Seminar Report: OPTY Seminar

Shiva Toutounchiavval, Fernando Morales, Cristián Ramón-Cortés

June 01, 2015

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Contents

1. Introduction 3

2. Work done 4

3. Experiments 5

3.1 Different number of concurrent clients in the system 5

3.2 Different number of entries in the store 6

3.3 Different number of read operations per transaction 8

3.4 Different number of write operations per transaction 10

3.5 Different ratio of read and write operations for a fixed amount of operations per transaction 11

3.6 Different percentage of accessed entries with respect to the total number of entries 13

4. Open questions 16

5. Distributed execution 17

6. Other concurrency control techniques 18

7. Personal Opinion 20

1. Introduction

We implemented a transactional server in *Erlang* with optimistic concurrency control. *Opty* is a concurrency control method for transactional systems such as relational database management systems. In this seminar we want to achieve optimistic concurrency control with backward validation. In short, to run concurrent transactions without locks acquisition, and using data access records to roll back any transaction if it cannot be validated.

2. Work done

We started from a partial implementation of *Opty* in *Erlang*, consisting of different modules corresponding to each element from the transactional context: server, client, handler, validator, etc. At this point, we already noticed how *Erlang* manages data store by means of additional processes that save a desired value as its internal state.

After completing and understanding the code, we started some experiments in order to test scalability and sensibility to particular parameters, such as the number of concurrent clients, entries, read/write operations ratio, etc. We obtained some numerical data regarding these experiments, which is discussed in the following chapters.

Next, we have developed a distributed version of *Opty*, that allows to run the server and the clients in different machines.

Finally, we measured the performance of our implementation with backward validation against an *Erlang* implementation that uses timestamp ordering provided in the course material.

3. Experiments

For each set of experiments we first define a value for each parameter that performs a global configuration with mostly no impact on the accuracy rate. This configuration is called *base* and it's meant to eliminate the effect of all non-studied parameters. This base is maintained in all the experiments and the only parameter varied during the experimentation is the studied one. This fact allows us to assume that the variation on the accuracy rate during a set of experiments can be totally assigned to the effect of the variation of the studied parameter.

3.1 Different number of concurrent clients in the system

In this section the base is defined as follows: 10 entries, 3 read operations per transaction, 2 write operations per transaction and 5seconds as duration time. The only modified parameter is the number of clients. The executed command for this set of experiments is `opty:start(NUMC,10,3,2,5)`. *Figure 1* shows an execution example.

```
4> opty:start(3,10,3,2,5).
Starting: 3 CLIENTS, 10 ENTRIES, 3 RDxTR, 2 WRxTR, DURATION 5 s
Stopping...
3: Transactions TOTAL:68164, OK:43729, -> 64.15263188780001 %
1: Transactions TOTAL:68235, OK:43793, -> 64.1796731882465 %
2: Transactions TOTAL:68129, OK:43777, -> 64.25604368183886 %
Stopped
ok
```

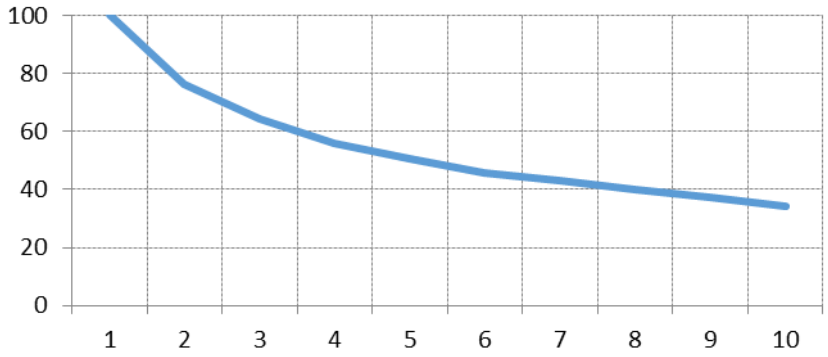
Figure 1: Example execution of the experimentation with the number of clients

The experiments done vary the number of clients from 1 to 10, and the obtained results are shown in *Table 1*.

