Seminar Report: Opty seminar

Oriol Martínez Acón & Imanol Rojas Pérez

December 16, 2021

1 Introduction

This seminar is about Opty, which is an implementation of a server with Optimistic Concurrency Control. The components of the system are: Clients, that do requests, transaction handlers bound to the clients that are responsible for communicating and handling the requests between the Clients and the Transaction Server, the store that is made of entries and, finally the validator which takes the transaction handler commits and validates them.

Transactions in Opty have three phases: Working, Validation and Update in the store. The amount of successes in the system is going to be analyzed in this memo, varying the parameters of the software, changing the execution to remote execution and also implementing forward validation instead of backward validation(which is the one that comes implemented in the vanilla version of Opty). In backwards validation each transaction checks that no other transaction has affected the data it has read before committing. If the conflicting modifications are discovered, the committing transaction is rolled back and can be reopened.

2 Experiments

In this section some experiments are going to be carried out to see the performance changes of the program when some of its parameters suffer variations. Also experiments on distributed execution and forward validation are going to be conducted in this section.

Is important to remark that in all the experiments the percentage of OKs are the mean of the percentage from 3 different clients.

2.1 Performance experiments

For the experiments in this section the base configuration of the Opty parameters will be 3 Clients, 10 Entries, 3 RDxTR, 2 WRxTR and 2 seconds of execution. The experiments to analyze the performance of the Opty software in its vanilla version are the following.

2.1.1 Variation of the number of clients

In this experiment the number of clients in the system is increased to see the outcome. The next figure shows the result obtained.

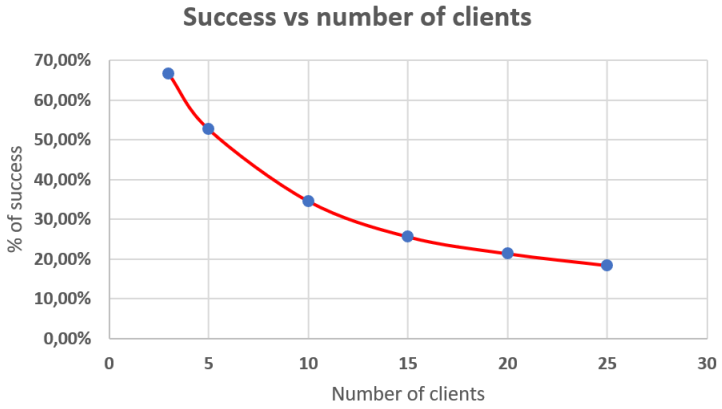

Figure 1: Success rate in % vs number of clients Graph

Increasing the number of clients in the system while maintaining the number of entries has a worsening effect in terms of success rate percentage. The explanation for this is simple. If the number of client increases while the number of entries stays the same, and all the entries have the same probability of being chosen by the clients, it is more probable that one or more clients end up requesting read/write operations for the same entries and thus, causing more aborted requests. This lowers the success rate.

0% success rate could be reached (for some clients) if the number of clients is high enough to make most of the transactions abort.This can be seen in the next figure, where the number of clients is 1500 and the number of entries is 10. Some of the clients have a 0% success rate, that means that they are not served.

```
506: Transactions TOTAL:129, OK:3, -> 2.3255813953488373 %
744: Transactions TOTAL:129, OK:2, -> 1.550387596899225 %
464: Transactions TOTAL:129, OK:1, -> 0.7751937984496124 %
1255: Transactions TOTAL:130, OK:1, -> 0.7692307692307693 %
1169: Transactions TOTAL:129, OK:0, -> 0.0 %
1364: Transactions TOTAL:130, OK:2, -> 1.5384615384615385 %
1106: Transactions TOTAL:129, OK:1, -> 0.7751937984496124 %
1095: Transactions TOTAL:129, OK:1, -> 0.7751937984496124 %
702: Transactions TOTAL:129, OK:2, -> 1.550387596899225 %
929: Transactions TOTAL:129, OK:1, -> 0.7751937984496124 %
1222: Transactions TOTAL:129, OK:1, -> 0.7751937984496124 %
1429: Transactions TOTAL:129, OK:1, -> 0.7751937984496124 %
1057: Transactions TOTAL:129, OK:4, -> 3.10077519379845 %
1167: Transactions TOTAL:129, OK:0, -> 0.0 %
1033: Transactions TOTAL:129, OK:1, -> 0.7751937984496124 %
1214: Transactions TOTAL:129, OK:4, -> 3.10077519379845 %
1393: Transactions TOTAL:130, OK:0, -> 0.0 %
1123: Transactions TOTAL:129, OK:1, -> 0.7751937984496124 %
1175: Transactions TOTAL:129, OK:0, -> 0.0 %
1179: Transactions TOTAL:129, OK:0, -> 0.0 %
1053: Transactions TOTAL:129, OK:0, -> 0.0 %
1301: Transactions TOTAL:130, OK:1, -> 0.7692307692307693 %
1118: Transactions TOTAL:129, OK:0, -> 0.0 %
1314: Transactions TOTAL:129, OK:3, -> 2.3255813953488373 %
Stopped
```

