Глава 15 ETS и DETS: механизмы хранения данных

ets и dets — это два системных модуля, используемых для эффективного хранения большого количества данных. ETS является сокращением от Erlang term storage (хранилище данных Эрланга), а DETS — сокращением от disk Erlang term Storage (дисковое хранилище данных эрланга).

ETS и DETS выполняют одинаковую задачу: предоставляют большие таблицы для хранения данных в формате ключ-значение. ETS хранит данные в памяти, а DETS на диске. Механизм ETS очень эффективен — используя ETS, вы можете хранить огромные количества данных (если у вас достаточно памяти) и осуществлять выборки за постоянное время (или, в некоторых случаях, за логарифмическое). DETS предоставляет интерфейс похожий на интерфейс ETS, но хранит таблицы на диске. Так как DETS хранит данные на диске, он гораздо медленнее чем ETS, но использует гораздо меньше памяти во время работы. Несколько процессов могут совместно работать с ETS и DETS таблицами, при этом обеспечивается высокоэффективный доступ к общим данным.

ETS и DETS представляют собой структуры данных для задания соответствия между ключами и значениями. Мы рассмотрим наиболее общие операции над таблицами: вставку и выборку. ETS и DETS таблицы представляют собой просто список кортежей Эрланга.

Данные в ETS являются временными и будут удалены, когда завершатся процессы работающие с таблицей. Данные же хранимые в DETS являются постоянными и должны остаться даже в случае краха всей системы. При открытии DETS таблица проверяется на консистентность. Если она повреждена, то предпринимается попытка восстановить таблицу (что может занять долгое время, пока все данные в таблице будут проверены). Это должно восстановить все данные в таблице, хотя последняя запись в таблице может потеряться, если она попала на момент краха системы.

ETS таблицы широко используются для программ, которые эффективно работают с большими количествами данных, и где слишком дорого использовать неразрушающее присваивание и базовые структуры данных Эрланга.

ETS таблицы выглядят как если бы они были реализованы на Эрланге, но на самом деле они реализованы на более низком системном уровне и обладают производительностью отличной от обычных объектов Эрланга. В частности, ETS таблицы не обрабатываются сборщиком мусора; это означает, что сборщик мусора не мешает при работе с очень большими таблицами, хотя он может немного повлиять на операции создания или доступа к ETS объектам.

15.1 Базовые операции с таблицами

Существуют четыре базовые операции с ETS и DETS таблицами:

Создание новой таблицы или открытие существующей.

Это производится с помощью ets:new или dets:open_file.

Вставка одного или нескольких кортежей в таблицу.

Вызываем insert(Tablename, X), где X — кортеж, или список кортежей. В ETS и DETS функция insert принимает одинаковые аргументы и выполняет одинаковую операцию.

Поиск кортежа в таблице.

Вызываем lookup(TableName, Key). Результатом будет список кортежей, которые соответствуют ключу (Key). lookup определена для ETS и DETS.

(Почему возвращаемое значение список кортежей? Если тип таблицы "bag", то одному значению ключа может соответствовать несколько кортежей. Мы рассмотрим типы таблиц в следующем параграфе.)

Если ни один из кортежей не соответствует указанному ключу, вернётся пустой список.

Завершение работы с таблицей.

Когда мы закончили работать с таблицей, мы можем сообщить об этом системе вызвав dets:close(TableId) или ets:delete(TableId).

15.2 Типы таблиц

ETS и DETS таблицы хранят кортежи. Один из элементов кортежа (по умолчанию — первый) называется ключом таблицы. Мы вставляем данные в таблицу и извлекаем данные оттуда по ключу. Что происходит, когда мы вставляем в таблицу данные, зависит от типа таблицы и значения вставляемого ключа. Некоторые таблицы, называемые множествами (set), требуют, чтобы все ключи в таблице были уникальными. Другие, называемые сумками (bag), позволяют нескольким кортежам иметь одинаковый ключ.

Выбор правильного типа таблицы оказывает сильное влияние на производительность ваших программ.

Каждый из двух типов таблиц имеет два варианта, что порождает четыре типа таблиц: множество (set), упорядоченное множество (ordered set), сумка (bag) и сумка с повторами (duplicate bag). В множестве все ключи кортежей таблицы должны быть

уникальными. В упорядоченном множестве кортежи отсортированы. В сумке может быть более одного кортежа с одним ключом, но не может быть двух полностью одинаковых кортежей. В сумке с повторами несколько кортежей могут обладать одинаковым ключом, и одинаковые кортежи могут встречаться много раз.

