Mnesia: СУБД для Эрланга (и на Эрланге)

Предположим, вы захотели создать многопользовательскую игру, сделать новый веб-сайт, или разработать систему онлайн-платежей. Вероятно, вам в этом случае понадобится какая-либо СУБД.

Среди тысячи-другой файлов, появившихся на диске после установки Эрланга, присутствует полнофункциональная СУБД под названием Mnesia. Она исключительно быстра, и она может хранить структуры, состоящие из любых доступных в Эрланге типов данных.

Кроме того, Mnesia поддаётся детальной настройке и конфигурированию. Определённые таблицы могут храниться в RAM (для обеспечения высокой скорости доступа), другие - сохраняться на диск (выживая при падении узла), а ещё таблицы могут реплицироваться на другие компьютеры в сети, что даёт возможность строить устойчивые к сбоям системы.

Давайте познакомимся со всем этим поближе.

17.1 Запросы к базе данных

Начнём, пожалуй, с изучения запросов к БД Mnesia. При ближайшем рассмотрении оказывается, что запросы Mnesia одновременно похожи и на SQL *1), и на обработчики запросов (*`list comprehensions`*), так что нам не придётся изучать так уж много нового.* 2)

> 1) SQL - это очень известный язык, используемый для доступа к реляционным СУБД.
>
> 2) На самом деле, не столь удивительно, что SQL и обработчики списков похожи. Обе этих вещи основаны на математической теории множеств.

Во всех примерах я буду подразумевать, что вы создали базу данных с двумя таблицами с названиями `shop` и `cost`. Эти таблицы содержат следующие данные.

Таблица `shop`:

Item	Quantity	Cost
apple	20	2.3
orange	100	3.8
pear	200	3.6
banana	420	4.5

potato	2456	1.2

Таблица `cost`:

Name	Price
apple	1.5
orange	2.4
pear	2.2
banana	1.5
potato	0.6

Чтобы представить эти таблицы в подходящем для Mnesia виде, мы должны создать определения записей, которые бы описывали колонки в таблицах. Определения эти таковы:

test_mnesia.erl

```
-record(shop, {item, quantity, cost}).
-record(cost, {name, price}).
```

Теперь немного чёрной магии. Я собираюсь показать, как работают запросы, и хочу, чтобы вы могли повторить это все у себя дома. Но для этого сначала кто-то должен создать и заполнить для вас базу данных. Так что пока поверьте мне на слово - в файле test_mnesia.erl я подготовил код, который производит инициализацию, и вы можете просто запустить его из оболочки `erl`.

```
1> c(test_mnesia).
{ok,test_mnesia}
2> test_mnesia:do_this_once().
=INFO REPORT==== 29-Mar-2007::20:33:12 ===
    application: mnesia
    exited: stopped
    type: temporary
stopped
```

Теперь мы можем приступить к нашим примерам.

Выборка всех данных из таблицы

Вот так выглядит код для выборки всех данных из таблицы shop (для тех, кто знаком с SQL, каждый фрагмент кода начинается с комментария, где показан соответствующий задаче оператор SQL).

test_mnesia.erl

```
%% SQL equivalent
%% SELECT * FROM shop;
demo(select_shop) ->
    do(qlc:q([X || X <- mnesia:table(shop)])).
```

Самая суть примера заключена в вызове функции `qlc:q`, которая компилирует запрос (её параметр) во внутреннее представление, которое уже используется для непосредственного выполнения запроса.

Мы передаем полученный запрос в функцию с названием `do()`, которая определена в модуле `test_mnesia` (ближе к концу модуля). Она отвечает за выполнение запроса и возврат результата.

Для того, чтобы её можно было легко вызывать из оболочки `erl`, мы заключили этот код в функцию `demo(select_shop)` (Полный листинг модуля `mnesia_test` приведён в конце главы).

Мы можем вызвать её таким образом:

test_mnesia.erl

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
{atomic, ok}
3> test_mnesia:demo(select_shop).
[{shop,orange,100,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
```

> Примечание: выбранные из таблицы строки могут оказаться в другом порядке.

Строка, которая задает запрос, в этом примере выглядит так:

```
qlc:q([X || X <- mnesia:table(shop)])
```

Это очень похоже на обработчики списков (`list comprehensions`) (см. раздел 3.6, Обработчики списков, на странице 61?). На самом деле, `qlc` расшифровывается как "query list comprehensions". Это один из модулей, которые служат для доступа к данным в БД Mnesia.

Выражение `[X || X <- mnesia:table(shop)]` означает “список `X`, таких, что X берется из таблицы Mnesia с названием "shop"”. Значения `X` - это Эрланг-записи типа `shop`.

> Важно: аргумент функции `qlc:q/1` должен быть литералом обработчика списков (`list comprehension`, а не чем-либо, что вычисляется в такое выражение. Таким образом, например, вот такой код не будет эквивалентным тому, что написано в примере:

```
Var = [X || X <- mnesia:table(shop)],
qlc:q(Var)
```

Выборка определенных колонок из таблицы (проекция)

Следующий запрос выбирает колонки `item` и `quantity` из таблицы shop.

test_mnesia.erl

```
%% SQL equivalent
%% SELECT item, quantity FROM shop;

demo(select_some) ->
    do(qlc:q([{X#shop.item, X#shop.quantity} || X <- mnesia:table(shop)])).

