databases

the PMT way

Databases

SQLDB

Overview

- 1. Introduction
- 2. SQLDB
- 3. Replication
- 4. Recovery from failure
- 5. Hardware

Introduction

databases
the PMT way

Who are we?

- Operating in SA and UK
- Been using Erlang since 2001
- Products for Financial and Telecoms purposes
- Emerald, Amber and Crimson

Powered by
Pattern Matched

SQLDB

databases
the PMT way

erlang database drivers

- Only one “single API” to multiple db’s
- OTP odbc application

Powered by
Pattern Matched

databases
the PMT way

erlang database drivers

but many incompatible and custom drivers:

- erlang-mysql-driver
- Yxa
- pgsql
- etc

Powered by
Pattern Matched

our requirements

- single, database-agnostic API
- modular design
- support for connection pools

our requirements

- we picked PostgreSQL as our preferred db
- It is robust
- many features
- VERY stable

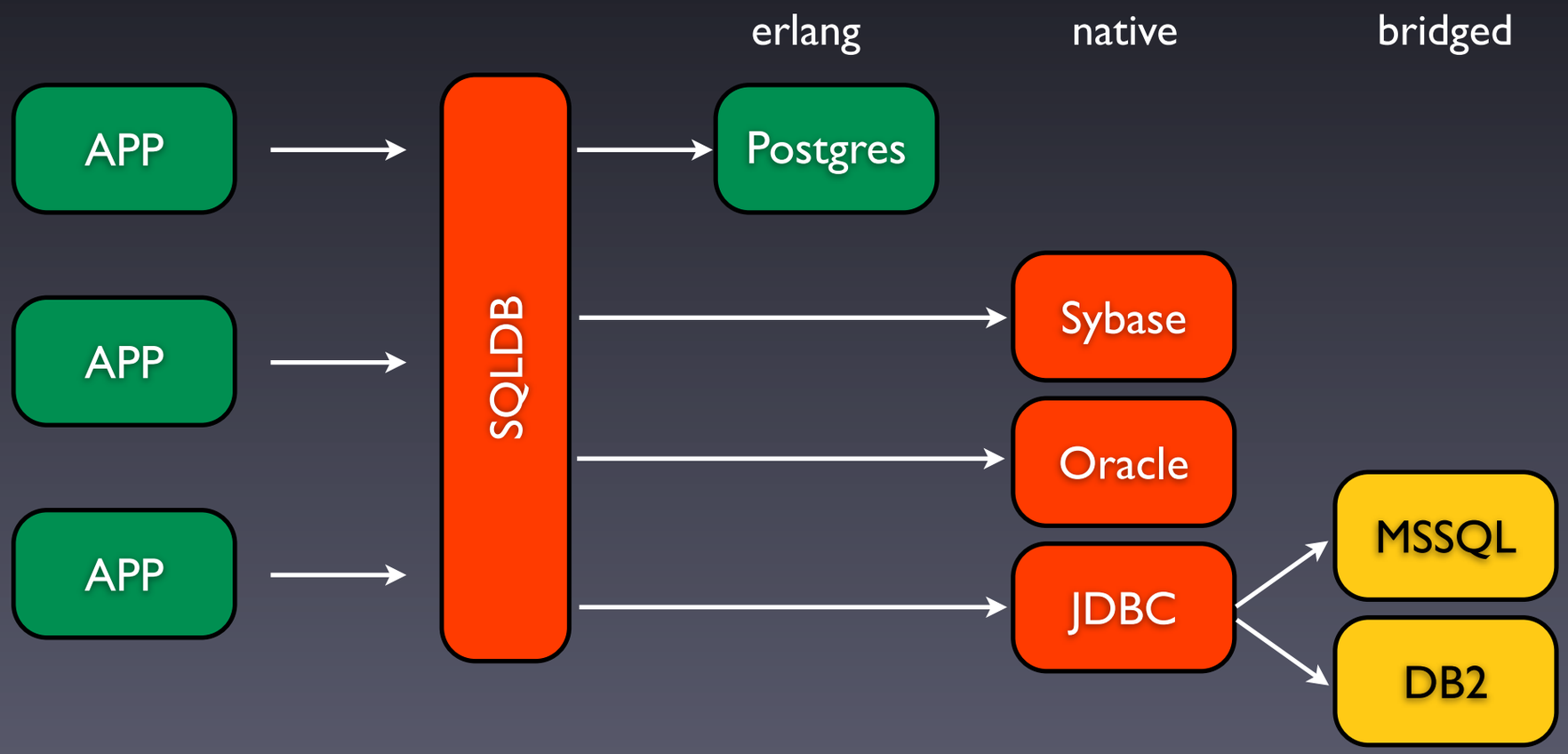
our PG solution

- native erlang code
- in use from 2004
- minimum performance 600 TPS/connection
- Ideal for connection pool model

databases

the PMT way

SQLDB



Powered by
Pattern Matched

the basic API - the pool

```
                                        1
ok = sqldb_pool:acquire_connection("ThePool"),
try
      2
  ...
  after
                                              3
    ok = sqldb_pool:release_connection()
  end.
```

1. Acquire a connection - potential to take a long time
2. Do your work
3. Release the connection

the basic API - transactions

```
ok = sqldb_pool:acquire_connection("ThePool"),
try