Мы можем показать как это работает с помощью следующей маленькой программы:

Скачать ets_test.erl

```
-module(ets_test). -export([start/0]).

start() -> lists:foreach(fun test_ets/1, [set, ordered_set, bag, duplicate_bag]).

test_ets(Mode) -> TableId = ets:new(test, [Mode]), ets:insert(TableId, {a,1}), ets:insert(TableId, {b,2}), ets:insert(TableId, {a,1}), ets:insert(TableId, {a,3}), List = ets:tab2list(TableId), io:format("\~-13w => \~p\~n", [Mode, List]), ets:delete(TableId).
```

Эта программа открывает ETS таблицу в каждом из четырёх режимов и вставляет кортежи {a,1}, {b,2}, {a,1} и {a,3}. Тогда мы вызываем tab2list, которая преобразует таблицу целиком в список, и печатаем её.

Когда мы запустим программу, получим:

```
1> ets_test:start(). set => [{b,2},{a,3}] ordered_set => [{a,3},{b,2}] bag => [{b,2},{a,1},{a,3}] duplicate_bag => [{b,2},{a,1},{a,1},{a,3}]
```

Для множества (set) каждый ключ встречается только один раз. Если мы вставили кортеж {a,1}, а вслед за ним {a,3}, тогда итоговое значение будет {a,3}. Единственное отличие между упорядоченным множеством (ordered set) и множеством состоит в том, что элементы упорядочены по значению ключа. Мы можем увидеть порядок, когда преобразуем таблицу в список вызвав tab2list.

Сумочные типы таблиц могут содержать несколько записей с одним ключом. Так, например, когда мы вставляем {a,1} и {a,3}, сумка (bag) будет содержать оба кортежа, а не только последний. В сумке с повторами (duplicate bag) допустимо наличие нескольких идентичных кортежей. Так, когда мы вставляем {a,1} и {a,1} в сумку с повтором, результирующая таблица содержит две копии кортежа {a,1}, хотя в обычной сумке, была бы только одна копия кортежа.

15.3 Соображения об эффективности ETS таблиц

Внутри, ETS таблицы представляются хеш таблицами (за исключением упорядоченных множеств, которые представляются сбалансированными двоичными деревьями). Это означает, что происходят незначительные затраты на память при использовании множеств и временные затраты при использовании упорядоченных множеств. Вставка

в множество занимает постоянное время, но вставка в упорядоченное множество занимает время пропорциональное логарифму общего количества записей в таблице.

Когда вы выбираете между множеством и упорядоченным множеством, вы должны решить, что будете делать с таблицей после её создания — если вам нужна отсортированная таблица, используйте упорядоченное множество.

Сумки более накладно использовать чем сумки с повторами, так как на каждую вставку все элементы с заданным ключом будут сравниваться на полную идентичность. Если число элементов с одинаковым ключом велико, то использование простых сумок может быть очень неэффективным.

ETS таблицы располагаются в отдельных областях памяти, которые не связаны с обычной памятью процесса. ETS таблица привязывается к процессу, который создал её — когда этот процесс умирает, или когда вызывается ets:delete, таблица удаляется. ETS таблицы не обрабатываются сборщиком мусора, это означает, что большие объёмы данных могут храниться в таблице без затрат на сборку мусора.

Когда кортеж вставляется в ETS таблицу, все структуры данных этого кортежа копируются из стека процесса и кучи в ETS таблицу. Когда производится выборка, результирующие кортежи копируются из ETS таблицы в стек и кучу процесса.

Сказанное верно для всех структур за исключением двоичных данных. Большие двоичные данные располагаются в отдельных хранилищах (не в куче). Эти хранилища могут совместно использоваться несколькими процессами и ETS таблицами. Двоичные данные управляются сборщиком мусора основанным на счётчике ссылок, который отслеживает сколько различных процессов и ETS таблиц использует двоичные данные. Когда счётчик ссылок опускается до нуля, место в хранилище может быть освобождено.

Всё это может показаться довольно сложным, но в результате отправка сообщений с большими двоичными данными между процессами довольно быстра, и вставка кортежей в ETS таблицы, которые содержат большие двоичные данные также очень быстра. Хорошее правило использовать двоичные данные как можно чаще для отображения строк и больших блоков нетипизированных данных.