4> test_mnesia:demo(select_some).
[{orange,100},{pear,200},{banana,420},{potato,2456},{apple,20}]
```

В предыдущем примере значения `X` были записями типа `shop`. Если вы вспомните синтаксис выражений с записями из раздела 3.9, Записи, на странице 69, вы увидите, что `X#shop.item` соответствует полю `item` записи типа `shop`. Таким образом, кортеж `{X#shop.item, X#shop.quantity}` состоит из двух полей записи `X` - `item` и `quantity`.

Выборка из таблицы данных, удовлетворяющих условию

Следующий запрос выберет все наименования товаров из таблицы `shop`, где количество товара на складе меньше, чем 250. Например, мы могли бы использовать этот вопрос, чтобы решить, какие товары нам надо дозаказать.

test_mnesia.erl

```
%% SQL equivalent
%% SELECT shop.item FROM shop
%% WHERE shop.quantity < 250;
```

```
demo(reorder) ->
    do(qlc:q([X#shop.item || X <- mnesia:table(shop),
        X#shop.quantity < 250
    ]));
5> test_mnesia:demo(reorder).
[orange,pear,apple]
```

Обратите внимание, как наше условие естественно вписалось в обработчик списков.

Выборка данных из двух таблиц (Join)

Теперь предположим, что мы хотим дозаказать товар только в том случае, если на складе его осталось менее 250 штук, а цена его - меньше чем 2.0 за штуку. Чтобы это сделать, нам придётся обратиться сразу к двум таблицам. Вот так выглядит нужный запрос:

test_mnesia.erl

```
%% SQL equivalent
%% SELECT shop.item, shop.quantity, cost.name, cost.price
%% FROM shop, cost
%% WHERE shop.item = cost.name
%% AND cost.price < 2
%% AND shop.quantity < 250

demo(join) ->
    do(qlc:q([X#shop.item || X <- mnesia:table(shop),
    X#shop.quantity < 250,
    Y <- mnesia:table(cost),
    X#shop.item =:= Y#cost.name,
    Y#cost.price < 2
])).
6> test_mnesia:demo(join).
[apple]
```

Здесь важным фрагментом является соединение между наименованием товара в таблице `shop` и наименованием товара в таблице `cost`:

```
X#shop.item =:= Y#cost.name
```

17.2 Вставка и удаление данных в/из БД.

Снова предполагаем, что мы создали БД и определили таблицу `shop`. Теперь мы хотим

добавить строку в таблицу, или, наоборот, удалить строку из таблицы.

Вставка строки

Мы можем добавить строку в таблицу `shop` следующим образом:

test_mnesia.erl

```
add_shop_item(Name, Quantity, Cost) ->
Row = #shop{item=Name, quantity=Quantity, cost=Cost},
F = fun() ->
    mnesia:write(Row)
    end,
mnesia:transaction(F).
```

Эта функция создаёт запись типа `shop` и вставляет её в таблицу.:

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
{atomic, ok}
%% list the shop table
3> test_mnesia:demo(select_shop).
[{shop,orange,100,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
%% add a new row
4> test_mnesia:add_shop_item(orange, 236, 2.8).
{atomic,ok}
%% list the shop table again so we can see the change
5> test_mnesia:demo(select_shop).
[{shop,orange,236,2.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
```

> Примечание: первичным ключом в таблице `shop` является первая колонка таблицы, то есть, наименование товара (`item`) в записи `shop`. Таблица создана с типом "`set`" (см. обсуждение типов `set` и `bag` в разделе 15.2, Types of Table, стр. 275). Если вновь созданная запись имеет такое же значение первичного ключа, как у уже существующей в БД строки, она перезапишет эту строку, в противном случае в БД добавится новая строка.

Удаление строки

Чтобы удалить строку, нам нужно знать ID объекта (`OID`) для этой строки. `OID` составляется из имени таблицы и значения первичного ключа:

test_mnesia.erl

```
remove_shop_item(Item) ->
    Oid = {shop, Item},
    F = fun() ->
        mnesia:delete(Oid)
        end,
    mnesia:transaction(F).

6> test_mnesia:remove_shop_item(pear).
{atomic,ok}
%% list the table -- the pear has gone
7> test_mnesia:demo(select_shop).
[{shop,orange,236,2.80000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
 [{shop,orange,236,2.80000},
8> mnesia:stop().
ok
```

17.3 Транзакции в Mnesia

Когда ранее мы добавляли или удаляли строки в/из БД, или выполняли запрос к БД, мы писали что-то вроде такого:

```
do_something(...) ->
F = fun() ->
    % ...
    mnesia:write(Row)
    % ... или ...
    mnesia:delete(Oid)
    % ... или ...
    qlc:e(Q)
    end,
mnesia:transaction(F)
```

F - это анонимная функция без аргументов. Внутри `F` мы вызывали некое сочетание

функций `mnesia:write/1`, `mnesia:delete/1` или `qlc:e(Q)` (где `Q` это запрос, предварительно скомпилированный при помощи `qlc:q/1`).