    ok = sqldb_api:begin_transaction(),   (1)
    ...
    ok = sqldb_api:commit_transaction(),  (2)
  after
    ok = sqldb_pool:release_connection()
  end.
```

1. start a transaction
2. commit or rollback the transaction

databases
the PMT way

the basic API - transactions

```
ok = sqldb_pool:acquire_connection("ThePool"),
try
  Fun =
   fun()(1)->
     sqldb_api:simple_query(“select count(*) from users;”(2),[])
   end,
    {ok,
      [{sqldb_result(3),[{column,"count",int,8}](4),
      [{row,UserCount}]}]}(5) =
    sqldb_api:transaction(Fun)(6)
   after
    ok = sqldb_pool:release_connection()
 end.
```

1. Wrap the statements in a Fun
2. Simple SQL
3. Multiple result sets
4. Column Information
5. Data returned from the query
6. Everything executed as a single transaction

Powered by
Pattern Matched

databases
the PMT way

how do you handle 9,000,000 rows?

```
SQL = “select * from users;”,
{ok,[{sqldb_result,Columns,Rows}] = sqldb_api:simple_query(SQL,[]),
...
```

```
Crash dump was written to: erl_crash.dump
eheap_alloc: Cannot allocate 1824525600 bytes of memory (of type "heap").
```

- Do not accumulate terms in one process!
- Yes you can use a (server side) cursor
- or...

Powered by
Pattern Matched

callback queries

```
SQL = “select * from users;”,                    1
CallBackModule = user_callback,                  2
CallBackState = [InitialState],                  3

ok = sqldb_connection:callback_query(Query, Params, CallBackModule, CallBackState),
                                                 4
```