Figure 2: Success with 1500 Clients. Some clients reach 0% success rate.

2.1.2 Variation of the entries

In this experiment the number of entries has been increased. This increase in the number of entries is compared with the success rate of the system. The next figure shows the change suffered.

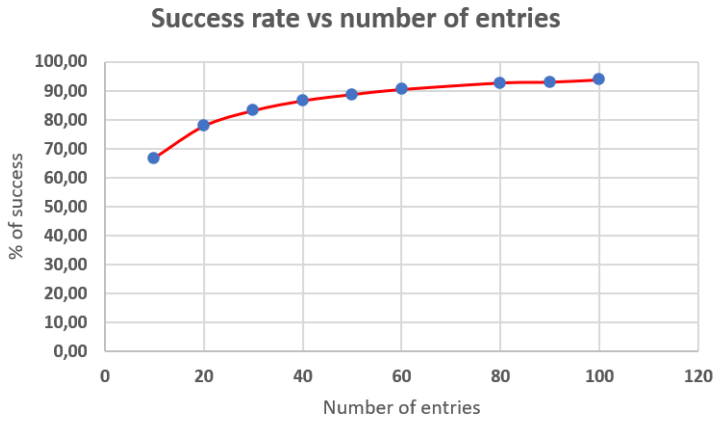

Figure 3: Success rate in % vs number of entries Graph

Figure 1 shows that the increase in the number of entries makes the success rate go higher. This graph only shows the increase from 10 to 100 entries in the shop but the next figure demonstrates the bigger the amount of entries, the closer to 100% success the system is.

```
16> opty:start(3,800,3,2,2).
Starting: 3 CLIENTS, 800 ENTRIES, 3 RDxTR, 2 WRxTR, DURATION 2 s
Stopping...
1: Transactions TOTAL:45427, OK:45088, -> 99.25374777114932 %
2: Transactions TOTAL:45406, OK:45057, -> 99.23137911289257 %
3: Transactions TOTAL:45453, OK:45100, -> 99.22337359470222 %
Stopped
ok
```

Figure 4: Success with 800 entries.

With 800 entries we can get up to 99% of success. This happens because as the entry pool is increased and the number of clients remains the same (3 clients), the requests done by the clients have less chance to be for the same entry and thus, the number of successes increases. For this to happen, is very important that the entry selection by the clients is done randomly and with a uniform distribution. This case would be the "inverse" of increasing the number of clients while maintaining the number of entries done in the previous experiment.

2.1.3 Variation on the read/write operations per transaction

In this experiment the number of the reads has been increased in order to see the impact of the success transactions (percentage OKs). The following plot depicts the impact talked about.