Зачем мы так делали? Что вообще значит "транзакция"? Чтобы пояснить это, предположим, что у нас есть два процесса, которые одновременно пытаются получить доступ к одним и тем же данным. Например, у меня есть 10 долларов на банковском счете. Теперь предположим, что два человека пытаются одновременно взять по 8 долларов с этого счета. В данном случае я бы хотел, чтобы одно из этих снятий произошло успешно, а другое - не произошло.

Именно эту гарантию предоставляет нам `mnesia:transaction/1`. Либо все чтения и записи данных в БД в пределах одной конкретно взятой транзакции завершаются успешно, либо не завершается успешно ни одна из этих операций. Если ни одна из составляющих транзакцию операций не успешна, мы говорим, что транзакция терпит неудачу. Если транзакция терпит неудачу, никаких изменений в БД не производится.

Стратегия, которой при этом следует Mnesia, является разновидностью пессимистических блокировок. Всякий раз, когда менеджер транзакций Mnesia обращается к таблице, он пытается заблокировать запись или всю таблицу, в зависимости от ситуации. Если он обнаруживает, что блокировка может привести к взаимоблокировке (`deadlock`), он немедленно отменяет транзакцию и откатывает все изменения, которые в пределах этой транзакции были сделаны.

Если транзакция с самого начала терпит неудачу по причине того, что другой процесс занял данные, система делает небольшую паузу и повторяет транзакцию. Одним из результатов такого поведения является то, что код внутри анонимной функции транзакции может выполняться большое количество раз.

Именно по этой причине анонимная функция транзакции не должна делать ничего такого, что приводит к побочным эффектам. Например, если бы мы написали что-то такое:

```
F = fun() ->
    ...
    io:format("reading ..." ), %% don't do this
    ...
    end,
mnesia:transaction(F),
```

мы могли бы получить на выводе большую кучу ненужного текста, просто потому что функция была перезапущена много раз.

> Примечание 1: функции `mnesia:write/1` и `mnesia:delete/1` должны вызываться только в анонимной функции, которую выполняет `mnesia:transaction/1`.

> Примечание 2: Никогда не пишите код, который явно перехватывает исключения от функций доступа Mnesia (`mnesia:write/1`, `mnesia:delete/1`, и т.п.), поскольку механизм транзакций Mnesia сам основан на том, что эти функции выбрасывают исключения при неудаче. Если вы будете перехватывать исключения и обрабатывать их сами, вы сломаете механизм транзакций.

Отмена транзакции

Около нашего магазина есть ферма, где выращивают яблоки. Хозяин фермы любит апельсины, и покупает их у нас по бартеру, на яблоки. "Курс обмена" - два к одному, то есть, чтобы купить N апельсинов, фермер даёт нам 2*N яблок.

Вот так выглядит функция, которая обновляет БД, когда фермер покупает апельсины:

test_mnesia.erl

```
farmer(Nwant) ->
%% Nwant = Number of oranges the farmer wants to buy
F = fun() ->
    %% find the number of apples
    [Apple] = mnesia:read({shop,apple}),
    Napples = Apple#shop.quantity,
    Apple1 = Apple#shop{quantity = Napples + 2*Nwant},
    %% update the database
    mnesia:write(Apple1),
    %% find the number of oranges
    [Orange] = mnesia:read({shop,orange}),
    NOranges = Orange#shop.quantity,
    if
        NOranges >= Nwant ->
            N1 = NOranges - Nwant,
            Orange1 = Orange#shop{quantity=N1},
            %% update the database
            mnesia:write(Orange1);
        true ->
            %% Oops -- not enough oranges
            mnesia:abort(oranges)
        end
    end,
mnesia:transaction(F).
```

Этот код, честно говоря, глуповат, но я так написал специально, чтобы продемонстрировать работу транзакций. Сначала я изменяю количество яблок в БД. Это делается ДО того, как я проверяю количество доступных апельсинов. Это сделано специально, чтобы показать, что изменения можно "откатить", если транзакция

неудачна. В обычной ситуации я бы отложил запись количества яблок, пока не убедился бы в наличии достаточного количества апельсинов. Посмотрим на этот код в действии. Утром фермер приходит и покупает 50 апельсинов.

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
{atomic, ok}
%% List the shop table
3> test_mnesia:demo(select_shop).
[{shop,orange,100,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
%% The farmer buys 50 oranges
%% paying with 100 apples
4> test_mnesia:farmer(50).
{atomic,ok}
%% Print the shop table again
5> test_mnesia:demo(select_shop).
[{shop,orange,50,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,120,2.30000}]
```

После обеда он приходит снова и хочет купить ещё 100 апельсинов (ничего себе, как же он любит апельсины!)