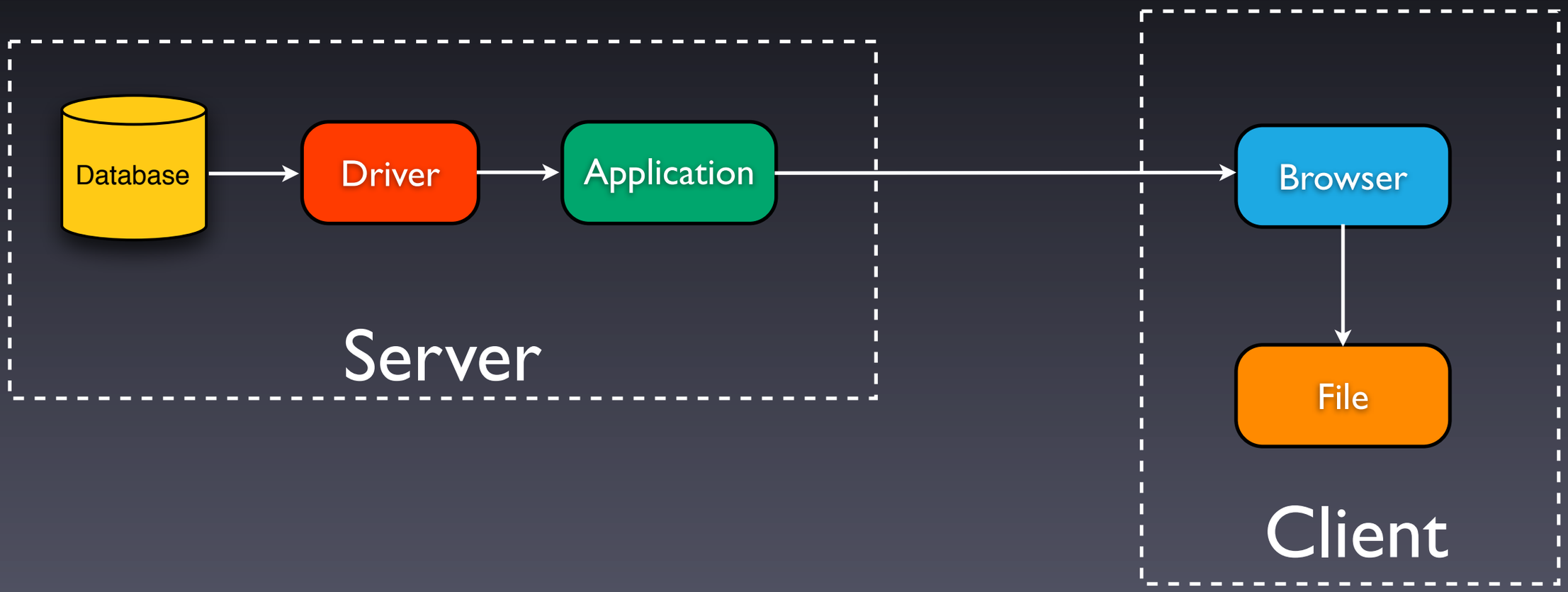
1. The same query
2. Name of the callback module
3. Initial state
4. callback_query function

callback queries

```
sqldb_init(InitialState) ->
  {ok, InitialState}.

sqldb_result_set_begin(ColumnInfo,ModuleState) ->
   {ok, ModuleState}.

sqldb_row(Row, ModuleState) ->
  {ok, ModuleState}.

sqldb_result_set_complete(_Command, ModuleState) ->
  {ok, ModuleState}.

sqldb_results_complete(ModuleState) ->
  {ok,ok,State}.
```

1. Initialisation
2. Function called when a new result set is being processed
3. Called for every row in the result set
4. Called when a result set finishes
5. Called when the query is done

interesting consequences



1. No accumulation of data anywhere
2. Flow control all the way from the client's file to the database
3. Very low latency - data is immediately available
4. Very efficient - because we have concurrency for free

next for SQLDB?

1. Our goal is to make SQLDB the single Erlang database API
2. We will soon make the source available... but
3. We still need drivers for the native erlang odbc application
4. And help to make it perfect...
5. Contact rvg@patternmatched.com if you are interested to help!

Replication

databases

the PMT way

mnesia vs SQL

Yes

Mnesia

1. Fast Reads and writes
2. Distributed Transactions
3. Easy!

SQL

1. Huge capacity
2. Fast queries
3. Complex joins
4. Strict Integrity if you use it

No

Mnesia

1. Indexes are terrible
2. Never store “bulk data”
3. Practical size limits
4. No real data integrity (hacks & cheats)

SQL

1. Hard to make redundant
2. Very hard to scale - you use hardware
3. Hard to replicate

Powered by

Pattern Matched

databases

the PMT way

mnesia AND SQL

- we use both
- sometimes Mnesia leads, sometimes SQL does
- we maintain a single, abstract model for both databases