Number of Clients	1	2	3	4	5	6	7	8	9	10
AverageAccuracy Rate (%)	100%	76.24%	64.25%	56.10%	50.41%	45.57%	42.88%	39.98%	37.31%	34.29%

Table 1: Average Accuracy Rate for different number of clients

Making a simple plot with the obtained values (*Plot 1*), we can demonstrate that, as we expected, the average accuracy rate decreases when the number of clients is increased.



Plot 1: Average Accuracy Rate vs. Number of Clients

3.2 Different number of entries in the store

For this set of experiments the base is defined with the following values: 5 clients, 3 read operations per transaction, 2 write operations per transaction and 5

seconds as duration time. The modified parameter is the number of entries. The executed command for this set of experiments is: `opty:start(5,NUME,3,2,5).` *Figure 2* shows an execution example.

```
18> opty:start(5,5,3,2,5).
Starting: 5 CLIENTS, 5 ENTRIES, 3 RDxTR, 2 WRxTR, DURATION 5 s
Stopping...
1: Transactions TOTAL:44039, OK:18894, -> 42.902881536819635 %
4: Transactions TOTAL:43920, OK:18925, -> 43.08970856102004 %
2: Transactions TOTAL:44082, OK:18942, -> 42.96991969511365 %
3: Transactions TOTAL:44113, OK:19155, -> 43.42257384444495 %
5: Transactions TOTAL:44102, OK:18938, -> 42.94136320348284 %
Stopped
ok
```

Figure 2: Example execution of the experimentation with the number of entries

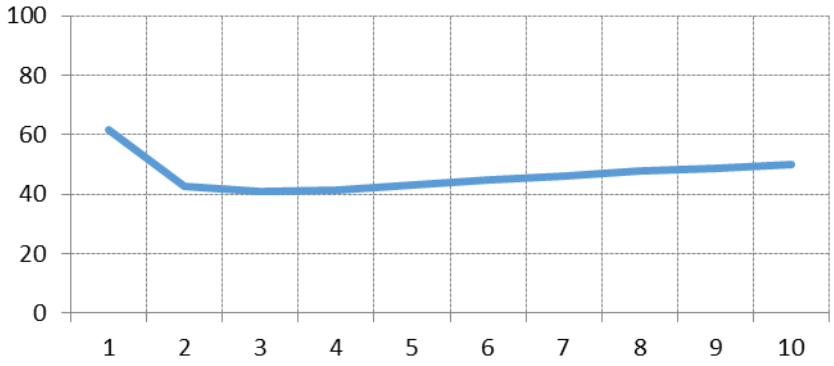
The experiments done vary the number of entries from 1 to 10, and the obtained results are shown in *Table 2*.

Number of Entries	1	2	3	4	5	6	7	8	9	10
Average Accuracy Rate (%)	61.83%	42.72%	41.7%	41.51%	42.94%	44.63%	46.14	47.60%	48.58%	50.04%

Table 2: Average Accuracy Rate for different number of entries

Plotting the obtained values, as shown in *Plot 2*, we can see how the number of abortions decreases as the number of entries increases. These results are coherent, since a higher number of available entries implies a lower chance of collision between

clients, considering that the requested entries of a client are chosen uniformly and that each client choses a fixed number of entries in a transaction.