```
6> test_mnesia:farmer(100).
{aborted,oranges}
7> test_mnesia:demo(select_shop).
[{shop,orange,50,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,120,2.30000}]
```

> **Врезка**
>
> ### Почему СУБД называется Mnesia?
>
> Изначально её название было "Amnesia". Одному из наших боссов это не понравилось, он сказал: "Нельзя называть её Amnesia - мы не можем выпускать базу

> данных, которая, судя по названию, забывает всё подряд!" Так что мы убрали букву "A", и название прижилось.

Когда транзакция отменилась (когда мы вызвали `mnesia:abort(Reason)`), те изменения, которые были сделаны при помощи `mnesia:write`, были также отменены. Таким образом, состояние СУБД было восстановлено таким, каким оно было до начала транзакции.

Загрузка тестовых данных

Теперь, когда мы знаем, как работают транзакции, мы можем посмотреть на код, который загружал нам тестовые данные. Функция `test_mnesia:example_tables/0` задает данные, необходимые для инициализации таблиц. Первый элемент кортежа задает имя таблицы, следом идет содержимое строки, в том же порядке, в котором поля были описаны при определении Эрланг-записи.

test_mnesia.erl

```
example_tables() ->
[%% The shop table
    {shop, apple,20,2.3},
    {shop, orange, 100,3.8},
    {shop, pear, 200,3.6},
    {shop, banana, 420,4.5},
    {shop, potato, 2456,1.2},
%% The cost table
    {cost, apple, 1.5},
    {cost, orange, 2.4},
    {cost, pear, 2.2},
    {cost, banana, 1.5},
    {cost, potato, 0.6}
].
```

Следующий код вставляет данные в Mnesia из таблиц примера.

```
reset_tables() ->
    mnesia:clear_table(shop),
    mnesia:clear_table(cost),
    F = fun() ->
        foreach(fun mnesia:write/1, example_tables())
        end,
    mnesia:transaction(F).
```

Он всего лишь вызывает `mnesia:write` для каждого кортежа из списка, который

возвращает `example_tables/1`

Функция do()

Функция `do`, которую вызывает `demo/1`, чуть сложнее:

test_mnesia.erl

```
do(Q) ->
    F = fun() -> qlc:e(Q) end,
{atomic, Val} = mnesia:transaction(F),
Val.
```

Она вызывает `qlc:e(Q)` в рамках транзакции Mnesia. `Q` - это скомпилированный запрос `QLC`, а `qlc:e(Q)` выполняет запрос и возвращает все ответы на запрос в виде списка. Возвращаемое значение `{atomic, Val}` означает, что транзакция завершилась успешно со значением `Val`. `Val` - это значение, возвращаемое анонимной функцией транзакции.

17.4 Сохранение сложных типов данных в Mnesia.

Если вы программируете на C, то как бы вы сохранили структуру C (`struct`) в базе данных SQL? Или, если вы программируете на Java, как бы вы стали сохранять в такой же базе объект? Ответ - с изрядными сложностями. Одной из проблем при использовании привычных СУБД является ограниченность встроенных типов колонок. Вы можете хранить в колонке БД целое число, строку, число с плавающей точкой, и т.п. Но если вы хотите сохранить сложный объект, это реальная проблема.

Mnesia разработана так, чтобы хранить структуры данных Эрланг. На самом деле, вы можете сохранять в таблице Mnesia любую структуру данных, которую можно создать в Эрланг. Чтобы проиллюстрировать это, предположим, что некоторое количество архитекторов хотят сохранять в Mnesia свои чертежи. Для начала мы должны определить запись (`record`), которая будет представлять собой чертёж.

test_mnesia.erl

```
-record(design, {id, plan}).
```

Затем мы пишем функцию, которая добавляет несколько чертежей в БД:

test_mnesia.erl

```
add_plans() ->
```

```
    D1 = #design{id = {joe,1},
        plan={circle,10}},
    D2 = #design{id = fred,
        plan={rectangle,10,5}},
    D3 = #design{id = {jane,{house,23}},
        plan = {house,
            [{floor,1,
                [{doors,3},
                 {windows,12},
                 {rooms,5}]},
             {floor,2,
                [{doors,2},
                 {rooms,4},
                 {windows,15}]}]}},
    F = fun() ->
        mnesia:write(D1),
        mnesia:write(D2),
        mnesia:write(D3)
    end,
    mnesia:transaction(F).
```

Теперь мы можем добавить несколько чертежей в БД

```
1> test_mnesia:start().
ok
2> test_mnesia:add_plans().
{atomic,ok}
```

Теперь мы имеем несколько чертежей, сохраненных в БД. Мы можем извлекать их оттуда при помощи следующей функции:

```
get_plan(PlanId) ->
    F = fun() -> mnesia:read({design, PlanId}) end,
    mnesia:transaction(F).