Powered by

Pattern Matched

databases

the PMT way

First the model

```
sql_model() ->
  [#sqldb_table{tablename=vcr_voucher_pin,
                fields=[#sqldb_field{fieldname=voucher_id,    type=int8,
                        #sqldb_field{fieldname=seed,          type=int4},
                        #sqldb_field{fieldname=hash,          type=bytea},
                        #sqldb_field{fieldname=crypto_format, type=int4},
                        #sqldb_field{fieldname=key_index,     type=int4},
                        #sqldb_field{fieldname=kcv,           type=char,length=6}],
                pk = [voucher_id],
                unique = [ #sqldb_unique{ unique_id = 1,
                                          fields=[hash] }],
                fk         = [#sqldb_fk{fk_id = 1,
                                        on_delete_cascade = true,
                                        fields = [voucher_id],
                                        ftable = vcr_voucher_base,
                                        fkeys  = [voucher_id]},
                              #sqldb_fk{fk_id  = 2,
                                        on_delete_cascade = true,
                                        ftable = cem_crypto_format,
                                        fields = [crypto_format],
                                        fkeys  = [crypto_format]
                                        }]}
  ].
```

Powered by

Pattern Matched

databases

the PMT way

and in mnesia...

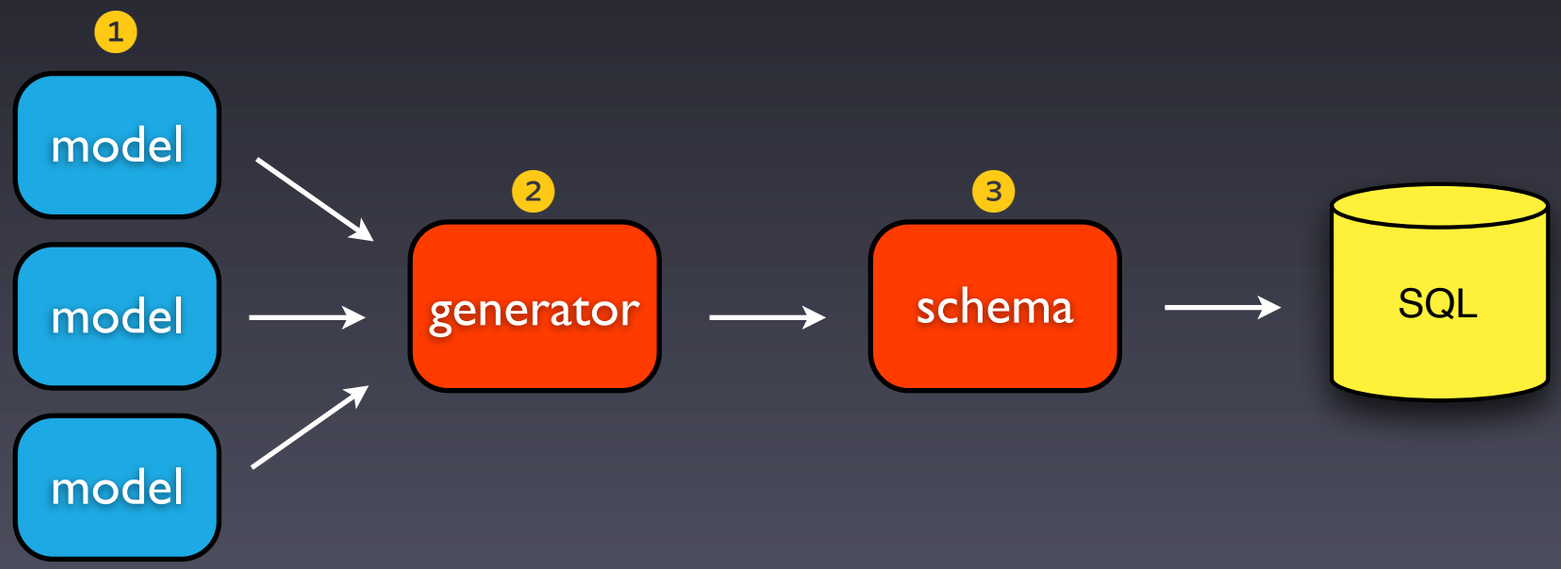
```
-record(user,{username,
                    company_id             = 1,
                    accounting_entity_id,
                    name                   = "",
                    surname                = "",
                    id_nr                  = "",
                    contact_nr             = "",
                    creation_date          = "",
                    msisdn                 = undefined,
                    address1               = "",
                    address2               = "",
                    address3               = "",
                    city                   = "",
                    code                   = "",
                    email_address          = "",
                    credential_type        = 1
                   }).
...
table_def() ->
  {emr_user,fields(),[replication,{replication_callback,asr_replication}]}.
...
```

Powered by

Pattern Matched

databases
the PMT way

together...

