Databases + OTP

Lectures F10 + F11 (Chapters 19..22)

Joe Armstrong

Ericsson AB

W8/F10 - ETS, DETS Mnesia Database,Debugging etc. Chapters 19, 20,21

Several databases have been implemented in Erlang:

- Mnesia - Real-time database. Fail-safe. Hot standby.
- CouchDB - “Document Database” stores JSON documents. Replicates over a cluster. Used by CERN. Stored data that led to the discovery of the Higgs boson.
- Riak - Key-Value “Eventually Consistant” database. “Internet scale”.
- Cloudant - Cloud database. Data layer “as a service”.
- Amazon - SimpleDB

Architecture

- Erlang has only non-destructive data types. So how can we store gigantic quantites of data?
- We cheat!
- ETS (Erlang Term Storage) is a *destructive* tuple store.
- DETS (Disk ETS) is a *destructive* disk store, with a journal for crash recovery.
- Basho have implemented their own disk store (bitcask) as the backend for Riak.
- CouchDB uses a B* tree with concurrency control.
- Replicating data is very very difficult.

ETS

- ETS tables behave like linked processes.
- ETS `named_tables` behave like registered processes.
- Very fast in-memory hash tables.
- No transations.
- Can share data between processes, but there is no locking when writing to the tables.
- Can save and restore to file.
- Object are stored off-stack and off-heap. This means objects in ETS tables do not slow down the garbage collection, but lookups copy data from tables to the heap.

ETS sets

```
1> ets:new(phone_numbers, [set,named_table]).
phone_numbers
2> ets:insert(phone_numbers, {joe,1234}).
true
3> ets:insert(phone_numbers, {bill,123456}).
true
4> ets:lookup(phone_numbers, joe).
[{joe,1234}]
5> ets:insert(phone_numbers, {joe,2234}).
true
6> ets:lookup(phone_numbers, joe).
[{joe,2234}]
```

ETS bags

```
7> ets:delete(phone_numbers).
true
9> ets:new(phone_numbers, [bag,named_table]).
phone_numbers
10> ets:insert(phone_numbers, {joe,1123}).
true
11> ets:insert(phone_numbers, {joe,2244}).
true
12> ets:lookup(phone_numbers, joe).
[{joe,1123},{joe,2244}]
```


ETS save and restore

- Save the table and exit:

```
13 > ets:tab2file(phone_numbers, "foo").
ok
15>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a
```

- Some time later ..

```
joe:paradis joe$ erl
Erlang/OTP 17 [RELEASE CANDIDATE 1] [erts-6.0] [source] [64-bit]

Eshell V6.0 (abort with ^G)
1> ets:file2tab("foo").
{ok,phone_numbers}
2> ets:lookup(phone_numbers, joe).
[{joe,1123},{joe,2244}]
```

Putting it all togther

```
-module(index).
-compile(export_all).

make() ->
   Ets = ets:new(index, [bag, named_table]),
   {ok, C} = re:compile("(?:\\,|\\.|\\;|\\:|\\s|[0-9]+)+"),
   {ok, Files} = file:list_dir("."),
   Orgs = [F || F <- Files, filename:extension(F) == ".org"],
   [add_index(File, C) || File <- Orgs],
   ets:tab2file(Ets, "index.ets"),
   ets:delete(index).

add_index(File, C) ->
   {ok, Bin} = file:read_file(File),
   [ets:insert(index, {to_lower(I),File}) || I <- re:split(Bin, C)].

to_lower(B) -> list_to_binary(string:to_lower(binary_to_list(B))).

lookup() ->
   ets:file2tab("index.ets"),
   V = ets:lookup(index,<<"armstrong">>),
   ets:delete(index),
   V.
```

Building and querying the index

```
1> index:make().
true
```

```
2> index:lookup().
[{<<"armstrong">>,"bug.org"},
 {<<"armstrong">>,"f10-f11.org"},
 {<<"armstrong">>,"f12-f13.org"},
 {<<"armstrong">>,"f19-f20.org"},
 {<<"armstrong">>,"f2-f3.org"},
 {<<"armstrong">>,"f4-f5.org"},
 {<<"armstrong">>,"f6-f7.org"},
 {<<"armstrong">>,"f8-f9.org"},
 {<<"armstrong">>,"plan.org"},
 {<<"armstrong">>,"test_slides.org"}]
```


ETS reference

- http://learnyousomeerlang.com/ets.

- http://www.erlang.org/doc/man/ets.html.

DETS

- Data is stored on disk.
- Interface “similar to” ETS (but not identical).
- Has crash recovery. DETS tables are repaired on restart if they were not closed properly. This can happen after a system crash.

DETS

- Yawn ...
- Same as ETS (almost) RTFM.
- There are some exercises on ets and dets.