3> test_mnesia:get_plan(fred).
{atomic,[{design,fred,{rectangle,10,5}}]}
4> test_mnesia:get_plan({jane, {house,23}}).
{atomic,[{design,{jane,{house,23}},
    {house,[{floor,1,[{doors,3},
        {windows,12},
        {rooms,5}]},
    {floor,2,[{doors,2},
        {rooms,4},
        {windows,15}]}]}}]}
```

Как вы можете видеть, и ключ, и извлекаемое значение могут быть произвольными термами Эрланг.

Говоря техническим языком, между структурами данных в БД и структурами в Эрланге отсутствует "разность сопротивлений". Это означает, что вставка и удаление сложных структур данных в/из БД работают очень быстро.

> **Врезка**
>
> ### Фрагментированные таблицы
>
> Mnesia поддерживает "фрагментированные таблицы" (горизонтальное партицирование, если пользоваться привычной терминологией СУБД). Это сделано для поддержки чрезвычайно больших таблиц. Таблицы разделены на фрагменты, которые сохраняются на разных компьютерах. Фрагменты сами по себе являются таблицами Mnesia. Фрагменты могут быть реплицируемы, могут иметь индексы, и тому подобное, как любые другие таблицы.
>
> Обратитесь к документации Mnesia User Guide за дальнейшей информацией.

17.5 Типы и расположения таблиц.

Мы можем настраивать таблицы Mnesia многими разными способами. Во-первых, таблицы могут храниться в RAM, на диске, и одновременно в RAM+на диске. Во-вторых, таблицы могут располагаться на одной машине или реплицироваться на несколько машин. Когда мы проектируем нашу БД, мы должны думать о том, какого типа данные мы хотим хранить в тех или иных таблицах. Свойства таблицы могут быть такими:

RAM tables

> Эти таблицы работают очень быстро. Данные в них непостоянны, так как теряются при сбое или перезагрузке компьютера, или когда вы останавливаете СУБД.

Disk tables

> Хранимые на диске таблицы должны выживать при сбоях (при условии, что диск физически не пострадал).
>
> Когда транзакция записывает данные в таблицу, которая хранится на диске, реально данные, затронутые транзакцией, сначала пишутся в дисковый журнал. Этот журнал постоянно пополняется, и через регулярные промежутки времени его информация консолидируется с другими данными в БД, после чего обработанные записи из журнала стираются. Если система падает, то при перезагрузке дисковый журнал

> проверяется на актуальность, и все необработанные записи из журнала заносятся в БД прежде чем БД становится доступной для работы. Как только транзакция успешно завершилась, её данные должны быть надлежащим образом записаны в дисковый журнал, и если после этого система упала, а потом перезагружена, изменения, сделанные транзакцией, при этом не потеряются.
>
> Если же система упала во время транзакции, то все изменения, сделанные в пределах этой транзакции, должны быть потеряны.

Прежде чем использовать таблицы в RAM, нужно экспериментально убедиться, что вся таблица убирается в физически доступный объём памяти. Если таблица не влезает в физическую память, система будет активно использовать файл подкачки, а это обычно убивает производительность.

Поскольку таблицы RAM непостоянны, мы должны спросить себя - будет ли нам плохо, если все данные в нашей таблице RAM вдруг исчезнут? Если ответ "да", то нам нужно реплицировать нашу таблицу на диск, или реплицировать её на другой компьютер (как таблицу RAM, как дисковую таблицу, или RAM+диск).

Создание таблиц

Для того, чтобы создать таблицу, мы вызываем `mnesia:create_table(Name, ArgS)`, где `ArgS` - это список кортежей вида `{Key,Val}`. `create_table` возвращает `{atomic, ok}`, если таблица была успешно создана; если нет - возвращает `{aborted, Reason}`.

Вот некоторые часто используемые аргументы для создания таблиц:

Name

> Задаёт имя создаваемой таблицы (это должен быть `atom`). По соглашению, это также имя соответствующей Эрланг-записи (`record`) - каждая строка в таблице будет являться экземпляром этой записи.

{type, Type}

> Задаёт тип таблицы. Тип может быть одним из следующих: `set`, `ordered_set`, или `bag`. Эти типы имеют такое же значение, как описано в разделе 15.2, Types of Table, на странице 275.

{disc_copies, NodeList}

> `NodeList` задаёт список узлов Эрланг, на которых будут храниться дисковые копии создаваемой таблицы. Когда мы используем эту опцию, система также создаёт RAM-копию таблицы на том узле, на котором было выполнено создание таблицы. В

> принципе, допустимо иметь реплицируемую таблицу типа `disc_copies` на одном узле, плюс иметь ту же таблицу с другим типом на другом узле. Это делается, если мы хотим добиться следующего:
>
> 1. Чтобы операции чтения были очень быстрыми и выполнялись из RAM
> 2. При этом операции записи выполнялись бы в постоянное хранилище.

{ram_copies, NodeList}

> `NodeList` задаёт список узлов Эрланг, на которых будут храниться копии в RAM.

{disc_only_copies, NodeList}

> `NodeList` задаёт список узлов Эрланг, на которых будут храниться копии данных в режиме только-на-диск. Такие таблицы не имеют реплики в RAM, и, соответственно, доступ к ним медленнее.

{attributes, AtomList}

> `AtomList` задаёт список имен колонок в данной таблице. Заметим, чтобы создать таблицу, содержащую экземпляры записи (`record`) с именем `xxx`, мы можем использовать синтаксис: `{attribute, record_info(fields, xxx)}` (конечно, по-другому, можно явно указать список имен).

Примечание:

> у функции `create_table` есть и другие опции, здесь я показал не все. За более точной информацией обратитесь к документации по Mnesia.

Часто используемые комбинации атрибутов таблиц

Во всех дальнейших примерах мы будем предполагать, что `Attrs` является кортежем вида `{attributes,...}`.

Приведём здесь некоторые часто используемые опции настройки таблиц, покрывающих большинство типовых случаев:

```
mnesia:create_table(shop, [Attrs])
```

- Таблица хранится в RAM на единственном узле.
- Если узел падает, таблица будет потеряна.
- Самый быстрый из всех методов.
- Таблица должна помещаться в физическую память.