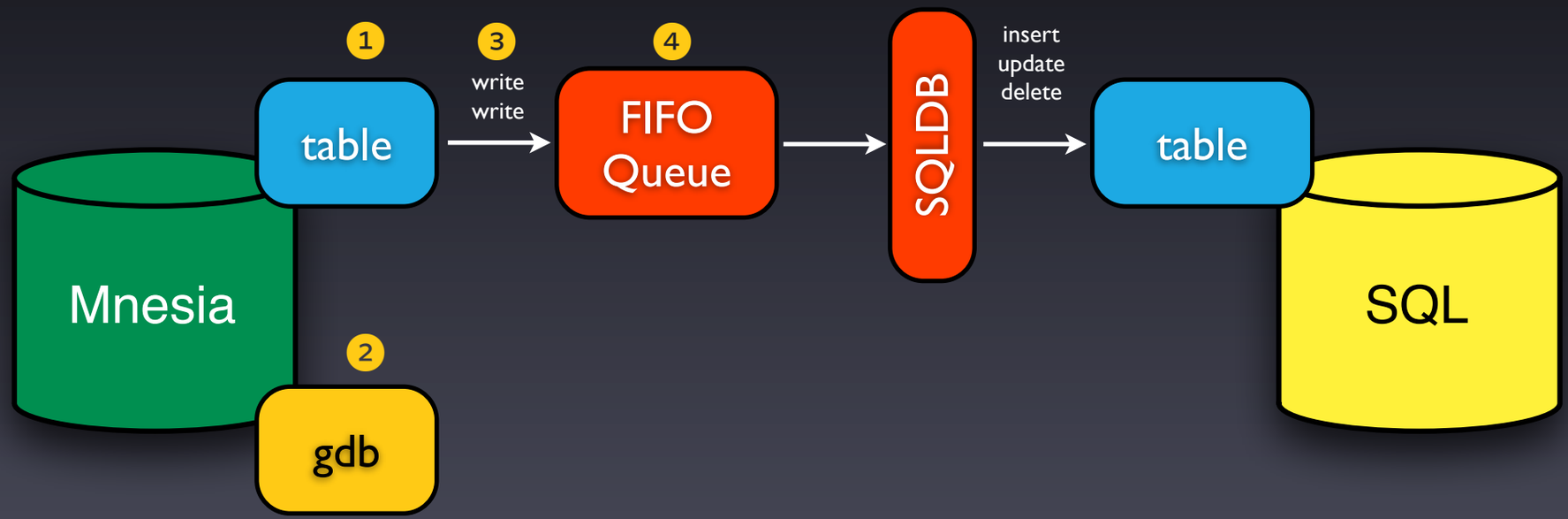


1. Use the abstract model for all the objects to
2. generate a database specific
3. schema

```
CREATE TABLE vcr_voucher_pin (
    voucher_id bigint NOT NULL,
    seed integer,
    encrypted_pin bytea,
    hash bytea,
    crypto_format integer,
    key_index integer,
    kcv character(6)
);
```

Powered by
Pattern Matched

mnesia vs SQL



1. table configured for replication
2. gdb hooks into the mnesia activity mechanism
3. fast module that intercepts writes and queue events in same transaction
4. strict fifo queue that converts events into SQL and push them out via SQLDB

databases
the PMT way

this means

- Both SQL and mnesia schemas are GENERATED
- it is simple to replicate to SQL
- The model ensures consistency

Powered by
Pattern Matched

Dealing with Failure and Recovery

Why and When?