Plot 2: Average Accuracy Rate vs. Number of Entries

3.3 Different number of read operations per transaction

To study the effect of the number of read operations per transaction the following values are defined as base: 4 clients, 3 entries, 6 write operations per transaction and 5 seconds as duration time. The only modified parameter is the number of read operations per transaction. The executed command for this set of experiments is: `opty:start(4,3,RxT,6,5)`. *Figure 3* shows an execution example.

```
24> opty:start(4,3,1,6,5).
Starting: 4 CLIENTS, 3 ENTRIES, 1 RDxTR, 6 WRxTR, DURATION 5 s
Stopping...
1: Transactions TOTAL:73093, OK:35513, -> 48.58604791156472 %
3: Transactions TOTAL:73135, OK:35273, -> 48.22998564298899 %
2: Transactions TOTAL:73031, OK:35276, -> 48.302775533677476 %
4: Transactions TOTAL:73082, OK:35554, -> 48.649462247885936 %
Stopped
ok
```

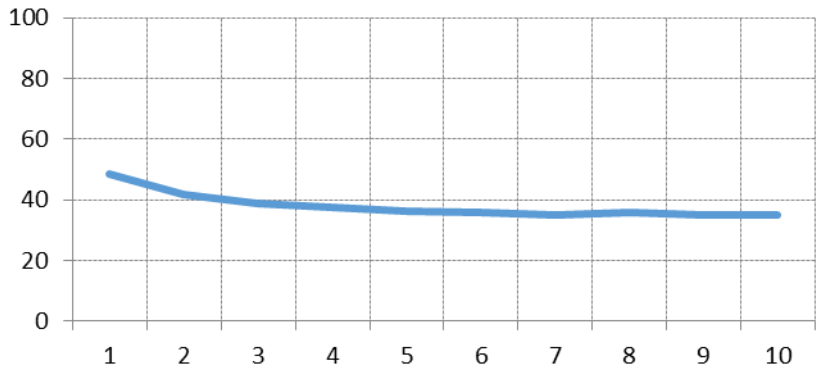
Figure 3: Example execution of the experimentation with the number of read operations per transaction

The experiments done vary the number of read operations per transaction from 1 to 10, and the obtained results are shown in *Table 3*.

Number of Read operations	1	2	3	4	5	6	7	8	9	10
Average Accuracy Rate (%)	48.64%	41.81%	38.86%	37.49%	36.20%	35.90%	35.10%	35.92%	34.96%	34.14%

Table 3: Average Accuracy Rate for different number of read operations per transaction

Plotting the obtained values (see *Plot 3*), we can see how the number of aborted transactions increases as the number of read operations increases. A higher number of read operations imply more entries to be read. Assuming a fixed number of write operations, this will lead to a higher chance of reading an entry already written by another client, thus aborting the transaction in the validation phase.



Plot 3: Average Accuracy Rate vs. number of read operations per transaction

3.4 Different number of write operations per transaction

In this section the base is defined as: 4 clients, 3 entries, 6 read operations per transaction and 5 seconds as duration time. The only parameter that is modified is the number of write operations per transaction. The command executed during this set of experiments is `opty:start(4,3,6,WxT,5)`. *Figure 4* shows an execution example.

```
26> opty:start(4,3,6,1,5).
Starting: 4 CLIENTS, 3 ENTRIES, 6 RDxTR, 1 WRxTR, DURATION 5 s
Stopping...
3: Transactions TOTAL:36337, OK:20353, -> 56.0117786278449 %
2: Transactions TOTAL:36412, OK:20335, -> 55.84697352521147 %
4: Transactions TOTAL:36300, OK:20220, -> 55.70247933884298 %
1: Transactions TOTAL:36318, OK:20287, -> 55.85935348862823 %
Stopped
ok
```

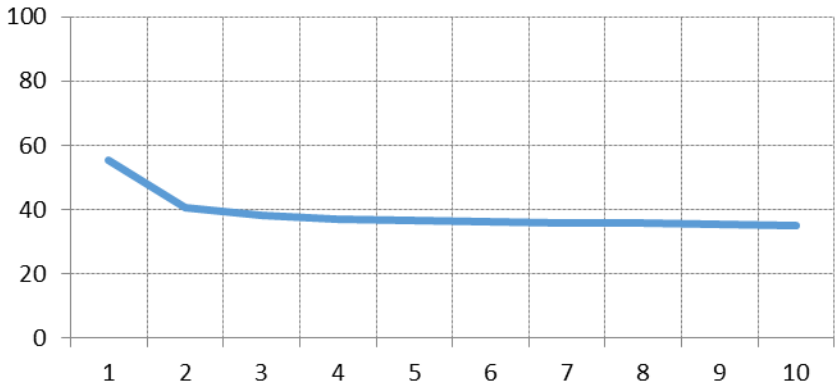
Figure 4: Example execution of the experimentation with the number of write operations per transaction