```
mnesia:create_table(shop,[Attrs,{disc_copies,[node()]}])
```

- Таблица располагается в RAM + копия на диске, на единственном узле.
- Если узел падает, таблица будет восстановлена с диска.
- Быстрые операции чтения, запись более медленная.
- Таблица должна помещаться в физическую память.

```
mnesia:create_table(shop, [Attrs,{disc_only_copies,[node()]}])
```

- Копия только-на-диске на единственном узле.
- Большие таблицы не обязаны помещаться полностью в память.
- Доступ медленнее, чем к таблицам с репликой в RAM.

```
mnesia:create_table(shop,[Attrs,{ram_copies,[node(),someOtherNode()]})
```

- Таблица хранится в RAM на двух разных узлах.
- Если оба узла падают, таблица будет потеряна.
- Таблица должна помещаться в физическую память.
- К таблице можно обращаться с любого из этих двух узлов.

```
mnesia:create_table(shop,[Attrs, {disc_copies, [node(),someOtherNode()]}])
```

- Дисковые копии на двух узлах.
- Таблица может быть восстановлена на любом из этих двух узлов.
- Таблица выживает после двойного сбоя.

Поведение таблиц при репликации

Когда таблица реплицируется на несколько узлов Эрланг, она синхронизируется, пока это возможно. Если один узел падает, система продолжает работать, но количество реплик уменьшается. Когда упавший узел возвращается в строй, он "догонит" состояние таблицы, обратившись к другим узлам, где хранятся актуальные реплики.

> Заметка: Mnesia может оказаться перегруженной, если узлы, на которых она запущена, прекращают работать. Если вы используете ноутбук, который "засыпает" при неактивности, то при "просыпании" Mnesia может оказаться временно перегруженной и выдавать множество сообщений о предупреждениях. На эти предупреждения можно не обращать внимания.

17.6 Первоначальное создание базы данных

Вот так выглядят команды, создающие базу данных Mnesia. Вам нужно выполнить это лишь один раз.