- complete loss of all mnesia nodes
- network partitioning
- Point-in-time recovery (For testing etc)

application design

we solve this by design...

- all critical data is also in SQL
- in most cases SQL leads... because PostgreSQL is fast
- a full recovery is one command:

```
execute() ->
  ok = lock_out_users(),
  ok = stop_services(),
  ok = clear_queues(),
  ok = clear_tables(),
  ok = recover_records(),
  ok = set_counters(),
  ok = clear_queues(),
  ok = start_services(),
  ok = open_for_users().
```

recovery

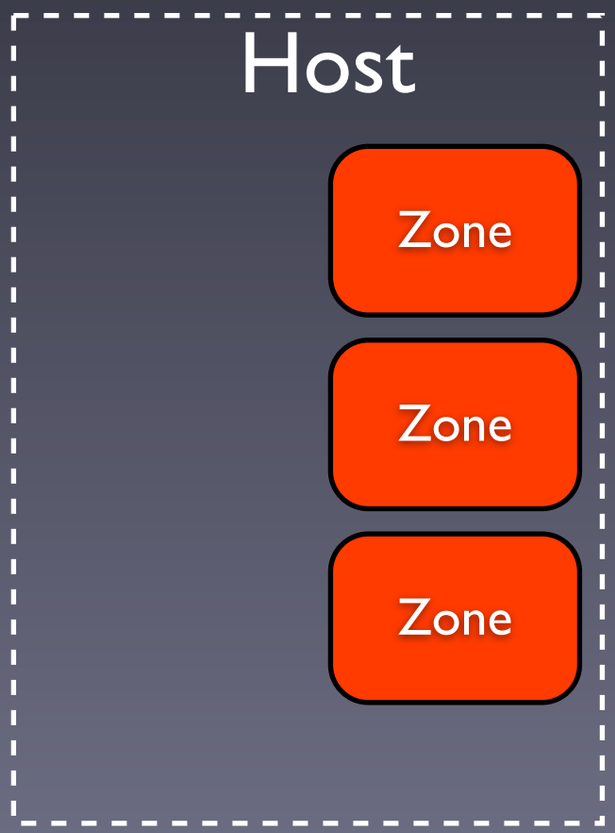
copying back data is trivial...

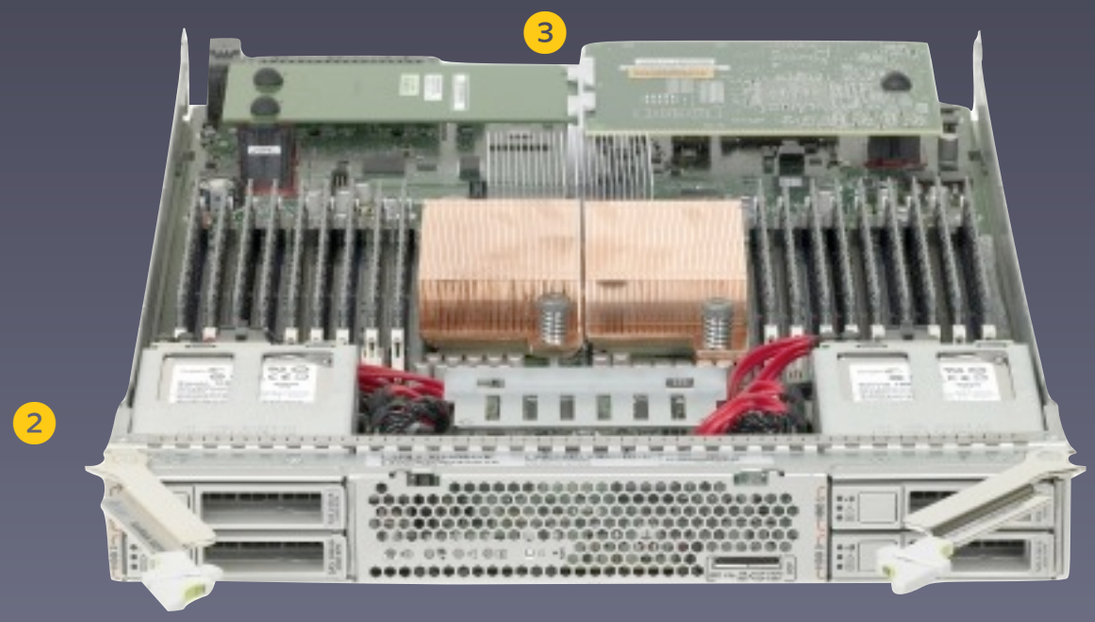
- we use the sqldb callback query feature
- combined with dirty writes
- finish off with full integrity test
- the model makes this process automatic
- we use the same code for all tables