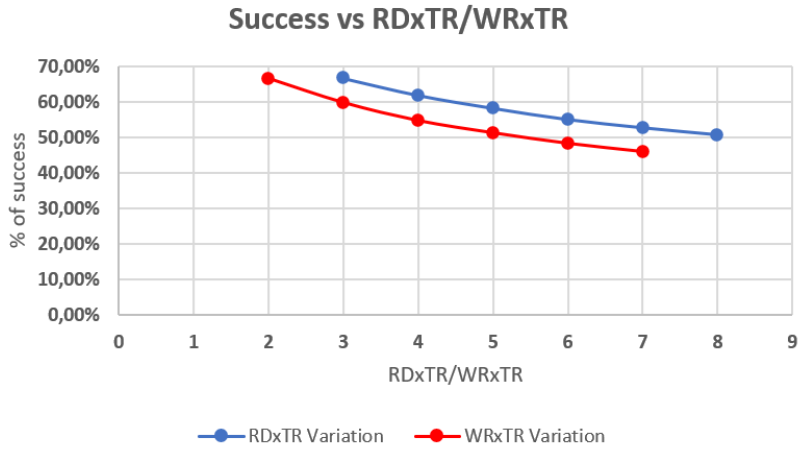

Figure 5: Variation on the read/write operations per transaction

The plot shows that if the number of the reads increase there is less successful transaction ratio. This happens because the reads have more chance to conflict with the write operations. The same happens when the write operations are increased. Thus, the best scenario would be if there was only read operation and no write operations or the opposite.

The plot also shows a difference between increasing reads and increasing writes, this happens because we started the test from different points. For the writes it was 2 and for the reads it was 3.

2.1.4 Variable ratio of reads and writes

There is a total number of operations, reads and writes, and the ratio of these reads and writes in the amount of total operations is what has been varied in this experiment. The following plot represents the variation of the ratio between the reads and writes. It is important to remark that the sum of the reads and writes has to be the equal to the total (100%), e.g, if reads is 10%, writes will be 100-10 = 90%.

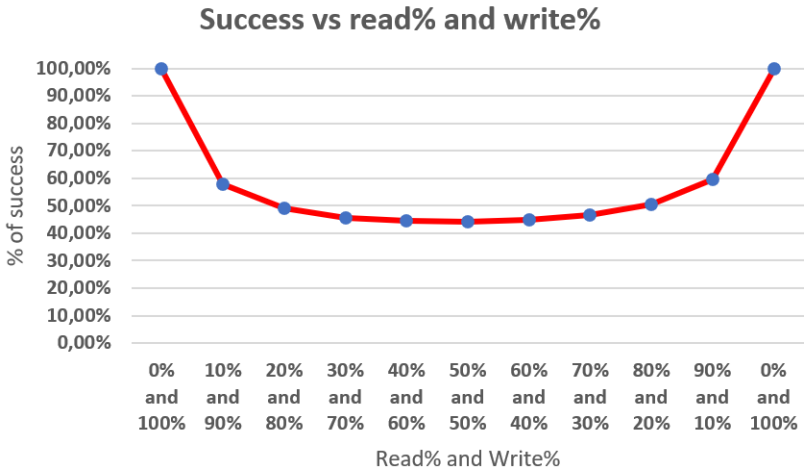

Figure 6: Variable ratio of reads and writes

The plot shows that if there's only one type of operation, Reads or Writes, in the transactions, those will be completed successfully. That happens because in the validator, the reads and writes are compared to know if the transaction is validated, so there's no conflict in the data that is used.

The plot also shows that the worst successful transaction scenario is when the total operations is divided equally with reads and writes.

2.1.5 Different percentage of accessed entries with respect to the total number of entries

Now, the objective of this experiment is to limit the amount of accessed entries over the total entries, by a percentage parameter send in the call of the function, to see how the system reacts (OK success ratio).

Firstly, the code of the "opty.erl" has been changed, specifically the amount of the paramaters that received the `start()` and the `startClients()` functions. Now these functions received one more parameter, the percentage of the entries accessed in the total of the entries.

```
1  -module(opty).
2  -export([start/6, stop/1]).
3
4  start(Clients, Entries, Reads, Writes, Time, Percentage) ->
5      register(s, server:start(Entries)),
6      L = startClients(Clients, [], Entries, Reads, Writes, Percentage),
7      io:format("Starting: ~w CLIENTS, ~w ENTRIES, ~w RDxTR, ~w WRxTR, DURATION ~w s~n",
8            [Clients, Entries, Reads, Writes, Time]),
9      timer:sleep(Time*1000),
10     stop(L).
```

```
1  startClients(0, L, _, _, _, _) -> L;
2  startClients(Clients, L, Entries, Reads, Writes, Percentage) ->
3      Pid = client:start(Clients, Entries, Reads, Writes, s, Percentage),
4      startClients(Clients-1, [Pid|L], Entries, Reads, Writes, Percentage).
```