15.4 Создание ETS таблицы

ETS таблицы создаются вызовом ets:new. Процесс, который создаёт таблицу называется "владельцем" (owner) таблицы. При создании таблицы, у неё есть набор параметров, которые не могут быть изменены в дальнейшем. Если процесс владелец умирает, место из под таблицы автоматически освобождается; либо таблица может быть удалена вызовом ets:delete.

ets:new принимает следующие аргументы:

@spec ets:new(Name, [Opt]) -> TableId

Name - атом.

[Opt] - список параметров из следующего перечня:

set | ordered_set | bag | duplicate_bag

Создать ETS таблицу указанного типа (мы говорили о них ранее).

private

Создать скрытую таблицу. Только процесс владелец может читать и писать в эту таблицу.

public

Создать публичную таблицу. Любой процесс, который знает идентификатор таблицы, может читать и писать в таблицу.

protected

Создать защищённую таблицу. Любой процесс, который знает идентификатор таблицы, может читать из неё, но только владелец может писать.

named_table

Если указан, то Name может быть использован для последовательных операций с таблицами.

{keypos, K}

Использовать K как позицию ключа в кортеже. Обычно позиция 1 используется для ключа. Вероятно, единственный случай, когда нам понадобится использовать этот параметр, если мы храним Erlang record,(который на самом деле замаскированный кортеж), где первый элемент содержит имя record.

ETS таблицы как аудиторные доски

Защищённые таблицы предоставляют некоторое подобие аудиторных досок (доски для мела в аудиториях). Вы можете думать о ETS таблицах как о поименованных аудиторных досках. Каждый, кто знает имя доски, может читать с неё, но только владелец может писать на ней.

Замечание: Если ETS таблица была открыта в публичном режиме, то любой процесс,

который знает её имя может и читать и писать в неё. В этом случае, пользователь должен самостоятельно убедиться, что чтение и запись производятся правильно и не нарушают целостности данных.

Замечание: Открытие таблицы ETS с пустыми параметрами равнозначно открытию таблицы с параметрами [set,protected,{keypos,1}].

Весь код в этой главе использует защищённые ETS таблицы. Защищённые таблицы особенно полезны, так как они позволяют предоставить общий доступ к данным практически без накладных затрат. Все локальные процессы, которые знают идентификатор таблицы, могут читать данные, но только один процесс может изменять таблицу.

15.5 Примеры программ использующих ETS

Примеры в данном параграфе предназначены для работы с формированием триграмм. Это отличный способ продемонстрировать силу ETS таблиц.

Наша цель — написать эвристическую программу, которая пытается предсказать, является ли заданная строка английским словом. Мы собираемся использовать её, когда будем создавать модуль полнотекстового поиска в параграфе 20.4, mapreduce и проводить индексирование нашего диска на странице 379.

Как же мы можем предсказать, является ли случайная последовательность букв действительным английским словом? Один из способов — использовать триграммы. Триграмма - это последовательность трёх букв. Не все триграммы допустимы в действительном английском слове. Например, не существует английских слов, где существуют комбинации akj или rwb. Итак, для проверки, может ли быть заданная строка английским словом, мы будем проверять все наборы из трёх последовательных букв в строке на предмет совпадения с множеством триграмм, собранном на большом количестве английских слов.

Первое, что наша программа сделает, это вычислит все триграммы в английском языке на очень большом множестве слов. Чтобы сделать это, мы воспользуемся таблицей ETS с типом множество. Решение использовать ETS множества основано на результатах измерения относительной производительности множеств и упорядоченных множеств ETS, а также "чистых" Эрланговых множеств предоставляемых модулем set.

Вот что мы собираемся сделать:

1. Создать итератор, который проходит по всем триграммам английского языка. Это очень сильно упростит написание кода вставки триграмм в различные типы таблиц.

2. Создадим ETS таблицы с типами множество и упорядоченное множество для хранения всех триграмм. Также, создадим множество содержащее все эти триграммы.

3. Замерим сколько времени потребуется для построения таблиц.

4. Замерим время поиска в этих таблицах.

5. Основываясь на измерениях, выберем наилучший тип и напишем функции доступа для этого типа.

Весь код находится в lib_trigrams. Мы собираемся представить его упустив некоторые детали. Но не беспокойтесь, вы найдёте полный листинг программы в конце главы. Таков наш план. Итак, приступим.