A Quick look at hardware

Servers

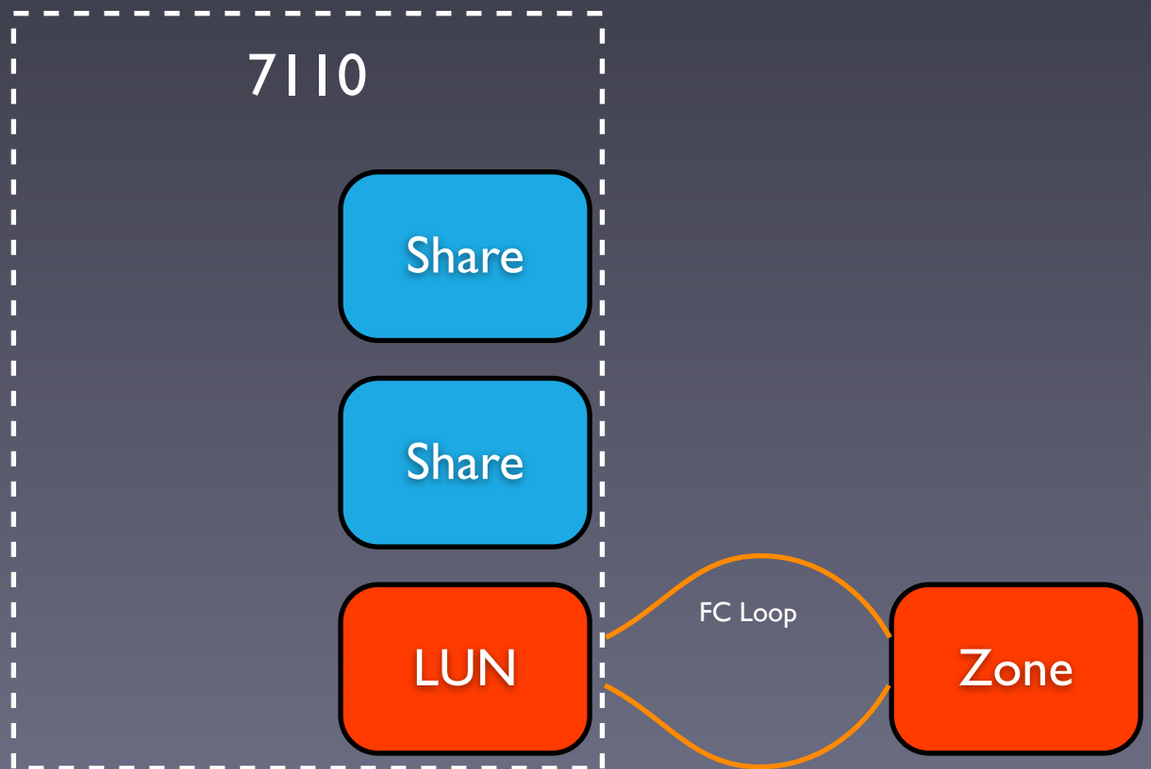
1. Sun Blade 6000 Chassis
2. Sun 6270 Blades
3. Fibre Channel HA