Finally, the code of the "client.erl" file has been changed to calculate the number of the accessed entries by the percentage parameter. The `start()` function makes a selection of the Accessed entries randomly. The `do_read` and `do_write` functions select an entry of the list of entries by an index.

```
1  -module(client).
2  -export([start/6]).
3
4  start(ClientID, Entries, Reads, Writes, Server, Percentage) ->
5      NumberRandomEntries = round(Entries * Percentage),
6    ListIntegers = lists:seq(1, Entries),
7    RandomSelectEntries = lists:sublist([X || {_ , X} <- lists:sort([{rand:uniform(), E} ||
         E <- ListIntegers])], NumberRandomEntries),
8    spawn(fun() -> open(ClientID, RandomSelectEntries, Reads, Writes, Server, 0, 0) end).
```

```
1  do_read(Entries, Handler) ->
2      Ref = make_ref(),
3    Index = rand:uniform(length(Entries)),
4      Num = lists:nth(Index, Entries),
5      Handler ! {read, Ref, Num},
6      receive
7          {value, Ref, Value} -> Value
8      end.
9
10 do_write(Entries, Handler, Value) ->
11     Index = rand:uniform(length(Entries)),
12     Num = lists:nth(Index, Entries),
13     Handler ! {write, Num, Value}.
```

In the following plot we can see the impact of the accessed entries, by the percentage parameter, in opty. The percentages for the experiment have been 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% and 100%.

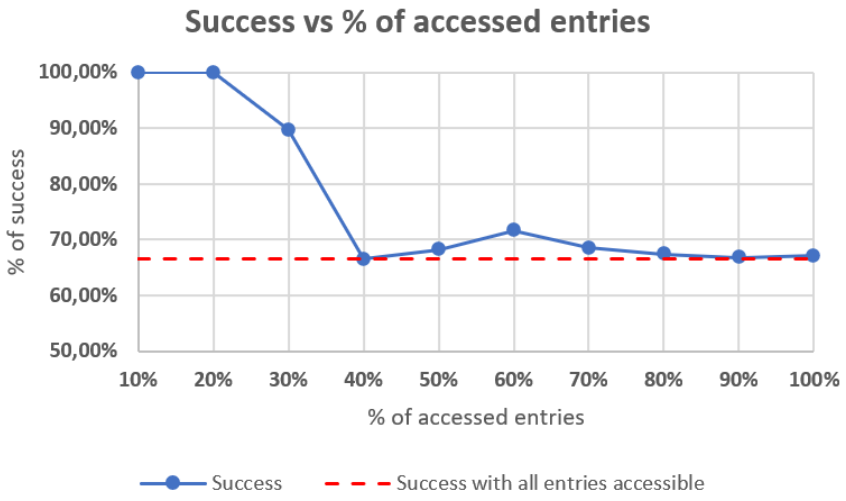

Figure 7: Success vs percentage of accessed entries

The plot depicts the success depending on the amount of accessed entries by increasing the percentage, the result is closer to the experiment where all the entries were accessible.

2.2 Distributed execution

In this experiment Opty is launched in a distributed manner. This means that the clients and the server will be executed in different Erlang instances. The server will run on a remote instance while the clients will run in the same instance as the Opty program.

The commands that are used in the console (two different consoles) to create this two different instances are the following:

- **`erl -name opty@127.0.0.1`**: Instance where Opty and the Clients run.

- **`erl -name opty-srv@127.0.0.1`**: Instance where the server runs.

