点上。模式表的存储类型可以是 disc_copies 或 ram_copies（不能是 disc_only_copies）。在启动的时候，Mnesia 使用其模式表来确定应该与哪些节点尝试建立联系。如果其它节点已经启动，启动的节点将其表定义与其它节点带来的表定义合并。这也应用于模式表自身的定义。应用参数 extra_db_nodes 包含一个 Mnesia 除了在其模式表中找到的节点之外也应该建立联系的节点列表。其默认值为空列表[ ]。

因此，当无盘节点需要从一个在网络上的远程节点找到模式定义时，我们需要通过应用参数-mnesia extra_db_nodes NodeList 提供这个信息。没有这个配置参数集，Mnesia 将作为单节点系统启动。也有可能在 Mnesia 启动后用 mnesia:change_config/2 赋值给'extra_db_nodes'强制建立连接，即 mnesia:change_config (extra_db_nodes, NodeList)。

应用参数 sschema_location 控制 Mnesia 将搜索何处来获得模式表。这个参数可以是下列原子之一：

Disc

强制磁盘。假定模式在 Mnesia 目录里。如果找不到模式，Mneia 拒绝启动。

Ram

强制内存。模式仅驻留在内存。在启动时生成一个极小的新模式。默认模式仅包含模式表的定义并且仅驻留在本地节点上。由于在默认模式中不能发现其它节点，必须使用配置参数 extra_db_nodes 让这个节点与其它节点共享其表定义。（extra_db_nodes 也可用于有盘节点上）。

opt_disc

可选磁盘。模式既可以驻留在磁盘，也可以驻留在内存。如果在磁盘上找到模式，Mnesia 作为一个有盘节点启动（模式表的存储类型是 disc_copie）。如果不能在磁盘中找到模式，Mnesia 作为一个无盘节点启动（模式表的存储类型是 ram_copies）。这个应用参数的默认值是 opt_disc。

当 schema_location 被设置为 opt_disc 时，函数 mnesia:change_table_copy_type/3 可用来改变模式的存储类型。如下所示：

```
1> mnesia:start().
 ok
```

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

如果调用 mnesia:start 不能在磁盘上找到任何模式，Mnesia 作为一个无盘节点启动，然后改变其道一个可用磁盘在本地存储模式的节点上。

5.6 更多的模式管理

一个 Mnesia 系统可以增加或删除节点。这是通过增加一个模式表的拷贝到这些节点上来实现的。

函数 mnesia:add_table_copy/3 和 mnesia:del_table_copy/2 可用于增加或删除模式表的副本[1]。增加一个节点到模式将被复制的节点列表中将影响两件事。首先，允许其它表可以复制到这个节点上；其次，在有盘节点启动时促使 Mnesia 尝试与此节点建立联系。

函数调用 mnesia:del_table_copy(schema, mynode@host)从 Mnesia 系统中删除'mynode@host'节点[2]，如果 Mnesia 正在'mynode@host'上运行这个调用将会失败。其它 Mnesia 节点将不再尝试与这个节点连接。注意：如果在'mynode@host'节点上有一个磁盘驻留模式，应该将整个 mnesia 目录删除。可用 mnesia:delete_schema/1 来完成。如果 mnesia 再次在'mynode@host'节点上启动并且目录还没有被清除，mnesia 的行为是不确定的。

如果模式的存储类型是 ram_copies，即我们有一个无盘节点，Mnesia 将不会在这个特定节点上使用磁盘。通过改变模式表的存储类型为 disc_copies 即可使用磁盘。

可显式的用 mnesia:create_schema/1 或隐式的在没有磁盘驻留模式的情况下启动 Mnesia 来创建新模式。无论何时，表（包括模式表）在创建是都会被赋予一个属于其自身的、唯一的 cookie。模式表不能用 mnesia:create_table/2 作为普通表来创建。

启动时不同的 Mnesia 节点互相连接，随后彼此之间交换并且合并表定义。在合并过程中 Mnesia 执行一个健康测试（sanity test）以确保相互之间的表定义是一致的。如果一个表存在于多个节点上，其 cookie 必须是同样的，否则 Mnesia 将关闭 cookie 不同的节点。如果一个表在两个没有相互连接的节点上被单独创建就会出现这种不幸的情况。要解决这个问题，必须删除其中一个表（尽管名称相同，但由于 cookie 不同，我们视其为两个不同的表）。

合并不同版本的模式表不总是要求有相同的 cookie。如果模式表的存储类型是 disc_copies，cookie 是不可改变的，所有其它数据库节点必须有相同的 cookie。当模式作为类型 ram_copies 存储时，

---

1 译者注：只能使用 ram_copies 类型来复制 schema 表，要改变存储类型可用 change_table_copy_type/3。

2 译者注：此时 mynode@host 上的 mnesia 必须是停止的。

其 cookie 能够被来自于其它节点（ram_copies 或 disc_copies）的 cookie 替换掉。cookie 替换（在合并模式表定义期间）在每次 RAM 节点连接到其它节点时被执行。

mnesia:system_info(schema_location)和 mnesia:system_info(extra_db_nodes)可用于确定 schema_location 和 extra_db_nodes 的实际值。mnesia:system_info(use_dir)可用于确定 Mnesia 实际使用的 Mnesia 目录。use_dir 甚至可以在 Mnesia 启动前确定。函数 mnesia:info/0 现在甚至可以在 Mnesia 启动前打印出一些系统信息。当 Mnesia 启动后这个函数打印出更多的系统信息。

更新表定义的事务要求 Mnesia 在所有模式存储类型为 disc_copies 的节点上启动。在这些节点上的全部表副本也必须被装载。这些有效规则可以有例外，在全部有盘节点没有启动之前，可以建表和增加新副本。只要其它副本中有一个被激活，在其它表副本没有全部装载的情况下也可增加新的副本。

5.7 Mnesia 事件处理

在不同情况下，Mnesia 将生成系统事件和表事件这样两类事件。

用户进程能够订阅 Mnesia 生成的事件。我们有下列两个函数：

mnesia:subscribe(Event-Category)

确保所有 Event-Category 类型事件的副本会发送给调用进程。

mnesia:unsubscribe(Event-Category)

删除对 Event-Category 类型事件的订阅。

Event-Category 可以是原子 system 或元组{{table, Tab, simple}或{table, Tab, detailed}二者之一。旧的事件类别{table, Tab}与事件类别{table, Tab, simple}是一样的。订阅函数激活对事件的订阅。对函数 mnesia:subscribe/1 求值即将事件作为消息发送给进程。系统事件的语法是{mnesia_system_event, Event}，表事件的语法是{mnesia_table_event, Event}。系统事件和表事件的含义描述如下：