Итератор триграмм

Мы определим функцию for_each_trigram_in_the_english_language(F, A). Эта функция применяет fun F к каждой триграмме в английском языке. F - это fun типа fun(Str, A) -> A, а Str пробегает все значения триграмм в языке и A - это аккумулятор.

Чтобы написать наш итератор HYPERLINK\ \l\ "FOOTNOTE-1"\ , нам потребуется большой список слов. Я использовал набор из 354984 английских слов HYPERLINK\ \l\ "FOOTNOTE-2"\ , для формирования триграмм. Используя этот список слов, мы можем определить итератор триграмм следующим способом:

Скачать lib_trigrams.erl for_each_trigram_in_the_english_language(F, A0) -> {ok, Bin0} = file:read_file("354984si.ngl.gz"), Bin = zlib:gunzip(Bin0), scan_word_list(binary_to_list(Bin), F, A0). scan_word_list([], *, A) -> A; scan_word_list(L, F, A) -> {Word, L1} = get_next_word(L, []), A1 = scan_trigrams([\$\s|Word], F, A), scan_word_list(L1, F, A1). %% scan the word looking for \r\n %% the second argument is the word (reversed) so it %% has to be reversed when we find \r\n or run out of characters get_next_word([\$\r,\$\n|T], L) -> {reverse([\$\s|L]), T}; get_next_word([H|T], L) -> get_next_word(T, [H|L]); get_next_word([], L) -> {reverse([\$\s|L]), []}. scan_trigrams([X,Y,Z], F, A) -> F([X,Y,Z], A); scan_trigrams([X,Y,Z|T], F, A) -> A1 = F([X,Y,Z], A), scan_trigrams([Y,Z|T], F, A1); scan_trigrams(*, _, A) -> A.

Здесь следует отметить два момента. Во-первых, мы использовали zlib:gunzip(Bin) чтобы разархивировать zip-архив. Список слов достаточно велик, так что мы предпочли сохранить его на диск в сжатом виде, а не как текстовый файл. Во-вторых, мы добавили пробел до и после каждого слова Second; в нашем анализаторе триграмм, мы хотим интерпретировать пробел как обычную букву.

Строим таблицы

Мы строим наши ETS таблицы так:

Скачать lib_trigrams.erl make_ets_ordered_set() -> make_a_set(ordered_set, "trigramsOS.tab"). make_ets_set() -> make_a_set(set, "trigramsS.tab"). make_a_set(Type, FileName) -> Tab = ets:new(table, [Type]), F = fun(Str, _) -> ets:insert(Tab, {list_to_binary(Str)}) end, for_each_trigram_in_the_english_language(F, 0), ets:tab2file(Tab, FileName), Size = ets:info(Tab, size), ets:delete(Tab), Size.

Обратите внимание, как мы вставляли отдельные триграммы, например ABC. Мы на самом деле мы вставляли кортеж {<<"ABC">>} в таблицу ETS. Это выглядит забавно — кортеж только с одним элементом. Что это значит? Конечно, кортеж это контейнер для нескольких элементов, но это не запрещает иметь контейнер только с одним элементом. Но вспомните, что все записи в ETS таблицах должны быть кортежами, и по умолчанию первый элемент кортежа является ключом. Итак, в нашем случае, кортеж {Key} является ключом без значения.

Вот код, который строит множество всех триграмм (на сей раз с использованием Эрлангового модуля sets и без ETS):

Скачать lib_trigrams.erl make_mod_set() -> D = sets:new(), F = fun(Str, Set) -> sets:add_element(list_to_binary(Str),Set) end, D1 = for_each_trigram_in_the_english_language(F, D), file:write_file("trigrams.set" , [term_to_binary(D1)]).

Сколько времени занимает построение таблиц?

Функция lib_trigrams:make_tables(), показанная в листинге в конце главы, строит все таблицы. Она включает несколько инструкций, которые помогут определить размер таблиц и время необходимое для их создания.

1> lib_trigrams:make_tables(). Counting - No of trigrams=3357707 time/trigram=0.577938 Ets ordered Set size=19.0200 time/trigram=2.98026 Ets set size=19.0193 time/trigram=1.53711 Module Set size=9.43407 time/trigram=9.32234 ok

О чём это говорит нам? Во-первых, у нас 3.3 миллиона триграмм, и требуется пол микросекунды для обработки каждой триграммы. Время вставки триграммы в ETS упорядоченное множество было 2.9 микросекунды, в ETS множество было 1.5 микросекунды и 9.3 микросекунды в множество Эрланга. Что касается хранилищ, ETS множества и упорядоченные множества потребляют 19 байт на триграмму, а модуль sets потребляет 9 байт на триграмму.