The "opty.erl" file has been changed in order to implement this experiment. Now, the `start()`, `stop()` and the `startClients()` functions receive one more parameter, the server node ('opty-srv@127.0.0.1'), also there are two new functions that were implemented. The first one is `startServer()`, used to start in the remote server the entries. The last one is `startDistributed()` and it is used to call the `startServer()` function, it spawns the server in the Server node, and the `start()` function.

```
1
2  -module(opty).
3  -export([start/6, stop/2, startDistributed/6, startServer/1]).
4
5  %% Clients: Number of concurrent clients in the system
6  %% Entries: Number of entries in the store
7  %% Reads: Number of read operations per transaction
8  %% Writes: Number of write operations per transaction
9  %% Time: Duration of the experiment (in secs)
10
11 startDistributed(Clients, Entries, Reads, Writes, Time, Server) ->
12   spawn(Server, opty, startServer, [Entries]),
13   timer:sleep(1000),
14   io:format("Starting clients ~n"),
15   start(Clients, Entries, Reads, Writes, Time, Server).
16
17 start(Clients, Entries, Reads, Writes, Time, Server) ->
18     L = startClients(Clients, [], Entries, Reads, Writes, Server),
19     io:format("Starting: ~w CLIENTS, ~w ENTRIES, ~w RDxTR, ~w WRxTR, DURATION ~w s~n",
20           [Clients, Entries, Reads, Writes, Time]),
21     timer:sleep(Time*1000),
22     stop(L, Server).
23
24 startServer(Entries)->
25   register(s, server:start(Entries)).
26
27 stop(L, Server) ->
28     io:format("Stopping...~n"),
29     stopClients(L),
30   waitClients(L),
31     {s, Server} ! stop,
32     io:format("Stopped~n").
33
34 startClients(0, L, _, _, _, _) -> L;
35 startClients(Clients, L, Entries, Reads, Writes, Server) ->
36     Pid = client:start(Clients, Entries, Reads, Writes, {s, Server}),
37     startClients(Clients-1, [Pid|L], Entries, Reads, Writes, Server).
38
39 stopClients([]) ->
40     ok;
41 stopClients([Pid|L]) ->
42     Pid ! {stop, self()},
43     stopClients(L).
44
45 waitClients([]) ->
46     ok;
47 waitClients(L) ->
48     receive
49         {done, Pid} ->
50             waitClients(lists:delete(Pid, L))
51     end.
```

The next figure shows that Opty successfully executes remotely without errors. It also shows that the success rate does not change from the normal execution to the remote execution.

Figure 8: Remote execution of Opty. At the right, server instance. At the left Opty and Clients instance.

2.3 Forward Validation

```
1  -module(entry).
2  -export([new/1]).
3
4  new(Value) ->
5      spawn_link(fun() -> init(Value) end).
6
7  init(Value) ->
8      entry(Value, []).
9
10 entry(Value, Active) ->
11     receive
12         {read, Ref, From, Client} ->
13             From ! {Ref, self(), Value},
14       case lists:member(Client, Active) of
15         true ->
16           entry(Value, Active);
17         false ->
18           entry(Value, [Client|Active])
19       end;
20         {write, New} ->
21             entry(New , Active);
22         {check, Ref, From, Client} ->
23       NewList = lists:delete(Client, Active),
24             if
25                  length(NewList) == 0 ->
26                     From ! {Ref, ok};
27                 true ->
28                     From ! {Ref, abort}
29             end,
30             entry(Value, Active);
31     {deactivate, From, Client} ->
32       From ! ok,
33       NewList = lists:delete(Client, Active),
34       entry(Value, NewList);
35
36         stop ->
37             ok
38     end.
```

```
-module(handler).
-export([start/3]).

start(Client, Validator, Store) ->
    spawn_link(fun() -> init(Client, Validator, Store) end).

init(Client, Validator, Store) ->
    handler(Client, Validator, Store, [], []).

handler(Client, Validator, Store, Reads, Writes) ->
    receive
        {read, Ref, N} ->
            case lists:keyfind(N, 1, Writes) of
                {N, _, Value} ->
                    Client ! {value, Ref, Value},
                    handler(Client, Validator, Store, Reads, Writes);
                false ->
                    store:lookup(N, Store) ! {read, Ref, self(), Client},
                    handler(Client, Validator, Store, Reads, Writes)
            end;
    {write, N, Value} ->
            Added = lists:keystore(N, 1, Writes, {N, store:lookup(N, Store), Value}),
            handler(Client, Validator, Store, Reads, Added);
        {Ref, Entry, Value} ->
            Client ! {value, Ref, Value},
            handler(Client, Validator, Store, [{Entry}|Reads], Writes);
        {commit, Ref} ->
            Validator ! {validate, Ref, Reads, Writes, Client};
        abort ->
            ok
    end.
```

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

start() ->
    spawn_link(fun() -> init() end).

init()->
    validator().