Mnesia Creating a database

```
$ cd mnesia
$ erl
1> mnesia:create_schema([node()]).
ok
2> init:stop().
ok
$ ls
```


Mnesia: Creating a table

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

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().
```


Adding and removing data

```
example_tables() ->
   [%% The shop table
    {shop, apple, 20, 2.3},
    {shop, orange, 100, 3.8},
    ...]
    %% The cost table
    {cost, apple, 1.5},
    {cost, orange, 2.4},
    ...
   ].

create_tables() ->
   F = fun() ->
        lists:foreach(fun mnesia:write/1, example_tables())
      end,
   mnesia:transaction(F).
```


Simple Mnesia Queries

- Reading data:

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

Transactions

- Mnesia is interfaced through transactions.

- Transactions either succeed or fail. If they fail the state of the database is unvhanged.

```
some_function() ->
    F = fun(Args) -> {aborted, Reason} | {atomic, Result},
    mnesia:transaction(F).
```


Mnesia: advanced

- Tables can be repliced in memory on disk and across machines.
- Tables can be striped across machines.
- In a fault-tolerent system data is replicted on different nodes. There is usually a master node and a hot standby.

Profiling

```
> cprof:start().
6505
8> orgmode_parse:transform(['f10-f11.org']).
Transforming:"f10-f11.org"
...
9> cprof:pause().
6505
10> cprof:analyse(orgmode_parse).
{orgmode_parse,10248,
            [{{orgmode_parse,is_stop,2},3638},
             {{orgmode_parse,get_body,3},3361},
             {{orgmode_parse,get_line,2},2619},
..
```


Coverage

- Finding code that has never run
- Finding hot spots

```
1> cover:start().
{ok,<0.34.0>}
2> cover:compile(orgmode_parse).
{ok,orgmode_parse}
3> orgmode_parse:transform(['f10-f11.org']).
Transforming:"f10-f11.org"
..
Created:f10-f11.org.tex
Z:ok
ok
4> cover:analyse_to_file(orgmode_parse).
{ok,"orgmode_parse.COVER.out"}
```


W8/F11 - OTP - Chapter 22

The Road to the gen server

- Write a client-server all in one module.

- Split the code into two modules. One with only sequential code, the other with concurrency primitives.

counter0 - in one module

```
-module(counter0).
-export([start/0, loop/1, tick/1, read/0, clear/0]).

start() ->
  register(counter0, spawn(counter0, loop, [0])).

tick(N) -> rpc({tick, N}).
read() -> rpc(read).
clear() -> rpc(clear).

loop(State) ->
  receive
     {From, Tag, {tick, N}} ->
        From ! {Tag, ack},
        loop(State + N);
     {From, Tag, read} ->
        From ! {Tag, State},
        loop(State);
     {From, Tag, clear} ->
        From ! {Tag, ok},
        loop(0)
  end.
```

counter0 (continued)

```
rpc(Query) ->
   Tag = make_ref(),
   counter0 ! {self(), Tag, Query},
   receive
     {Tag, Reply} ->
        Reply
   end.
```

```
$ erl
1> counter0:start().
true
2> counter0:tick(5).
ack
3> counter0:tick(10).
ack
4> counter0:read().
15
```


Reorganize counter0 into two files

- The server:

```
-module(gen_server_lite).
-export([start/2, loop/2, rpc/2]).

start(Mod, State) ->
  register(Mod, spawn(gen_server_lite, loop, [Mod, State])).

loop(Mod, State) ->
  receive
    {From, Tag, Query} ->
       {Reply, State1} = Mod:handle(Query, State),
       From ! {Tag, Reply},
       loop(Mod, State1)
     end.

rpc(Mod, Query) ->
   Tag = make_ref(),
   Mod ! {self(), Tag, Query},
   receive
     {Tag, Reply} ->
       Reply
   end.
```

... and the client

```
-module(counter1).
-compile(export_all).

start() -> gen_server_lite:start(counter, 0).

tick(N) -> gen_server_lite:rpc(counter, {tick, N}).
read() -> gen_server_lite:rpc(counter, read).
clear() -> gen_server_lite:rpc(counter, clear).

handle({tick,N}, State) -> {ack, State+N};
handle(read, State) -> {State, State};
handle(clear,_) -> {ok, 0}.
```


Running the program

```
1> counter1:start().
true
2> counter1:read().
0
3> counter1:tick(10).
ack
4> counter1:tick(14).
ack
5> counter1:read().
24
6> counter1:clear().
ok
7> counter1:read().
0
8> counter1:read().
```

Abstracting out concurrency

- counter.erl has no send receive spawn etc.
- *Everything to do with concurrency is in* `gen_server_lite.erl`

The real gen_server

```
-module(counter2).
-export([start/0, init/1, tick/1, read/0, clear/0, handle_call/3]).
%% the real gen_server

start() -> gen_server:start_link({local,counter2}, counter2, 0, []).

init(N) -> {ok, N}.

tick(N) -> gen_server:call(counter2, {tick, N}).
read() -> gen_server:call(counter2, read).
clear() -> gen_server:call(counter2, clear).

handle_call({tick,N}, _From, M) -> {reply, ack, M+N};
handle_call(read, _From, N) -> {reply, N, N};
handle_call(clear,_From, _State) -> {reply, ok, 0}.
```


OTP behaviours

- gen_server – client server model
- gen_fsm – finite state machine
- supervisor