Каково время доступа к таблицам?

Итак, требуется некоторое время для построения таблиц, но в нашем случае это не так уж важно. Более важный вопрос, каково время доступа к таблице? Чтобы ответить на этот вопрос, надо написать код, который будет измерять время доступа к таблице. Мы

будем искать каждую триграмму в таблице ровно один раз и определять среднее время на поиск. Вот код, который определяет время:

Скачать lib_trigrams.erl timer_tests() -> time_lookup_ets_set("Ets ordered Set" , "trigramsOS.tab"), time_lookup_ets_set("Ets set" , "trigramsS.tab"), time_lookup_module_sets(). time_lookup_ets_set(Type, File) -> {ok, Tab} = ets:file2tab(File), L = ets:tab2list(Tab), Size = length(L), {M, *} = timer:tc(?MODULE, lookup_all_ets, [Tab, L]), io:format("\~s lookup=\~p micro seconds\~n" ,[Type, M/Size]), ets:delete(Tab). lookup_all_ets(Tab, L) -> lists:foreach(fun({K}) -> ets:lookup(Tab, K) end, L). time_lookup_module_sets() -> {ok, Bin} = file:read_file("trigrams.set"), Set = binary_to_term(Bin), Keys = sets:to_list(Set), Size = length(Keys), {M, }* = timer:tc(?MODULE, lookup_all_set, [Set, Keys]), io:format("Module set lookup=\~p micro seconds\~n" ,[M/Size]). lookup_all_set(Set, L) -> lists:foreach(fun(Key) -> sets:is_element(Key, Set) end, L).

Вот что мы получим: 1> lib_trigrams:timer_tests(). Ets ordered Set lookup=1.79964 micro seconds Ets set lookup=0.719279 micro seconds Module sets lookup=1.35268 micro seconds ok

Здесь время - среднее время за одну выборку.

И победитель это...

Ну, здесь всё просто. ETS множество выиграло с большим отрывом. На моей машине, множеству требуется полмикросекунды на выборку - это очень хорошо!

Примечание: выполнение тестов наподобие этих, и вообще измерения сколько требуется конкретной операции, является признаком хорошего стиля программирования. Нам не надо моделировать экстремальные условия и измерять всё подряд, только наиболее долгие операции в нашей программе. Быстрые операции должны быть запрограммированы наиболее красивым способом. Если нам приходится писать невероятно страшный и запутанный код, надо его детально документировать.

А сейчас мы можем написать функцию, которая попробует предсказать, является ли строка действительным английским словом.

Чтобы определить является ли строка английским словом, мы просматриваем все триграммы в строке и проверяем каждую на наличие в вычисленном ранее в списке. Функция is_word делает это.

Скачать lib_trigrams.erl is_word(Tab, Str) -> is_word1(Tab, "\s" ++ Str ++ "\s"). is_word1(Tab, [,,]*=X) -> is_this_a_trigram(Tab, X); is_word1(Tab, [A,B,ClD]) -> case is_this_a_trigram(Tab, [A,B,C]) of true -> is_word1(Tab, [B,ClD]); false -> false end; is_word1(*, _) -> false. is_this_a_trigram(Tab, X) -> case ets:lookup(Tab, list_to_binary(X)) of [] -> false; _ -> true end. open() -> {ok, I} = ets:file2tab(filename:dirname(code:which(?MODULE)) ++

"/trigramsS.tab"), I. close(Tab) -> ets:delete(Tab).

Функции open и close открывают ETS таблицу и заполняют её. Всякий вызов is_word должен "оборачиваться" в эти функции.

Другой трюк, который я использовал здесь — метод, которым я определяю внешний файл с таблицей триграмм. Я положил его в директорию из которой был загружен текущий модуль. code:which(?MODULE) возвращает имя файла, в котором обнаружен объектный код для ?MODULE.

15.6 DETS

DETS предоставляет хранилище кортежей на диске. Максимальный размер DETS файла 2Гб. DETS файлы должны быть открыты перед использованием и правильно закрыты после. Если они не были правильно закрыты, они будут автоматически восстановлены при следующем открытии. Так как восстановление может занять долгое время, важно закрывать их правильно до завершения приложения.