所有的系统事件通过 Mnesia 的通用事件处理器（gen_event handler）来订阅，默认的通用事件处理器是 mnesia_event，但可通过应用参数 event_module 来改变。这个参数的值必须是一个模块名，该模块是使用标准库（STDLIB）的 gen_event 模块来实现的完整的事件处理模块。mnesia:system_info(subscribers) 和 mnesia:table_info(Tab, subscribers)用来确定哪些进程订阅了何种事件。

5.7.1 系统事件

系统事件的细节描述如下：

```
{mnesia_up, Node}
```

Mnesia 已经在一个节点上启动。Node 是此节点的名字。 在默认情况下此事件被忽略。

{mnesia_down, Node}

Mnesia 已经在一个节点上停止。Node 是此节点的名字。 在默认情况下此事件被忽略。

{mnesia_checkpoint_activated, Checkpoint}

一个名为 Checkpoint 的检查点已经被激活并且当前节点涉及到此检查点。检查点可用 mnesia:activate_checkpoint/1 函数显式激活，或通过备份、增加一个表副本、节点之间内部传送数据等隐式激活。在默认情况下此事件被忽略。

{mnesia_checkpoint_deactivated, Checkpoint}

一个名为 Checkpoint 的检查点已经被解除并且当前节点涉及到此检查点。检查点可用 mnesia:deactivate/1 函数显式解除，或在涉及到此检查点的表的最后一个副本变得不可用（如节点宕机）时隐式解除。在默认情况下此事件被忽略。

{mnesia_overload, Details}

当前节点上的 Mnesia 出现并且过载订阅者应该采取行动。

典型的过载情况是应用程序对磁盘驻留表执行的更新超过了 Mnesia 的处理能力。无视这种类型的过载将导致磁盘空间耗尽的情况出现（不管存在磁盘上的文件尺寸有多大)。

每次更新被添加到事务日志并且有时（取决于是如何配置的）会转储到表文件。表文件的存储比事务日志的存储更紧凑，特别是在某些记录被反复更新的情况下。如果在前面的更新转储尚未结束之前，事务日志转储的阈值达到就会触发过载事件。

过载的另一种典型情况是事务管理器不能在对磁盘驻留表执行更新的应用程序的相同空间里提交事务。这使得事务管理器的消息队列持续增长直到内存耗尽或负载能力下降。

脏更新时也可能出现同样的问题。在当前节点发现的本地过载有可能是其它节点所引起的，应用程序处理驻留在其它节点上的表（副本或非副本）时可能会引起重负载。在默认情况下此事件报告给错误日志（error_logger）。

{inconsistent_database, Context, Node}

Mnesia 关注数据库的潜在不一致并且给其应用程序一个从矛盾中恢复的机会，即安装一个一致的备份作为回滚，然后重新启动系统，或者从 mnesia:system_info(db_nodes) 中找出主节点（MasterNode），调用 mnesia:set_master_node([MasterNode])。在默认情况下一个错误将报告给错误日志（error_logger）。

{mnesia_fatal, Format, Args, BinaryCore}

Mnesia 遇到致命错误并且将（在一个短时间内）被终止。 关于此致命错误的原因在 Format 和 Args 中解释，也可作为 io:format/2 的输入或送给错误日志。在默认情况下其将被送交错误日志。BinaryCore 是一个包含在此错误出现时 Mnesia 内部状态概要的二进制数。在默认情况下此二进制数被写入当前目录下的一个有唯一名字的文件中。在 RAM 节点此 core 被忽略。

{mnesia_info, Format, Args}

Mnesia 发现一些在调试系统时或许感兴趣的东西。这些东西在 Format 和 Args 中解释，也可作为 io:format/2 的输入或送给错误日志。 在默认情况下这个事件被 io:format/2 打印出来。

{mnesia_error, Format, Args}

Mnesia 遇到一个错误。关于此错误的原因在 Format 和 Args 中解释，也可作为 io:format/2 的输入或送给错误日志。在默认情况下其将被送交错误日志。

{mnesia_user, Event}

一个应用程序调用了 mnesia:report_event(Event)。Event 可以是任何 Erlang 数据结构。在跟踪一个 Mnesia 应用系统时获得 Mnesia 自身与应用相关的事件以及给出关于应用上下文的信息是很有用的。在应用程序执行期间，无论是在应用启动一个新的、高要求的 Mnesia 作业时，还是在进入一个新的、令人感兴趣的阶段时使用 mnesia:report_event/1 都会是一个好主意。

5.7.2 表事件

另一种事件类别是表事件，这类事件与表的更新有关。表事件分为简单和详细两种类类型。

简单表事件是一些类似{Oper, Record, ActivityId}的元组。这里 Oper 是被执行的操作，Record 是操作涉及到的记录，ActivityId 是执行操作的事务标识。注意：这里纪录名是表名，即使是 record_name 被另行设置过。可能出现的几种与表相关的事件是：

{write, NewRecord, ActivityId}

写入一条新纪录。NewRecord 包含记录的新值。

{delete_object, OldRecord, ActivityId}

一条记录可能被 mnesia:delete_object/1 删除。 OldRecord 包含的是被应用程序作为参数规定的旧记录的值。 注意：如果表是 bag 类型，有相同键的其它记录可被保留。

{delete, {Tab, Key}, ActivityId}

一条或多条记录可能被删除。在表 Tab 中键为 Key 的所有记录被删除。

详细表事件是一些类似{Oper, Table, Data, [OldRecs], ActivityId}的元组。这里 Oper 是被执行的操作，Table 是操作涉及到的表，Data 是被写入记录的 oid 或是被删除的记录，OldRecs 操作发生之前的内容，ActivityId 是执行操作的事务标识。可能出现的几种与表相关的事件是：

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

写入一条新记录。NewRecord 包含记录的新值，OldRecords 包含在操作被执行前的记录，注意：新内容依赖于表的类型。

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

What（元组{Table, Key}或记录{RecordName, Key, ...}）指定的记录被删除。注意：新内容依赖于表的类型。

5.8 调试 Mnesia 应用

Mnesia 应用程序调试困难的原因主要是事物和表加载的机制难以理解。此外，嵌套事务的语义也令人困惑。

我们可以通过调用下列函数来设置 Mnesia 的调试级别：

- mnesia:set_debug_level(Level)