validator() ->
    receive
        {validate, Ref, Reads, Writes, Client} -> %%Write = pending write operations
            (store), Reads (List of reads performed), Client = ID process of client to
            reply
            Tag = make_ref(),
            send_write_checks(Writes, Tag, Client),
            case check_writes(length(Writes), Tag) of
                ok ->
                    update(Writes),
          deactivate(Reads, Client),
                    Client ! {Ref, ok};
                abort ->
          deactivate(Reads, Client),
          Client ! {Ref, abort}
            end,
            validator();
        stop ->
            ok;
        _Old ->
            validator()
    end.

update(Writes) ->
    lists:foreach(fun({_, Entry, Value}) ->
```

```
          Entry ! {write, Value}
                  end,
                  Writes).

send_write_checks(Writes, Tag, Client) ->
  Self = self(),
  lists:foreach(fun({_, Entry, _}) ->
        Entry ! {check, Tag, Self, Client}
        end,
        Writes).

check_writes(0, _) ->
    ok;
check_writes(N, Tag) ->
    receive
        {Tag, ok} ->
            check_writes(N-1, Tag);
        {Tag, abort} ->
            abort
    end.

deactivate(Reads, Client) ->
  Self = self(),
    lists:foreach(fun({Entry}) ->
          Entry ! {deactivate, Self, Client}
                  end,
                  Reads),
  check_deactivate(length(Reads)).

check_deactivate(N) ->
  if
    N == 0 ->
      ok;
    true ->
      receive
        ok ->
          check_deactivate(N-1)
      end
  end.
```