У DETS таблиц параметры совместного доступа отличаются от ETS таблиц. Когда открывается DETS таблица, ей должно быть присвоено глобальное имя. Если два или более локальных процесса откроют DETS таблицу с одинаковым именем и параметрами, они будут иметь к ней совместный доступ. Таблица будет оставаться открытой до тех пор пока все процессы не закроют её (или пока они не завершатся).

Пример: Индекс имён файлов

Мы начнём с примера и ещё одной утилиты, которая нам потребуется для полнотекстового движка, который будет рассматриваться в параграфе 20.4, для mapreduce и параграфа "Индексируем наш диск", на странице 379.

Мы собираемся создать дисковую таблицу, которая поставит соответствие целых числ именам файлов (пронумерует), и т.д. Мы определим функцию filename2index и обратную ей index2filename.

Чтобы реализовать их, мы создадим DETS таблицу и поместим в неё три различных типа кортежей:

{free, N}

N - первый свободный индекс в таблице. Когда мы вставляем новое имя файла в таблицу, ему будет присвоен индекс N.

{FileNameBin, K}

FileNameBin (двоичная строка) соответствует индексу K.

{K, FileNameBin}

K (целое) соответствует файлу FilenameBin.

Заметьте как добавление каждого нового файла, добавляет две новые строки в таблицу: запись Файл -> Индекс и запись Индекс -> Файл. Это сделано из соображений эффективности. Когда ETS или DETS таблицы создаются, только один элемент кортежа может выступать в роли ключа. Поиск кортежа не по ключевому полю возможен, но он очень неэффективен, потому что он приводит к перебору всех значений таблицы. Это достаточно дорогостоящая операция, особенно, когда таблица располагается на диске.

Сейчас давайте напишем программу. Начнём с функций открытия и закрытия DETS таблицы для хранения имён всех наших файлов.

Скачать lib_filenames_dets.erl -module(lib_filenames_dets). -export([open/1, close/0, test/0, filename2index/1, index2filename/1]). open(File) -> io:format("dets opened:\~p\~n" , [File]), Bool = filelib:is_file(File), case dets:open_file(?MODULE, [{file, File}]) of {ok, ?MODULE} -> case Bool of true -> void; false -> ok = dets:insert(?MODULE, {free,1}) end, true; {error,_Reason} -> io:format("cannot open dets table\~n"), exit(eDetsOpen) end. close() -> dets:close(?MODULE).

Код для открытия автоматически инициализирует DETS таблицу вставляя кортеж {free, 1}, если была создана новая таблица. filelib:is_file(File) возвращает true, если файл File существует, в противном случае, она возвращает false. Заметьте, что dets:open_file создаёт новый файл, или открывает существующий, именно поэтому мы проверяем существовал ли файл до вызова dets:open_file.

В этом коде я использовал макрос ?MODULE много раз; ?MODULE преобразуется в имя текущего модуля (lib_filenames_dets). Многие вызовы DETS таблиц требуют уникальный атом — имя таблицы.

Для генерации уникального имени таблицы нам достаточно имени модуля. Так как в системе не может быть загружено два модуля с одним именем, то, следуя этому соглашению везде, мы обоснованно можем считать, что мы присваиваем уникальные имена таблицам.

Я использовал макрос ?MODULE вместо того, чтобы каждый раз писать имя модуля, потому что у меня есть привычка менять названия модулей, пока я пишу код. Если использовать макрос, то даже при изменении имени модуля код останется верным.

Мы уже открыли файл, вставка нового имени файла в таблицу очень проста. Это реализовано как сторонний эффект вызова filename2index. Если имя файла уже сеть в таблице, то возвращается его индекс; в противном случае рассчитывается новый

индекс и таблица обновляется, на этот раз тремя кортежами:

Скачать lib_filenames_dets.erl filename2index(FileName) when is_binary(FileName) -> case dets:lookup(?MODULE, FileName) of [] -> [{*,Free}] = dets:lookup(?MODULE, free), ok = dets:insert(?MODULE, [{Free,FileName},{FileName,Free},{free,Free+1}]), Free; [{*,N}] -> N end.

Заметьте как мы записываем три кортежа в таблицу. Второй аргумент dets:insert должен быть либо кортежем, либо списком кортежей. Заметьте также, что имя файла представлено двоичным значением. Это сделано из соображений эффективности. Вообще хорошо иметь привычку использовать двоичные данные для представления строк в ETS и DETS таблицах.