The experiments done vary the number of write operations per transaction from 1 to 10, and the obtained results are shown in *Table 4*.

Number of Write operations	1	2	3	4	5	6	7	8	9	10
Average Accuracy Rate (%)	55.85%	40.64%	38.28%	37.01%	36.79%	36.23%	35.72%	35.71%	35.51%	35.19%

Table 4: Average Accuracy Rate for different number of write operations per transaction

Plotting the obtained values (see *Plot 4*), we can see a similar behaviour than in the previous experiment. A higher number of write operations will produce a higher number of invalid read operations, thus aborting more transactions in the validation phase.



Plot 4: Average Accuracy Rate vs. number of write operations per transaction

3.5 Different ratio of read and write operations for a fixed amount of operations per transaction

For this set of experiments the base is defined with 4 clients, 3 entries and 5 seconds as duration time. We fix 11 operations per transaction and we vary the number of read and write operations accordingly. The executed command during this experiments is: `opty:start(4,3,RxT,WxT,5)`. *Figure 5* shows an execution example.

```
27> opty:start(4,3,1,10,5).
Starting: 4 CLIENTS, 3 ENTRIES, 1 RDxTR, 10 WRxTR, DURATION 5 s
Stopping...
1: Transactions TOTAL:65098, OK:30447, -> 46.77102215121816 %
4: Transactions TOTAL:65047, OK:30191, -> 46.414131320429846 %
3: Transactions TOTAL:64838, OK:29980, -> 46.23831703630587 %
2: Transactions TOTAL:64941, OK:29973, -> 46.15420150598235 %
Stopped
ok
```

Figure 5: Example execution of the experimentation with the number of read and write operations per transaction with a fixed amount of operations per transaction

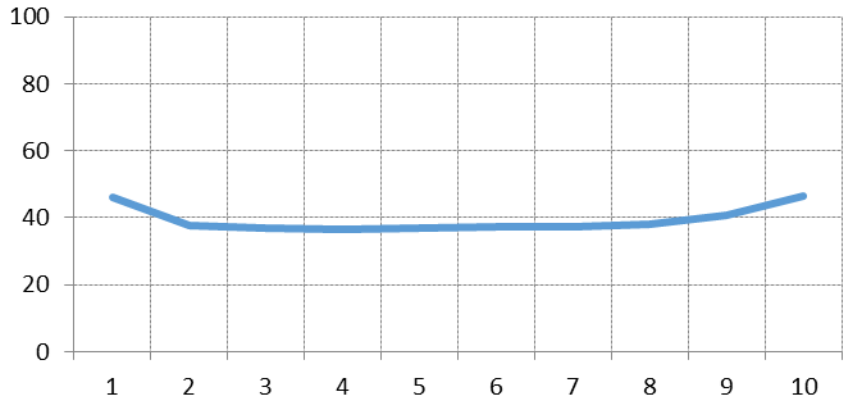
The set of experiments varies the number of read operations per transaction from 1 to 10 and the number of write operations per transaction from 10 to 1. The obtained results are shown in *Table 5*.