这里的参数 Level 是：

none
: 完全没有跟踪输出。这是默认的。

verbose
: 激活对重要调试事件的跟踪。这些调试事件将产生{mnesia_info, Format, Args}系统事件。进程可用 mnesia:subscribe/1 订阅这些事件。这些事件都要发送给 Mnesia 的事件处理器。

debug
: 激活全部 verbose 级别的事件外加对全部调试事件的跟踪。这些调试事件将产生{mnesia_info, Format, Args}系统事件。进程可用 mnesia:subscribe/1 订阅这些事件。这些事件都要发送给 Mnesia 的事件处理器。在这个调试级别 Mnesia 的事件处理器启动对模式表更新的订阅。

trace
: 激活调试级别的全部事件。在这个调试级别 Mnesia 的事件处理器启动对全部 Mnesia 表更新的订阅。 这个级别会产生大量事件，仅用于调试小型的玩具系统。

false
: none 的别名。

true
: debug 的别名。

下列代码在 Erlang 系统启动时通过应用程序参数的方式设置 Mnesia 自身的调试级别。同时，在初始化启 动阶段打 开Mnesia 调试：

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

5.9 Mnesia 里的并发进程

Erlang 系统的并发程序设计是另一本书的主题。不过，下面这些允许并发进程在 Mnesia 系统中存在的特性还是值得引起注意的。

在一个事务中能够调用一组函数或进程。一个事务可以包括若干对来自数据库管理系统的数据进行读、写或删除的语句。像这样的事务能够大批量并发运行而不需要程序员显式同步对这些数据进行操作的进程。所有通过事务系统对数据库进行存取的程序可当作是唯一对数据进行存取的程序一样来编写。这是非常值得拥有的特性，因为所有的同步都是由事务处理器来处理。如果一个程序读或写数据，系统确保没有其它的程序试图同时操作同样的数据。

可以用不同的方法移动、删除表或者重新配置表的布局。重要的是当表在重新配置时用户程序能够继续使用这个表。例如，可以在移动一个表的同时对这个表执行写操作。这对许多要求提供不间断服务的应用程序是非常重要的。获得更多信息可参见第 4 章：事务和其它存取上下文。

5.10 原型

如果并且当我们愿意启动和操控 Mnesia 的时候， 将定义和数据写到一个普通文本文件会比较容易一些。起初，既没有表也没有数据存在，也不知道需要什么样的表。在原型的初始阶段，把所有数据写到一个文件，处理这个文件并且将这个文件中的数据插入到数据库是很明智的。从一个文本文件读数据来初始化 Mnesia 是可能的。我们有下面这两个函数可以处理文本文件：

- `mnesia:load_textfile(Filename)`。 这个函数加载一系列可在文件中找到的表定义和数据到 Mnesia。此函数也启动 Mnesia 并且有可能创建一个新模式。此函数仅在本地节点操作。
- `mnesia:dump_to_textfile(Filename)` 转储一个 Mnesia 系统的全部本地表到一个能够被普通文本编辑器编辑并且以后可以重新加载的文本文件中。

这些函数与 Mnesia 的普通存储和加载函数相比当然是非常慢，但主要用于小型实验系统和初始的原型，其好处是非常容易使用。

文本文件的格式为：

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

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

`Options` 是一个符合 `mnesia:create_table/2` 选项要求的`{Key,Value}`元组列表。

假如我们开始玩一个关于健康食品的小型数据库，可输入下列数据到文件 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}.
```

下列 Erlang shell 会话显示如何加载 fruits 数据库：

```
% 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>
```

这里我们能够看到怎样从一个规范的文本文件来初始化数据库管理系统。

5.11 Mnesia 基于对象的编程

第 2 章介绍的公司数据库有三个存储记录（employee, dept, project）的表和三个存储关系(manager, at_dep, in_proj)的表。这是一个规范化的数据模型，与非规范化的数据模型相比有许多优点。

在一个规范化的数据库中做通用搜索很有效。在一个规范化的数据模型上执行某些操作也会更容易。如下例所示，我们能够容易的删除一个项目（project）：

```
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.
```

事实上，数据模型很少能完全规范化。一个实际可行的、用以替代规范化数据模型的数据模型甚至不符合第一范式。Mnesia 非常适合如电信这样的应用，因为它可以很容易地用非常灵活的方式组织数据。Mnesia 数据库是以表的集合组织起来的。每个表用行/对象/记录来填充。使得 Mnesia 与众不同的是其字段（field），一个在记录中的单独字段能够包含任何类型的复合数据结构。在一条记录中的一个字段就能够包含列表、元组、函数甚至记录代码。

许多电信应用对查找特定类型的记录有时间方面的特殊要求。如果我们的公司数据库是电信系统的一部分，对雇员以及雇员参与其工作的项目列表的查找时间应该是最少的。如果是这种情况，我们或许选择一个彻底不同的数据模型，这个数据模型没有直接关系。我们仅有记录自身，不同的记录既能包含对其它记录的直接引用，也能包含不是 Mnesia 模式一部分的其它记录。

我们能够创建下列记录定义：

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


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

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

一条描述一个雇员的记录或许看起来像这样：

```
    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},
```

这个模型仅有三个不同的表，雇员记录包含对其它记录的引用。在记录用我们有下列引用：

- `'B/SFR'`引用一条部门记录。
- `[erlang, mnesia, otp]`。这是一个直接引用三个不同 `projects` 记录的列表。
- `114872`。这是对其他雇员记录的引用。

我们也能用 Mnesia 记录标识符({Tab, Key})作为引用。在这种情况下，dept 属性的值被设为{dept, 'B/SFR'}来代替'B/SFR'。

我们的公司数据库使用这个数据模型在执行某些操作时会比用规范化的数据模型快得多。另一方面，某些其它操作变得更加复杂，尤其是确保记录没有包含指向其它不存在或者是被删除的记录的、不确定的指针变得很困难。

下列代码示例用非规范化的数据模型做一次搜索，找出在部门 Dep 以及工资高于 Salary 的所有雇员：