```
$ erl
1> mnesia:create_schema([node()]).
ok
2> init:stop().
ok
$ ls
Mnesia.nonode@nohost
```

Вызов функции `mnesia:create_schema(NodeList)` инициализирует новую базу данных Mnesia на всех узлах из списка `NodeList` (он должен содержать список реально существующих Эрланг-узлов). В нашем случае мы задаём список узлов, как `[node()]`, то есть, содержащим лишь один, текущий, узел. Mnesia инициализируется и создаёт на диске структуру каталогов с названием `Mnesia.nonode@nohost`, где физически хранит базу данных. Мы можем выйти из оболочки и выполнить команду `ls`, чтобы проверить это.

Если мы повторим это упражнение на запущенном в распределённом режиме узле с именем `joe`, получим такой результат:

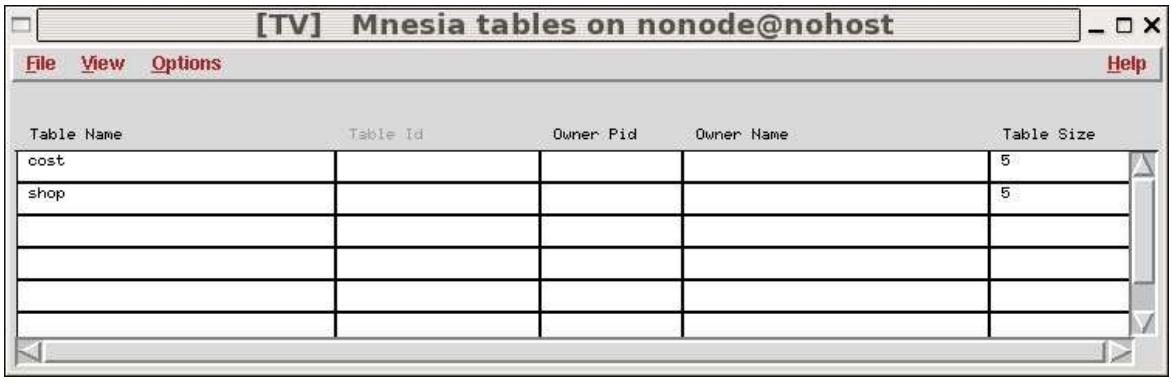
```
$ erl -name joe
(joe@doris.myerl.example.com) 1> mnesia:create_schema([node()]).
mnesia:create_schema([node()]).
ok
2> init:stop().
ok
$ ls
Mnesia.joe@doris.myerl.example.com
```

С другой стороны, мы можем сами указать путь к нужной нам конкретной базе данных при старте Эрланга:

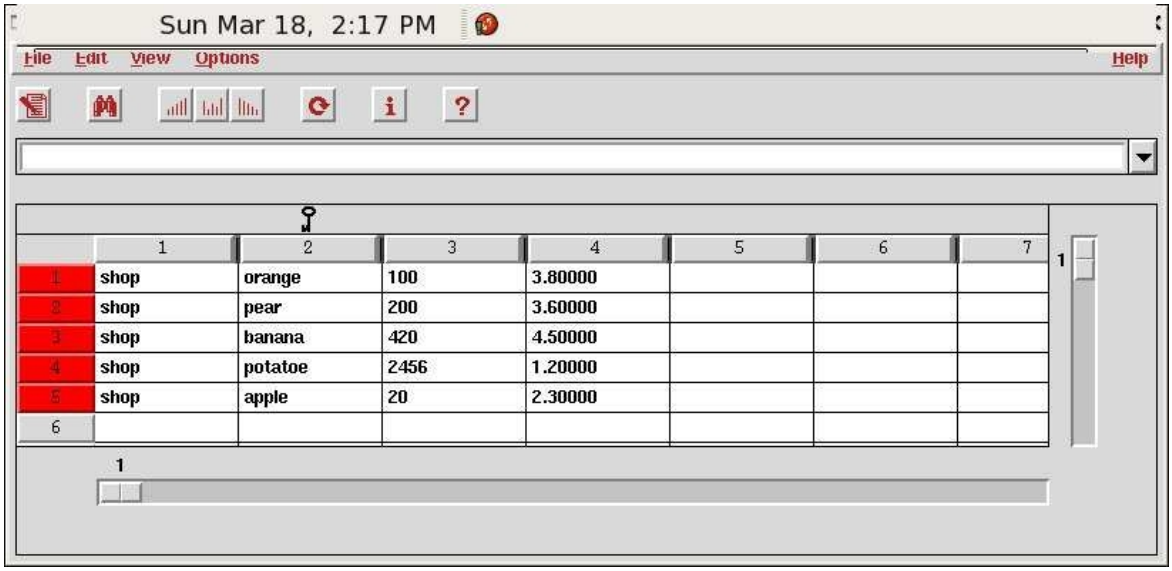
```
$ erl -mnesia dir '"/home/joe/some/path/to/Mnesia.company"'
1> mnesia:create_schema([node()]).
ok
2> init:stop().
ok
```

`/home/joe/some/path/to/Mnesia.company` - это путь к директории, где будет храниться база данных.

17.7 The Table Viewer



Просмотрщик таблиц представляет собой GUI для просмотра таблиц Mnesia, а также таблиц ETS. Команда `tv:start()` запускает `table viewer`. Вы увидите начальный экран наподобие того, что приведен на рисунке 17.1. Чтобы увидеть таблицы Mnesia, вам нужно выбрать в меню `View` > `Tab`. На рисунке 17.2 мы видим, как `table viewer` показывает нам таблицу `shop` (на следующей странице).



17.8 Куда копать?

Я надеюсь, мне удалось пробудить ваш аппетит к Mnesia. Mnesia является очень мощной СУБД. Она работает в промышленном использовании в ряде весьма требовательных телекоммуникационных систем, выпускаемых компанией Ericsson с 1998 года.

Поскольку вы читаете книгу об Эрланге, а не о СУБД Mnesia, мне придётся ограничиться лишь несколькими примерами наиболее типовых способов использования Mnesia. Приёмы, которые я продемонстрировал в этой главе, я использую в работе

сам. Я бы не сказал, что использую (и понимаю) сильно больше того, что я уже показал вам. Но при помощи того, что я показал вам, вы можете делать довольно много, и создавать довольно развитые приложения.

Перечислю главные области, которые я не включил в эту главу:

• Резервирование и восстановление: у Mnesia есть ряд опций для настройки операций резервного копирования, подходящих для различных способов восстановления.

• "грязные" операции: Mnesia может выполнять ряд "грязных" операций (`dirty_read`, `dirty_write`, ...). Это операции, которые выполняются в обход транзакционного контекста. Это довольно опасные операции, которыми, впрочем, можно пользоваться, если ваше приложение является однопоточным, или при некоторых других особых обстоятельствах. "Грязные" операции используются в целях повышения эффективности.

• Таблицы SNMP: Mnesia предоставляет встроенный тип таблиц SNMP. Это делает реализацию систем управления SNMP довольно простой.

Исчерпывающий справочник по СУБД Mnesia - это Mnesia User's Guide, его можно найти на главном сайте дистрибутивов Erlang (см Приложение C, на стр. 399). Дополнительно, поддиректория examples, включенная в дистрибутив Mnesia (`/usr/local/lib/erlang/lib/mnesia-X.Y.Z/examples` на моей машине) содержит ряд примеров использования Mnesia.

17.9 Listings

test_mnesia.erl

```
%% ---
%%  Excerpted from "Programming Erlang",
%%  published by The Pragmatic Bookshelf.
%%  Copyrights apply to this code. It may not be used to create training mc
%%  courses, books, articles, and the like. Contact us if you are in doubt.
%%  We make no guarantees that this code is fit for any purpose.
%%  Visit http://www.pragmaticprogrammer.com/titles/jaerlang for more book
%%---
-module(test_mnesia).
-import(lists, [foreach/2]).
-compile(export_all).