Storage

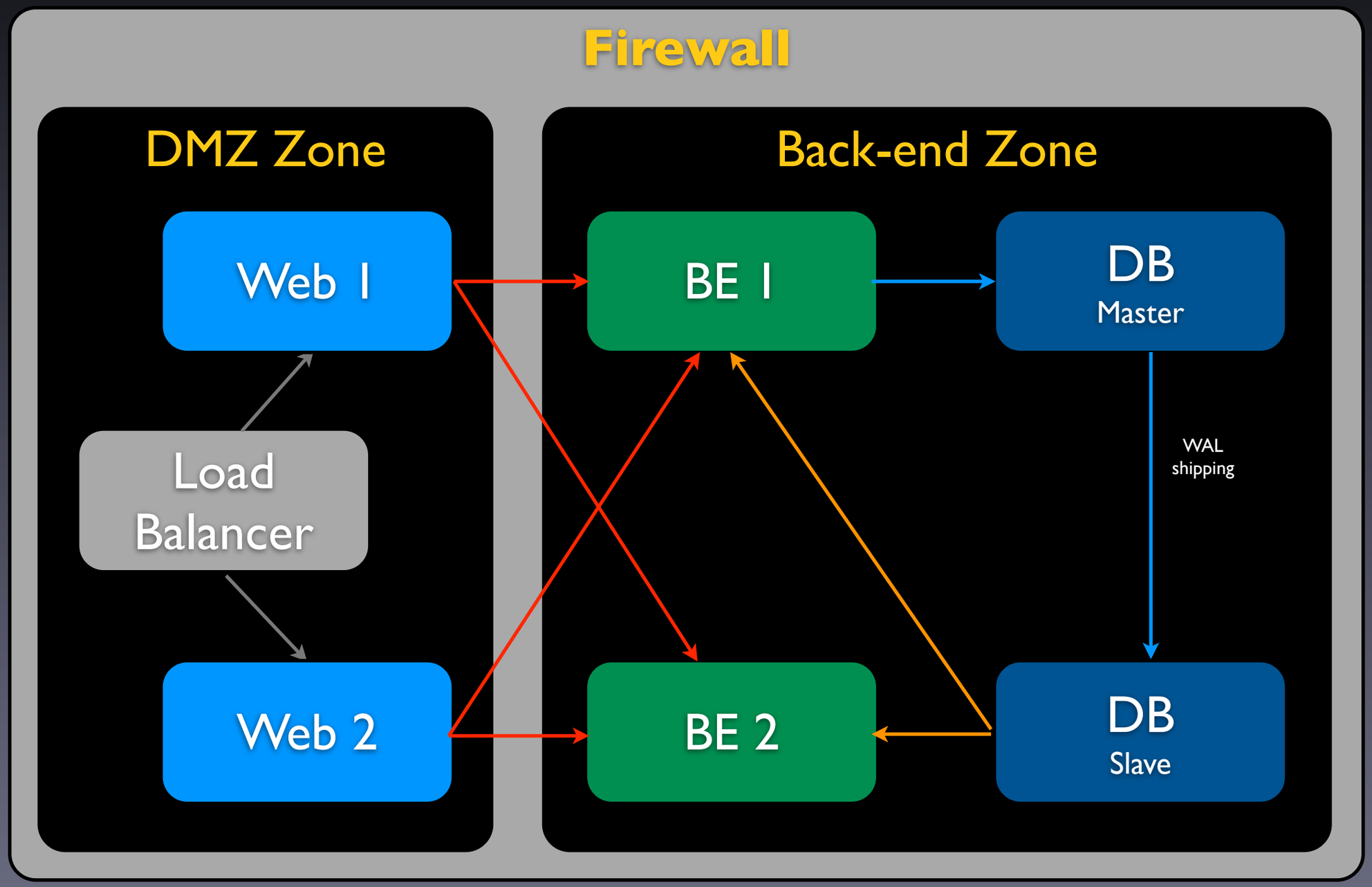
1. Oracle OpenStorage 7110/7310
2. Fibre Channel NAS
3. 4mbps



Open Storage 7110

Brocade FC Switch

The System


Firewall
DMZ Zone
Back-end Zone
Web 1
BE 1
DB
Master
Load
Balancer
WAL
shipping
Web 2
BE 2
DB
Slave

Thank You

- Questions?