```
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).
```

这段代码不仅容易编写和理解，而且执行也非常之快。

我们用非规范化的数据模型来代替规范化的数据模型会很容易的表现出执行速度更快的代码实例。这样做的主要理由是需要的表会更少。因此，我们能更容易的在联合（join)操作中从不同的表组合数据。在上述示例中，get_emps/2 函数从联合操作转换为由一个选择和一个投影组合的、对一个单独的表所做的简单查询。

6 Mnesia 系统信息

6.1 数据库配置数据

下列两个函数能够用于检索系统信息。其详细描述见参考手册。

- `mnesia:table_info(Tab, Key) ->Info | exit({aborted, Reason})`. 返回有关表的信息，如表的当前尺寸、位于哪些节点等等。
- `mnesia:system_info(Key) -> Info | exit({aborted, Reason})`. 返回有关 Mnesia 系统的信息。例如， 事务统计、数据库节点、配置参数等。

6.2 内核转储(Core Dumps)

如果 Mnesia 故障，系统信息被转储到名为 MnesiaCore.Node.When 的文件。包含在这个文件里的系统信息类型也可以通过函数 mnesia_lib:coredump()产生。如果 Mnesia 系统行为古怪，建议在 bug 报告中包括 Mnesia core dump 文件。

6.3 转储表

定义为 ram_copies 类型的表仅能存储在内存中。但可以按一定的时间间隔或在系统关机之前将表转储到磁盘上。函数 mnesia:dump_tables(TabList)将 RAM 表的全部副本转储到磁盘上。转储到磁盘的表能够被存取。可被转储表的所有副本必须有存储类型 ram_copies。
表的内容放在磁盘上的一个后缀为.DCD 的文件里。当 Mnesia 系统启动时，将从.DCD 文件加载数据初始化 RAM 表。

6.4 检查点

对一个跨越多个表的事务，检查点是此事务的一致性状态。当一个检查点被激活时，系统将记住表集合的当前内容。检查点保留表的事务一致性状态，当检查点被激活时允许读和更新表。检查点的典型应用是备份表到外部介质，但也可在 Mnesia 内部用于其它目的。每一个检查点都是独立的，多个检查点可同时涉及同一个表。

为了执行判断应用程序，每个表在一个检查点保持器中保留其过去的内容，知道检查点相关的处理开销是很重要的。在最坏的情况下，检查点保持器比表本身耗用的内存更多。在那些附有检查点保持器的表所在的节点上，每次更新也要慢一些。

对每个表可以选择是对其所有的表副本都附一个检查点保持器还是仅对其中的一个副本附一个检查点保持器。每个表只要一个检查点保持器耗用的内存较少，但在节点崩溃时会很脆弱。每个表有数个冗余的检查点保持器时只要至少有一个存活，就可以使得检查点得以幸存。

可以用函数 mnesia:deactivate_checkpoint(Name)显式解除检查点，这里 Name 是一个活动检查点的名字。如果成功，函数返回 ok，如果出现错误，返回{error, Reason}。在一个检查点内，所有的表都必须至少附有一个检查点保持器。只要有一个表没有检查点保持器，这个检查点就会被 Mnesia 自动解除，在节点宕机或副本被删除时就有可能出现这种情况。使用下面描述的 min 和 max 参数来控制检查点保持器的冗余度。

使用用函数 mnesia:activate_checkpoint(Args)来激活检查点，这里 Args 是下列元组的列表：

- `{name,Name}`. `Name` 指定一个检查点的临时名字。当检查点被解除时，此名字可以被复用。如果没有指定名字，将自动生成一个名字。
- `{max,MaxTabs}`. `MaxTabs` 是检查点所包括的数据库表的列表。默认是`[]`(一个空表)。对这些表将有最大冗余。当应用程序更新主表时，表中旧的内容将在检查点保持器中得到保留。如果表有多个副本，检查点将提供更多的容错能力。在使用模式操作函数 `mnesia:add_table_copy/3` 增加新的副本时，其也将被附加到本地的检查点保持器。
- `{min,MinTabs}`. `MinTabs` 是应该被包括进检查点的数据库表的列表。默认是`[]`。对这些表将有最小冗余，每个表只有单个检查点保持器， 更适宜在本地节点上。
- `{allow_remote,Bool}`. `False` 意味着全部检查点保持器必须是本地的。如果表没有驻留在本地，检查点不能被激活。`true` 允许检查点保持器可分配到任何节点上。默认是 `true`。
- `{ram_overrides_dump,Bool}`. 这个参数仅能用于 `ram_copies` 类型的表。 `Bool` 指定内存表状态是否应该优先于磁盘表状态。`True` 表明在内存中最后提交的纪录被包括在节点保持器中。这是应用程序存取的纪录。`false` 表明在磁盘.DAT 文件中的纪录被包括在节点保持器中。 这是启动时将被加载的纪录。默认是 `false`。

mnesia:activate_checkpoint(Args)返回下列值之一：

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

Name 是检查点的名字，Nodes 是检查点所在的节点。

可用下列函数来获得活动检查点的列表：

- `mnesia:system_info(checkpoints)`. 这个函数返回当前节点上所有活动的检查点。
- `mnesia:table_info(Tab,checkpoints)`. 这个函数返回指定表的活动检查点。

6.5 文件

这一节描述由 Mnesia 系统创建并且维护的内部文件。

6.5.1 启动文件

在第 3 章里我们详细说明了下列启动 Mnesia 的前置条件（参见第 3 章：启动 Mnesia）：

- 我们必须为我们的数据库启动一个 Erlang 会话并且指定一个 Mnesia 目录。
- 我们必须用函数 `mnesia:create_schema/1` 初始化一个数据库模式。

下列实例显示怎样执行这些任务：

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

2. 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
```

我们能够检查 Mnesia 目录看看创建了一些什么样的文件。键入下列命令：

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

响应显示文件 FALLBACK.BUP 被创建。这个文件被称为备份文件，包含一个初始模式。如果我们在 mnesia:create_schema/1 函数中指定了一个以上的节点，同一个备份文件将在所有这些节点上被创建。

3.继续启动 Mnesia:

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

我们现在能够在 Mnesia 目录中看到下列文件：

```
-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
```

在备份文件 FALLBACK.BUP 中的模式被用来生成文件 schema.DAT。由于除了模式我们没有其它驻留在磁盘上的表，所以没有创建其它数据文件。在成功的“物归原主”之后，FALLBACK.BUP 文件被移除。我们还看到一些用于 Mnesia 内部的文件。

4. 键入下列命令创建一个表：

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

我们现在能够在 Mnesia 目录中看到下列文件：

```
% 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
```