HYPERLINK\ \l\ "FOOTNOTE-3"\ Внимательный читатель может заметить, что в примере есть возможный race condition (состояние гонки) в filename2index. Если два параллельных процесса вызовут dets:lookup до вызова dets:insert, в этом случае filename2index вернёт некорректное значение. Чтобы код работал, мы должны убедиться, что он работает в единственном экземпляре в каждый момент времени.

Преобразовать индекс в имя файла просто:

Скачать lib_filenames_dets.erl index2filename(Index) when is_integer(Index) -> case dets:lookup(?MODULE, Index) of [] -> error; [{_,Bin}] -> Bin end.

Небольшое дизайнерское решение. Что будет произойдёт, если мы вызовем index2filename(Index), а в таблице не будет имени файла привязанного к этому индексу? Мы можем завершиться вызвав exit(ebadIndex). Но мы выбрали более элегантную альтернативу: мы просто возвращаем атом error. Вызывающий код может сам выбрать между правильным и неправильным значениями, так как все правильные имена файлов имеют тип binary.

Отметьте также guard tests в filename2index и index2filename. Они проверяют правильный ли тип имеют аргументы. Вообще, это правильно, потому что ввод данных неверного типа в DETS таблицу может вызвать ситуации, которые очень трудно отлаживать после. Мы можем представить запись данных с неверным типом в таблицу и чтение её через несколько месяцев, когда уже поздно делать что-либо. Потому, лучше проверять, что все данные верны перед вставкой в таблицу.

15.7 О чём мы не сказали? ETS и DETS таблицы поддерживают набор операций, которые мы не рассмотрели в этой главе. Эти операции разделяются на следующие категории:

получение и удаление объектов по шаблону

преобразования между ETS и DETS таблицами и между ETS таблицами и файлами на диске

определение ресурсов используемых таблицей

пересечение (traversing) всех элементов в таблице

восстановление повреждённых DETS таблиц

визуализация таблиц

Вы можете найти больше информации на страницах руководства, доступного на HYPERLINK "http://www.erlang.org/doc/man/ets.html" http://www.erlang.org/doc/man/ets.html и HYPERLINK "http://www.erlang.org/doc/man/dets.html" http://www.erlang.org/doc/man/dets.html.

В заключение скажем, что ETS и DETS таблицы изначально разрабатывались для реализации Mnesia. Мы ещё не говорили о Mnesia, так как она — тема главы 17: База данных Эрланга, страница 313. Mnesia — база данных реального времени написанная на Эрланге. Mnesia использует ETS и DETS таблицы внутри себя, и много функций из модулей ETS и DETS предназначены для внутреннего пользования Mnesia. Mnesia может выполнять все типы операций, которые невозможно сделать на чистых ETS и DETS таблицах. Например, мы можем создавать индексы не только по первичному ключу, подобный пример мы использовали при двойной вставке в функции filename2index. Mnesia на самом деле создаёт несколько ETS или DETS таблиц, для реализации такого функционала, но это создание спрятано от пользователя.

15.8 Листинги кода

Скачать lib_trigrams_complete.erl

%% --- %% Excerpted from "Programming Erlang", %% published by The Pragmatic Bookshelf. %% Copyrights apply to this code. It may not be used to create training material, %% 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 information. %%---

-module(lib_trigrams). -export([for_each_trigram_in_the_english_language/2, make_tables/0, timer_tests/0, open/0, close/1, is_word/2, how_many_trigrams/0, make_ets_set/0, make_ets_ordered_set/0, make_mod_set/0, lookup_all_ets/2, lookup_all_set/2]). -import(lists, [reverse/1]).

make_tables() -> {Micro1, N} = timer:tc(?MODULE, how_many_trigrams, []), io:format("Counting - No of trigrams=\~p time/trigram=\~p\~n",[N,Micro1/N]), {Micro2, Ntri} = timer:tc(?MODULE, make_ets_ordered_set, []), FileSize1 = filelib:file_size("trigramsOS.tab"),

io:format("Ets ordered Set size=\~p time/trigram=\~p\~n",[FileSize1/Ntri, Micro2/N]), {Micro3, *} = timer:tc(?MODULE, make_ets_set, []), FileSize2 = filelib:file_size("trigramsS.tab"), io:format("Ets set size=\~p time/trigram=\~p\~n",[FileSize2/Ntri, Micro3/N]), {Micro4, } = timer:tc(?MODULE, make_mod_set, []), FileSize3 = filelib:file_size("trigrams.set"), io:format("Module sets size=\~p time/trigram=\~p\~n",[FileSize3/Ntri, Micro4/N]).