%% IMPORTANT: The next line must be included
%%            if we want to call qlc:q(...)
```

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


-record(shop, {item, quantity, cost}).
-record(cost, {name, price}).


-record(design, {id, plan}).



do_this_once() ->
    mnesia:create_schema([node()]),
    mnesia:start(),
    mnesia:create_table(shop,   [{attributes, record_info(fields, shop)}]),
    mnesia:create_table(cost,   [{attributes, record_info(fields, cost)}]),
    mnesia:create_table(design, [{attributes, record_info(fields, design)}]
    mnesia:stop().


start() ->
    mnesia:start(),
    mnesia:wait_for_tables([shop,cost,design], 20000).



%% SQL equivalent
%%  SELECT * FROM shop;

demo(select_shop) ->
    do(qlc:q([X || X <- mnesia:table(shop)]));




%% SQL equivalent
%%  SELECT item, quantity FROM shop;

demo(select_some) ->
    do(qlc:q([{X#shop.item, X#shop.quantity} || X <- mnesia:table(shop)]));
```

```
%% SQL equivalent
%%   SELECT shop.item FROM shop
%%   WHERE  shop.quantity < 250;

demo(reorder) ->
    do(qlc:q([X#shop.item || X <- mnesia:table(shop),
                 X#shop.quantity < 250
                ]));


%% SQL equivalent
%%   SELECT shop.item
%%   FROM shop, cost
%%   WHERE shop.item = cost.name
%%     AND cost.price < 2
%%     AND shop.quantity < 250

demo(join) ->
    do(qlc:q([X#shop.item || X <- mnesia:table(shop),
                 X#shop.quantity < 250,
                 Y <- mnesia:table(cost),
                 X#shop.item =:= Y#cost.name,
                 Y#cost.price < 2
                ])).



do(Q) ->
    F = fun() -> qlc:e(Q) end,
    {atomic, Val} = mnesia:transaction(F),
    Val.




example_tables() ->
    [%% The shop table
     {shop, apple,   20,   2.3},
     {shop, orange,  100,  3.8},
     {shop, pear,    200,  3.6},
     {shop, banana,  420,  4.5},
     {shop, potato,  2456, 1.2},
     %% The cost table
     {cost, apple,   1.5},
     {cost, orange,  2.4},
```

```
     {cost, pear,    2.2},
     {cost, banana,  1.5},
     {cost, potato,  0.6}
    ].



add_shop_item(Name, Quantity, Cost) ->
    Row = #shop{item=Name, quantity=Quantity, cost=Cost},
    F = fun() ->
        mnesia:write(Row)
    end,
    mnesia:transaction(F).



remove_shop_item(Item) ->
    Oid = {shop, Item},
    F = fun() ->
        mnesia:delete(Oid)
    end,
    mnesia:transaction(F).




farmer(Nwant) ->
    %% Nwant = Number of oranges the farmer wants to buy
    F = fun() ->
        %% find the number of apples
        [Apple] = mnesia:read({shop,apple}),
        Napples = Apple#shop.quantity,
        Apple1  = Apple#shop{quantity = Napples + 2*Nwant},
        %% update the database
        mnesia:write(Apple1),
        %% find the number of oranges
        [Orange] = mnesia:read({shop,orange}),
        NOranges = Orange#shop.quantity,
        if
            NOranges >= Nwant ->
            N1 =  NOranges - Nwant,
            Orange1 = Orange#shop{quantity=N1},
            %% update the database
            mnesia:write(Orange1);
            true ->
```

```
            %% Oops -- not enough oranges
            mnesia:abort(oranges)
        end
    end,
    mnesia:transaction(F).




reset_tables() ->
    mnesia:clear_table(shop),
    mnesia:clear_table(cost),
    F = fun() ->
        foreach(fun mnesia:write/1, example_tables())
    end,
    mnesia:transaction(F).




add_plans() ->
    D1 = #design{id   = {joe,1},
         plan = {circle,10}},
    D2 = #design{id   = fred,
         plan = {rectangle,10,5}},
    D3 = #design{id   = {jane,{house,23}},
         plan = {house,
             [{floor,1,
               [{doors,3},
                {windows,12},
                {rooms,5}]},
              {floor,2,
               [{doors,2},
                {rooms,4},
                {windows,15}]}]}},
    F = fun() ->
        mnesia:write(D1),
        mnesia:write(D2),
        mnesia:write(D3)
    end,
    mnesia:transaction(F).
```

```
get_plan(PlanId) ->
    F = fun() -> mnesia:read({design, PlanId}) end,
    mnesia:transaction(F).
```