这里文件 foo.DCD 被创建。写到 foo 表中的所有数据最终将存储在这个文件中。

6.5.2 日志文件

当我们启动 Mnesia 的时候，一个名为 LATEST.LOG 的文件被创建并且放在数据库目录内。这个文件被 Mnesia 用来对基于磁盘的事务做日志。这包括所有在存储类型为 disc_copies 或 disc_only_copies 的表中至少写入一条记录的事务。还包括对模式本身所作的全部操作，如创建新表等。Mnesia 的不同实现的日志格式可能有变化。当前实现的 Mnesia 是标准库模块 disc_log。

日志文件会持续增长，因此需要定期转储。对于 Mnesia“转储日志文件”意味着执行在日志中列出的所有操作并且将记录存放到对应的.DAT、.DCD 和.DCL 文件中。例如，如果“写记录{foo, 4, elvis, 6}” 操作被列入日志，Mnesia 插入此操作到 foo.DCL 中，随后在 Mnesia 认为.DCL 文件已经变得太大时，再将数据移入.DCD 文件。如果日志很大，转储操作可能非常耗时。因此，理解 Mesia 系统在日志转储期间要持续运转是很重要的。

在默认状态下，只要日志中写入了 100 条记录或者过去了 3 分钟这两种情况之一出现，Mnesia 即转储日志。可用两个应用程序参数-mnesia dump_log_write_threshold WriteOperations 和-mnesia dump_log_time_threshold MilliSecs 来对此进行控制。

在日志被转储之前，文件 LATEST.LOG 改名为 PREVIOUS.LOG，并且创建一个新的 LATEST.LOG 文件。日志转储成功后，文件 PREVIOUS.LOG 被删除。

在启动时以及每当一个模式操作被执行时，也要转储日志。

6.5.3 数据文件

目录列表中还包括.DAT 文件。模式本身包含在 schema.DAT 文件中。DAT 文件是建有索引的文件，可用指定的键在这些文件中高效地插入和搜索记录。.DAT 文件被用于模式和 disc_only_copies 表。Mnesia 数据文件的当前实现是用标准库模块 dets，在 dets 文件上能被执行的所有操作都能够在 Mnesia 数据文件上被执行。例如，dets 的函数 dets:traverse/2 也能用于查看 Mnesia 数据文件的内容。但是，只有在 Mnesia 没有运行的时候才能这样做。因此，要查看我们的模式文件，我们能：

```
{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).
```

注意

参见参考手册，std_lib 获得有关 dets 的信息。

警告

数据文件必须用{repair, false}选项打开， 以确保文件不会被自动修复。 没有这个选项， 数据库或许会变得不一致， 因为 Mnesia 可能以为文件被正确关闭。参看参考手册关于配置参数 auto_repair 获得有关信息。

警告

在 Mnesia 正在运行的时候不得擅动数据文件，否则 Mnesia 的行为将难以预料。

disc_copies 表通过.DCL 和 DCD 文件被储存在磁盘上，这些文件是标准的磁盘日志文件。

6.6 在启动时加载表

为了让应用程序能够存取表，Mnesia 在启动时要加载表。有时 Mnesia 决定加载所有驻留在本地的表，有时直到 Mnesia 从其它节点获得表的拷贝之前，或许都不能对表进行存取。

只有在懂得 Mnesia 在与其它 Mnesia 节点失去联系时会做出何种反应才能理解 Mnesia 在启动时的这种行为。在此阶段，Mnesia 无法分辨出通讯失败与节点“正常”宕机。当这种情况发生时，Mnesia 将认为其它节点不再运行。但实际上只不过是节点之间通讯失败。

在这种情况下，简单地试着重启在失败的节点上正在对表进行存取操作的事务并且在日志文件中写入一条 mnesia_down 条目。

在启动时，需要注意到即使是那些没有 mnesia_down 条目的表也或许会有更新鲜的副本，这些副本在当前节点上的 Mnesia 被终止后或许被更新过。为了捕获最后的更新，需要从其它“新鲜”节点上传送表的一个拷贝过来。如果你很不幸，其它节点宕机，那么，在收到来自这些节点的一个表的新鲜拷贝使得表可被加载之前，你就只能等待。

在应用程序对表进行首次存取之前，应执行 mnesia:wait_for_tables(TabList, Timeout)以确保可以从本地节点对表进行存取。如果这个函数超时，应用程序或许可选择使用 mnesia:force_load_table(Tab)强行加载一个本地副本并且故意丢失本地节点宕机时在其它节点上已经执行过的全部更新。如果 Mnesia 在其它节点已经加载过表或者打算这样做，我们将从加载过表的节点复制表以避免不必要的矛盾。

警告

记住 用 `mnesia:force_load_table(Tab)`只能加载一个表， 而提交的事务或许更新了多个表，强行加载或许会导致这些表变得不一致。

可用 read_only （只读）或 read_write（读写）来定义表的读写状态（AccessMode），并且可用函数 mnesia:change_table_access_mode(Tab, AccessMode)在运行时切换。read_only 表和 local_content（本地内容）将总是从本地加载，因此不需要从其它节点复制。其它表主要从远程加载，或者是来自于表已被加载过的节点上的活动副本，或者按照已在运行的 Mnesia 做出的决定加载表。

在启动时，如果下列两种情况之一被发现，`Mnesia` 将认为其本地副本是最新版本并且从磁盘加载表：

- 从其它所有保存有磁盘副本的节点上都返回 `mnesia_down` 或
- 如果所有副本都是 `ram_copies` 类型

这通常是一个明智的决定，但在因掉线导致通信失败的情况下或许会造成灾难性的后果，因为 Mnesia 正常的表加载机制无法应付通信失败。

当多个表被加载时，Mnesia 按照默认的加载顺序加载。不过，可用函数 mnesia:change_table_load_order(Tab, LoadOrder)显式改变表的 load_order 属性来影响加载顺序。所

有的表默认的 LoadOrder 为 0，但可被设置为任何整数。有最高 load_order 的表将首先被加载。对需要确保重要的表能及时使用的应用程序来说，改变加载顺序非常有用。大量次要的表应设置低的加载顺序，或许低于 0。