```
PS C:\Users\Imanol\OneDrive\Escritorio\Master\Q1\CPDS\Labs\DistributedSystems\Opty\CPDS-opty\Forward-validation> erl
Eshell V12.1  (abort with ^G)
1> opty:start(3,10,3,2,3).
Starting: 3 CLIENTS, 10 ENTRIES, 3 RDxTR, 2 WRxTR, DURATION 3 s
Stopping...
2: Transactions TOTAL:84419, OK:55975, -> 66.30616330446938 %
3: Transactions TOTAL:84310, OK:56205, -> 66.66468983513225 %
1: Transactions TOTAL:84311, OK:56104, -> 66.5441045652406 %
Stopped
ok
2>
```

Figure 9: Results of executing the forward validation version with the same parameters as in the vanilla version (3,10,3,2,3)

3 Open questions

The questions that the seminar guide proposes are answered in this section. The questions are the following ones:

Q1) What is the impact of each of these parameters on the success rate?

As it has been seen and explained in the experiment section, varying the parameters of the system changes the amount of success that the requests have.

First of all, it has been explained that increasing the number of clients decreases the success rate because more clients try to access the same number of entries and thus is more probable to end up having a conflict. The inverse case happens if the entries are the ones increased. In this case the success rate is better due to the inverse phenomenon. More entries with the same number of clients means less chance of the clients making requests on the same entries.

Secondly, increases on the number of write or read requests per transaction also worsen the success rate and if the rate of read and write requests is close to be the same (i.e. 50% Read request per transaction and 50% write requests per transaction), this success rate gets close to the worst case.

Finally, in the fifth experiment (VI in the seminar guide), the results show that if the entries are accessed randomly with different percentages the success rate can differ between clients. Also the success rate is worse (but fits the results in the previous experiments) when all entries are accessible.

Q2) Is the success rate the same for the different clients?

In most of the cases the success rate is the same or similar from client to client. The experiments where this did not happen where the ones where failure was forced as the first experiment where the number of clients was increased until a point where some clients had no services and others had low success rates but where served. Other case where the clients had different success rates was the fifth experiment (sixth in the seminar guide).

Q3) If we run this in a distributed Erlang network, where is the handler running?

As it is shown in section 2.2. the Transaction Handler runs where the Client runs. This is because in this software, the creation of a Client spawns a Transaction Handler that is completely bound with the client. This means that when the Client dies, the Transaction Handler will also die with the Client.

4 Personal opinion

Implementing Opty we learned how Optimistic Concurrency Control works and how it reacts to changes in its principal parameters. We found easy to understand how the program works but we had problems with the language used. As the last seminar we find that maybe wit other programming language would have been easier to understand and also to program the system.

5 Appendix: Tables

Clients Variation				
	OK % Client 1	OK % Client 2	OK % Client 3	OK % mean
3	66,8499%	66,5381%	66,4925%	66,6268%
5	52,7632%	52,2089%	53,0514%	52,6745%
10	34,2268%	34,7942%	34,6506%	34,5572%
15	25,9634%	25,8054%	24,9279%	25,5656%
20	20,8465%	22,0299%	21,0707%	21,3157%
25	18,9591%	18,0051%	18,0556%	18,3399%

Table 1: Success rate in % vs number of clients Table

Success rate vs number of entries				
	SR1	SR2	SR3	SR Mean
10	66,78	66,7	66,77	66,75
20	77,72	77,88	77,72	77,77
30	82,99	83,08	83,33	83,13
40	86,46	86,84	86,35	86,55
50	88,54	88,76	88,83	88,71
60	90,55	90,42	90,37	90,45
80	92,47	92,84	92,8	92,70
90	92,94	93,12	92,94	93,00
100	93,74	93,71	93,9	93,78

Table 2: Success rate in % vs number of entries Table

RDxTR Variation				
	OK % Client 1	OK % Client 2	OK % Client 3	OK % mean
3	66,8499%	66,5381%	66,4925%	66,6268%
4	61,1887%	62,0642%	62,0272%	61,7600%
5	58,3425%	57,8633%	58,0583%	58,0880%
6	54,3876%	55,1073%	55,2156%	54,9035%
7	52,2997%	52,6697%	52,7306%	52,5667%
8	50,2162%	50,7591%	50,7427%	50,5727%

Table 3: Success rate in % vs number of read requests per transaction (RDxTR) Table

WRxTR Variation				
	OK % Client 1	OK % Client 2	OK % Client 3	OK % mean
2	66,5122%	66,5664%	66,5340%	66,54%
3	60,1503%	59,4998%	59,4942%	59,71%
4	54,6991%	54,6016%	54,5329%	54,61%
5	51,1332%	50,8559%	51,4897%	51,16%
6	47,9144%	47,8754%	48,7290%	48,17%
7	46,4692%	45,4440%	45,6360%	45,85%

Table 4: Success rate in % vs number of write requests per transaction (WRxTR) Table

Read & Write variation				
	OK % Client 1	OK % Client 2	OK % Client 3	OK % mean
0% and 100%	100,0000%	100,0000%	100,0000%	100,0000%
10% and 90%	58,3840%	57,6449%	57,9706%	57,9998%
20% and 80%	49,4524%	49,4076%	48,7012%	49,1871%
30% and 70%	45,6196%	45,4968%	46,1604%	45,7589%
40% and 60%	44,8715%	44,0975%	44,9100%	44,6263%
50% and 50%	44,2718%	44,7690%	43,5308%	44,1905%
60% and 40%	45,4338%	44,9540%	44,5159%	44,9679%
70% and 30%	47,4907%	45,9068%	46,4426%	46,6134%
80% and 20%	50,4066%	50,6775%	50,4520%	50,5120%
90% and 10%	60,0961%	59,1579%	59,0751%	59,4430%
0% and 100%	100,0000%	100,0000%	100,0000%	100,0000%

Table 5: Success rate in % vs % of variation of read and write requests Table

Experiment V					
	OK %	OK %	OK %	OK % mean	Normal OK%
10%	100,0000%	100,0000%	100,0000%	100,0000%	66,6268%
20%	100,0000%	100,0000%	100,0000%	100,0000%	66,6268%
30%	84,3360%	100,0000%	84,6852%	89,6737%	66,6268%
40%	63,9031%	72,0351%	63,6690%	66,5357%	66,6268%
50%	68,7099%	74,4049%	61,5171%	68,2106%	66,6268%
60%	74,9632%	70,2902%	69,7192%	71,6575%	66,6268%
70%	68,5291%	68,6338%	68,5044%	68,5558%	66,6268%
80%	71,1943%	65,4348%	65,5889%	67,4060%	66,6268%
90%	68,1152%	65,9279%	66,2829%	66,7753%	66,6268%
100%	67,4315%	66,9126%	66,8052%	67,0498%	66,6268%

Table 6: Success rate in % when varying the percentage of accessed entries