make_ets_ordered_set() -> make_a_set(ordered_set, "trigramsOS.tab"). make_ets_set() -> make_a_set(set, "trigramsS.tab").

make_a_set(Type, FileName) -> Tab = ets:new(table, [Type]), F = fun(Str, _) -> ets:insert(Tab, {list_to_binary(Str)}) end, for_each_trigram_in_the_english_language(F, 0), ets:tab2file(Tab, FileName), Size = ets:info(Tab, size), ets:delete(Tab), Size.

make_mod_set() -> D = sets:new(), F = fun(Str, Set) -> sets:add_element(list_to_binary(Str),Set) end, D1 = for_each_trigram_in_the_english_language(F, D), file:write_file("trigrams.set", [term_to_binary(D1)]).

timer_tests() -> time_lookup_ets_set("Ets ordered Set", "trigramsOS.tab"), time_lookup_ets_set("Ets set", "trigramsS.tab"), time_lookup_module_sets().

time_lookup_ets_set(Type, File) -> {ok, Tab} = ets:file2tab(File), L = ets:tab2list(Tab), Size = length(L), {M, _} = timer:tc(?MODULE, lookup_all_ets, [Tab, L]), io:format("\~s lookup=\~p micro seconds\~n",[Type, M/Size]), ets:delete(Tab).

lookup_all_ets(Tab, L) -> lists:foreach(fun({K}) -> ets:lookup(Tab, K) end, L).

time_lookup_module_sets() -> {ok, Bin} = file:read_file("trigrams.set"), Set = binary_to_term(Bin), Keys = sets:to_list(Set), Size = length(Keys), {M, _} = timer:tc(?MODULE, lookup_all_set, [Set, Keys]), io:format("Module set lookup=\~p micro seconds\~n",[M/Size]).

lookup_all_set(Set, L) -> lists:foreach(fun(Key) -> sets:is_element(Key, Set) end, L).

how_many_trigrams() -> F = fun(_, N) -> 1 + N end, for_each_trigram_in_the_english_language(F, 0).

%% An iterator that iterates through all trigrams in the language

for_each_trigram_in_the_english_language(F, A0) -> {ok, Bin0} = file:read_file("354984si.ngl.gz"), Bin = zlib:gunzip(Bin0), scan_word_list(binary_to_list(Bin), F, A0).

scan_word_list([], _, A) -> A; scan_word_list(L, F, A) -> {Word, L1} = get_next_word(L, []), A1 = scan_trigrams([\$\sIWord], F, A), scan_word_list(L1, F, A1).

%% scan the word looking for \r\n %% the second argument is the word (reversed) so it %% has to be reversed when we find \r\n or run out of characters

get_next_word([\$\r,\$\n|T], L) -> {reverse([\$\s|L]), T}; get_next_word([H|T], L) -> get_next_word(T, [H|L]); get_next_word([], L) -> {reverse([\$\s|L]), []}.

scan_trigrams([X,Y,Z], F, A) -> F([X,Y,Z], A);

scan_trigrams([X,Y,Z|T], F, A) -> A1 = F([X,Y,Z], A), scan_trigrams([Y,Z|T], F, A1); scan_trigrams(_, _, A) -> A.

%% access routines %% open() -> Table %% close(Table) %% is_word(Table, String) -> Bool

is_word(Tab, Str) -> is_word1(Tab, "\s" ++ Str ++ "\s").

is_word1(Tab, [_,_,_]=X) -> is_this_a_trigram(Tab, X); is_word1(Tab, [A,B,C|D]) -> case is_this_a_trigram(Tab, [A,B,C]) of true -> is_word1(Tab, [B,C|D]); false -> false end; is_word1(_, _) -> false.

is_this_a_trigram(Tab, X) -> case ets:lookup(Tab, list_to_binary(X)) of [] -> false; _ -> true end.

open() -> {ok, I} = ets:file2tab(filename:dirname(code:which(?MODULE)) ++ "/trigramsS.tab"), I.

close(Tab) -> ets:delete(Tab).

notes

^1\ ^Я называю это итераторм, но если быть точным, то это оператор свёртки (fold) очень похожий на lists:foldl.

^2\ ^1. Взято с http://www.dcs.shef.ac.uk/research/ilash/Moby/

^3\ ^Напротив абзаца прикольный жёлтенький значок.