6.7 从通信失败中恢复

当 Mnesia 发现网络被断开导致通信失败时存在几种情况。
一种情况是 Mnesia 已启动并且正在运行时，Enlang 节点再次取得联系。Mnesia 将尝试与其它节点上的 Mnesia 联系看看是否可认为网络只是暂时中断。如果 Mnesia 在两个节点上彼此都在日志中记入 mnesia_down 条目，Mnesia 生成一个称为{inconsistent_database, running_partitioned_network, Node}的系统事件，发送给 Mnesia 的事件处理器和其它可能的订阅者。默认的事件处理器向错误日志报告一个错误。
另一种情形是启动时，如果 Mnesia 发现本地节点与另一个节点彼此都收到 mnesia_down，产生一个{inconsistent_database, starting_partitioned_network, Node}系统事件并且采取如上所述的行动。
如果应用程序发现由于通信失败导致数据库的不一致，可用函数 mnesia:set_master_nodes(Tab, Nodes)精确发现每个表可以从哪个节点来加载。
Mnesia 启动时通常所用的表加载算法将被绕过，从为这个表规定的主节点（master nodes）之一来加载表，忽略日志中可能存在的 mnesia_down 条目。Nodes 仅包含表有副本的节点，如果其为空，这个特定表的主节点恢复机制将被重置，下次重启时将使用通常的加载机制。

函数 mnesia:set_master_nodes(Nodes)为全部表设置主节点。此函数为每个表确定其副本节点并且用包括在 Nodes 列表中的副本节点作为参数（TabNodes）调用 mnesia:set_master_nodes(Tab, TabNodes)函数（即 TabNodes 是 Nodes 和表的副本节点的交集）。如果这个交集为空，则这个特定表的主节点恢复机制将被重置，下次重启时将使用通常的加载机制。

函数 mnesia:system_info(master_node_tables)和 mnesia:table_info(Tab, master_nodes)可用来获得有关潜在主节点的信息。

函数 mnesia:force_load_table(Tab)可用来强行加载表而无视其被激活的加载机制。

6.8 事务的恢复

一个 Mnesia 表可以驻留在一个或多个节点上。当表被更新时，Mnesia 将确保更新被复制到表所驻留的全部节点上。如果由于某些原因（例如节点暂时宕机）复制无法进行，Mnesia 将在晚些时候再执行复制。

在应用程序启动的节点上，将有一个事务协调器进程。如果事务是分布式的，在所有需要执行提交工作的节点上也将有一个事务参与者进程。

Mnesia 内部使用多种提交协议。协议的选择取决于是哪些表在事务内部被更新。如果全部相关的表被对称复制（即这些表有同样类型的节点且当前可从协调器节点存取），将使用一个轻量级的事务提交协议。

事务协调器与其参与者之间需要交换的消息数量很少，如果提交协议中断，Mnesia 的表加载机制会处理事务恢复。因为所有相关表被对称复制的事务将会由在一个失败节点启动从相同节点加载相关表来自动恢复。如果中止或提交事务都能确保原子性、一致性、隔离性和持久性（ACID），我们就不必担心。轻量级提交协议是非阻塞的，即在提交协议期间无论节点是否崩溃，存活的参与者和其协调器将结束这个事务。

如果在一个脏操作的中间有节点宕机，表加载机制将确保或者全部副本，或者没有任何副本执行更新。异步脏操作和同步脏操作采用与轻量级事务同样的恢复原理。

如果事务涉及非对称表或模式表的更新，将使用重量级的提交协议。重量级提交协议不管表是如何复制都能够完成事务。重量级事务的典型用途是将一个副本从一个节点移送到另一个节点上。我们必须确保副本要么全部移送，要么原样保留。在事务结束时绝不能让两个节点上都有或者一个节点上都没有副本。即使在提交协议的中间有节点宕机，也必须保证事务的原子性。重量级提交协议在事务的协调者和参与者之间产生的消息比轻量级协议要多得多并且为了完成中断或提交的工作将在在启动时执行恢复工作。

重量级提交协议也是非阻塞的，允许存活的参与者和其协调者无论如何（即使在提交协议的中间有节点宕机）都能结束事务。当启动时有节点失败，Mnesia 将确定事务的结果并且恢复它。要做出正确的重量级事务恢复决定，取决于在其它正在运行的节点上的是轻量级提交协议、重量级提交协议还是脏更新。

如果 Mnesia 在一些涉及到事务的节点上没有启动，并且本地节点或任何已在运行的节点知道事务的结果，Mnesia 将默认等待其中之一。在最坏的情况下，在 Mnesia 能够做出有关事务以及结束其启动的正确决定之前，所有涉及事务的节点必须启动。

这意味着如果出现双重失败，即当两个节点同时崩溃并且其中一个尝试启动而另一个由于如硬件故障等原因拒绝启动时，Mnesia(在某个节点上)将会被挂起。

可以指定 Mnesia 在做事务恢复决定时等待其它节点响应的最大时间。配置参数 max_wait_for_decision 默认是无限（这或许会引起如上所述的无限期挂起），但如果其被设置为一个有限的时间段（例如三分钟），Mnesia 将根据需要强制执行事务恢复决定以使得其启动过程能够继续。

强制执行事务恢复决定的不利方面是，由于有关其它节点恢复决定的信息不足，这个决定或许是不正确的。其结果将会是由于事务在一些节点上被提交而在另一些节点上被中止而导致数据库的不一致。

在幸运的情况下矛盾将仅仅出现在表所归属的特定应用程序中，但如果一个模式事务由于强制事务恢复决定被不一致地恢复，其结果将是灾难性的。不过，如果可用性的优先级比一致性更高，冒险或许是值得的。

如果 Mnesia 碰到一个不一致的事务决定，将产生一个{inconsistent_database, bad_decision, Node}系统事件，给应用程序一个安装回滚或其它适当措施的机会来解决这个矛盾。Mnesia 事件处理器的默认行为与上述由于网络故障使得数据库不一致时的行为是同样的。

6.9 备份、回滚以及灾难恢复

下列函数用于备份数据、安装备份作为回滚以及灾难恢复：

- mnesia:backup_checkpoint(Name, Opaque, [Mod])。这个函数执行一个包含在检查点中的表备份。
- mnesia:backup(Opaque, [Mod])。 这个函数激活一个覆盖全部 Mnesia 表的新检查点并且执行一次备份。备份以最大冗余度执行 (也可参见函数 mnesia:activate_checkpoint(Args), {max, MaxTabs} and {min, MinTabs})。
- mnesia:traverse_backup(Source,[SourceMod,]Target,[TargetMod,]Fun,Ac)。这个函数能用来读存在的备份，从一个现存的备份创建一个新的备份，或者在不同介质之间拷贝备份。
- mnesia:uninstall_fallback()。这个函数移除先前安装的回滚文件。
- mnesia:restore(Opaque, Args)这个函数从先前的备份恢复表。
- mnesia:install_fallback(Opaque, [Mod])这个函数能够配置成从一个现存的备份重启 Mnesia 并且重新加载数据库表以及可能的模式表。当数据或模式表损毁时，此函数被用于灾难恢复。

这些函数在下列章节中解释。也可参考本章中的检查点一节，其中描述了两个用于激活以及解除检查点的函数。

6.9.1 备份

用下列函数执行备份操作：

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

在默认情况下，对备份介质的实际存取是通过 mnesia_backup 模块来读写的。当前的 mnesia_backup 模块是用标准库模块 disc_log 来实现的，但也可能用与 mnesia_backup 相同的接口写你自己的模块并且配置 Mnesia 使得这个替换模块被用来执行对备份介质的实际存取。这意味着用户可以在 Mnesia 不知道的介质上，也可能是 Erlang 没有运行的主机上放置备份。配置参数-mnesia backup_module <module>就是用与这个目的。

备份源是一个被激活的检查点。在大多数情况下使用的备份函数是 mnesia:backup_checkpoint(Name, Opaque,[Mod])。这个函数返回 ok 或{error,Reason}。其有下列参数：

- Name 是一个被激活的检查点名。 参见本章中检查点一节中的函数 mnesia:activate_checkpoint(ArgList) ，来获得怎样在检查点中包含表名的有关细节。
- Opaque。Mnesia 不解释这个参数，但会被转发给备份模块。Mnesia 的默认备份模块 mnesia_backup 将这个参数解释为一个本地文件名。
- Mod。替换模块的名字。

函数 mnesia:backup(Opaque[, Mod])激活一个以最大冗余度覆盖全部 Mnesia 表并且执行一次备份。最大冗余度的含义是每个表副本都有一个检查点保持器。local_contents 表的备份作为在当前节点上一样看待。
为了将一个备份转换为一个新备份或者仅仅为了读备份的目的，对备份进行迭代是可能的。正常返回{ok, LastAcc}的函数 mnesia:traverse_backup(Source, [SourceMod,]Target, [TargeMod,] Fun, Acc)被用于这两个目的。

在遍历开始之前，用 SourceMod:open_read(Source)打开源备份介质，用 TargetMod:open_write(Target)打开目的备份介质。这些参数是：

- SourceMod 和 TargetMod 是模块名。
- Source 和 Targe 是由模块 SourceMod 和 TargetMod 为了初始化备份介质的目的专用的不透明数据。
- Acc 是累加器的初值。
- Fun(BackupItems, Acc)被应用于
- 份中的每个项目上。这个函数必须返回一个元组{ValGoodBackupItems, NewAcc}，这里的 ValidBackupItems 是有效备份项目的列表，NewAcc 是新的累加器值。用函数 TargetMod:write/2 将 ValidBackupItems 写入目标备份。
- LastAcc 是上一个累加器的值。即由 Fun 返回的上一次的 NewAcc 值。

对源备份执行只读遍历而不更新目标备份是可能的。如果 TargetMod==read_only，则目标备份根本就不会被存取。
通过设置 SourceMod 和 TargetMod 到不同的模块把备份从一种备份介质复制到另一种是可能的。

有效 BackupItems 是下列元组：

- {schema, Tab}指定一个被删除的表。
- {schema, Tab, CreateList} 指定一个被创建的表。参看 mnesia_create_table/2 获取更多关于 CreateList 的信息。
- {Tab, Key} 指定一个被删出记录的完整标识。
- {Record} 指定一条被插入的纪录。可以是以 Tab 作为第一个字段的元组。注意记录名被设置为表名而不管 record_name 被设置为什么。

备份数据被分为两个部分。第一部分包含于模式相关的信息。所有与模式相关的项目是第一个字段等于原子模式的元组。第二部分是记录部分。模式记录与其它记录不能混合并且所有的模式记录必须首先置于备份中。

模式本身是一个表并且有可能被包含在备份中。所有驻留有模式表的节点被认为是数据库节点 db_node。

下列实例演示如何用 mnesia:traverse_backup 对在一个备份文件中的 db_node 节点改名：

```
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).
```

6.9.2 恢复

能够在不重启 Mnesia 的情况下从备份中在线恢复表。用函数 mnesia:restore(Opaque,Args)来执行恢复，这里的参数 Args 能够包含下列元组：

- `{module,Mod}`. 备份模块 `Mod` 被用来存取备份介质。如果省略，将使用默认备份模块。
- `{skip_tables, TableList}`这里 `TableList` 是那些不应该从备份里读出的表的列表。
- `{clear_tables, TableList}`这里 `TableList` 是在记录从备份中插入之前应该被清除的表的列表，即这些表中的全部记录在表被恢复之前被删除。有关这些表的模式信息不能被删除或者从备份中读出。
- `{keep_tables, TableList}` 这里 `TableList`是在记录从备份中插入之前不应该被清除的表的列表， 即在备份中的这些记录将被加到表中。有关这些表的模式信息不能被删除或者从备份中读出。

- {recreate_tables, TableList} 这里 TableList 是在记录从备份中插入之前应该被重新创建的表的列表，表首先被删除，然后再用来自备份的模式信息创建。备份中的全部节点需要启动和运行。
- {default_op, Operation}。Operation 是下列操作之一， skip_tables, clear_tables, keep_tables 或 recreate_tables。默认操作指定何种操作应该用于任何在前述列表中没有被指定的、来自备份的表。如果省略，将使用操作 clear_tables。

参数 Opaque 被转发给备份模块。如果成功，返回{atomic, TabList}，如果出错，返回元组{aborted, Reason}。在恢复操作期间，对被恢复的表加写锁。然而，应用程序能够忽略任何由此引起的锁冲突而继续完成其工作。

恢复被作为一个单独事务执行。如果数据库非常大，有可能无法在线恢复。在这种情况下，老的数据库必须通过安装一个回滚来恢复，然后重新启动。

6.9.3 回滚（Fallbacks）

函数 mnesia:install_fallback(Opaque, [Mod])用于安装作为回滚的备份。函数使用备份模块 Mod 或者默认的备份模块来存取备份介质。如果成功，返回 ok，如果出错，返回{error, Reason}。

安装回滚是仅能在所有数据库节点（db_nodes）执行的分布式操作。回滚用于在下一次系统启动之前恢复数据库。如果一个安装有回滚的 Mnesia 节点发现在另一个节点上的 Mnesia 由于某种原因死亡，将无条件的中止自身。

回滚典型的用途是在系统升级被执行时。这包括安装新版本的软件以及常见的将 Mnesia 表转换为新布局等等。如果系统在升级时崩溃，将很有可能要求重新安装老的应用程序并将数据库恢复到此前的状态。这可以通过在系统开始升级之前执行一次备份并将其作为回滚安装来实现。

如果系统升级失败，为了恢复老的数据库 Mnesia 必须在全部数据库节点上重启。在成功启动之后，回滚将被自动卸载。函数 mnesia:uninstall_fallback()也可用于在系统成功升级后卸载回滚。再次强调，回滚是一个分布式操作，要么在所有数据库节点上被执行，要么根本就不被执行。安装或卸载回滚要求 Erlang 在所有数据库节点上被启动和运行，但不在乎 Mnesia 是否在运行。

6.9.4 灾难恢复

掉电将会使系统变得不稳定。UNIX 的文件系统检查（fsck）特性能够修复文件系统，但无法保证文件的内容是一致的。

如果 Mnesia 发现可能因掉电使得文件没有正确关闭，将尝试用类似的方式修复损坏的文件。或许会丢失数据，但尽管数据可能是不一致的，仍然能够启动 Mnesia。配置参数-mnesia auto_repair <bool>可被用于控制 Mnesia 启动时的行为。如果<bool>的值为 true，Mnesia 将尝试修复文件；如果<bool>的值为 false，Mnesia 如果发现有可疑文件时将不会重启。此配置参数影响日志文件、DAT 文件以及默认备份介质的修复行为。

配置参数-mnesia dump_log_update_in_place <bool>控制 mnesia:dump_log()函数的安全级别。在默认情况下，Mnesia 将直接转储事务日志到 DAT 文件。在转储期间如果掉电将会损坏 DAT 文件。如果此参数被设置为 false，Mnesia 将复制 DAT 文件并且将目标（指事务日志-译者注）转储到新的临时文件。如果转储成功，这个临时文件将更名为其通常的 DAT 后缀。采用这种策略将使得数

据文件存在不可恢复的矛盾的可能性变得非常小。另一方面，事务日志的实际转储将会变得相当慢。系统设计者必须决定是优先考虑速度还是安全。

disc_only_copies 类型的副本仅在启动时初始化日志文件的转储期间受到这个参数的影响。当设计一个有很高要求的应用程序时，根本不使用 disc_only_copies 表或许是合适的。理由是通常的操作系统具有随机存取的本性。如果某个节点由于诸如掉电等原因宕机，由于没能正确关闭或许将导致文件损坏。disc_only_copies 的 DAT 文件依据每次事务被更新。

如果灾难发生并且 Mnesia 数据库被损坏，能够用备份中重建数据库。这应该被视为最后的补救办法，因为备份包含的是旧的数据。数据可望是一致的，但当使用一个旧的备份来恢复数据库时，显然，将会丢失数据。

7 Mnsia 与 SNMP 的结合

7.1 结合 Mnesia 与 SNMP

许多电信应用程序必须从远程控制和重配置。有时采用一个如简单网络管理协议（SNMP）这样的开放协议来执行远程控制是有好处的。可替代这样做的方法有：

- 根本就不能远程控制应用程序。
- 采用私有的控制协议。
- 采用一个可以将私有协议的控制信息映射到标准管理协议或相反的桥。

这些方法有利有弊。Mnesia 应用程序能容易的向 SNMP 协议开放。可在 Mnesia 表与 SNMP 表之间建立直接的一对一映射。这意味着一个 Mnesia 表能够被配置为两个：一个 Mnesia 表和一个 SNMP 表。一些用于控制这种行为的函数在 Mnesia 参考手册中被描述。

8 附录 A：Mnesia 错误信息

当 Mnesia 的操作返回一个错误时，都会有关于此错误的描述。例如，函数 mnesia:transaction(Fun) 或 mnesia:create_table(N,L)或许会返回元组{aborted, Reason}，这里的 Reason 即是关于此错误的描述项。下列函数被用于检索有关错误的更多的细节信息：

- `mnesia:error_description(Error)`

8.1 Mnesia 中的错误

下面是在 Mnesia 中有效的错误列表：

- `badarg`. 坏的或非法参数， 可能是坏的类型。
- `no_transaction`. 不能在事务外部执行的操作。
- `combine_error`. 非法表选项组合。
- `bad_index`. 索引已经存在或没有被绑定。
- `already_exists`.被激活的模式选项已经打开。
- `index_exists`. 依据索引对表做的某些操作不能被执行。
- `no_exists`.; 试图在不存在（非活动）的项目上执行操作。
- `system_limit`.; 系统限制被耗尽。
- `mnesia_down`. 在事务完成之前，事务涉及到的记录所在的远程节点不可使用。位于网络其它节点上的记录不再可用。
- `not_a_db_node`. 所提及的节点在模式中不存在。
- `bad_type`.; 在参数中指定了坏类型。
- `node_not_running`. 节点没有运行。
- `truncated_binary_file`. 在文件中截断二进制。
- `active`. 一些删除操作要求全部活动记录被移除。
- `illegal`.在这条记录上操作不被支持。

下列实例显示函数返回一个错误以及检索错误细节信息的方法。

函数 mnesia:create_table(bar, [{attributes, 3.14}])将返回元组{aborted,Reason}，这里 Reason 是元组{bad_type,bar,3.14000}。

函数 mnesia:error_description(Reason)返回项目{"Bad type on some provided arguments",bar,3.14000}，其为关于此错误的描述。

9 附录 B：备份回调函数接口

9.1 Mnesia 备份回调行为

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% 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}
```

10 附录 C: 作业存取回调接口

10.1 Mnnesia 存取回调行为

```
-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'.
```

11 附录 D：分片表哈希回调接口

11.1 mnesia_frag_hash 回调行为

```
-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, next_n_to_split = SplitN, n_doubles = L}, Key) ->
    P = SplitN,
    A = erlang:phash(Key, power2(L)),
```

```
    if
        A < P ->
            erlang:phash(Key, power2(L + 1));
        true ->
            A
    end;
key_to_frag_number(#hash_state{function = phash2, next_n_to_split = SplitN, n_doubles = L}, Key) ->
    P = SplitN,
    A = erlang:phash2(Key, power2(L)) + 1,
    if
        A < P ->
            erlang:phash2(Key, power2(L + 1)) + 1;
        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 tuple(HeadPat), 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)).
```