Number of Readand Writeoperations	1-10	2-9	3-8	4-7	5-6	6-5	7-4	8-3	9-2	10-1
Average Accuracy Rate (%)	46.15%	37.65%	36.72%	36.43%	36.99%	37.23%	37.12%	38.17%	40.68%	46.36%

Table 5: Average Accuracy Rate for different number of read and write operations per transaction with a fixed amount of operations per transaction

Plotting the obtained values (see Plot 5), we can see some quadratic relationship between the accuracy rate and the two parameters. This can be explained as follows: the improvement produced by decreasing the number of write operations is overcome by the deterioration produced by increasing the number of readings. This

achieves worst performance with parameters (5, 6). From that point on, it starts to improve again, since the improvement produced by decreasing the number of write operations will be higher than its reading counterpart.



Plot 5: Average Accuracy Rate vs number of read operations per transaction

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

For this set of experiments we use a new implementation of O*pty* (located under the *src/percentage*) that allows us to define the number of entries accessible from each client. This parameter needs to be between 0 and the total number of entries, having the same behavior as the previous implementation when the number of entries accessible from each client is the same as the total number of entries. For each client a subset of the entries is calculated randomly and he can only access the entries inside this subset.

We define as base: 5 clients, 4 entries, 3 read operations per transaction, 2 write operations per transaction and 10 seconds as duration time. The size of the entries'

subset of each client is varied during the experimentation. The command executed is `opty:start(5,15,3,2,size,10)`. *Figure 6* shows an execution example.

```
2> opty:start(5,15,3,2,1,10).
Client <0.36.0> has entries [7]
Client <0.37.0> has entries [11]
Client <0.38.0> has entries [15]
Client <0.39.0> has entries [8]
Client <0.40.0> has entries [5]
Starting: 5 CLIENTS, 15 ENTRIES, 3 RDxTR, 2 WRxTR, 1 SUBSETSIZE, DURATION 10 s
Stopping...
1: Transactions TOTAL:121192, OK:121192, -> 100.0 %
3: Transactions TOTAL:121047, OK:121047, -> 100.0 %
5: Transactions TOTAL:121053, OK:121053, -> 100.0 %
2: Transactions TOTAL:121304, OK:121304, -> 100.0 %
4: Transactions TOTAL:120698, OK:120698, -> 100.0 %
Stopped
ok
```

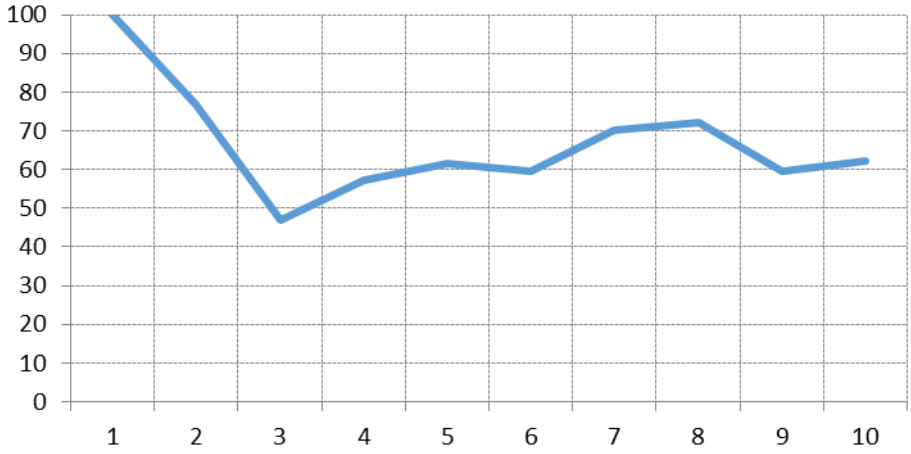
Figure 6: Example execution of the experimentation with the percentage of accessed entries

The set of experiments varies the number of accessible entries per client from 1 to 10. Since the number of total entries is left constant and equal to 15, this experimentation increases the percentage of accessed entries per client from 6.6% to 66.6%. The obtained results are shown in *Table 6*.

Number of entries per subset	1	2	3	4	5	6	7	8	9	10
Average Accuracy Rate (%)	100 %	76.82%	47.07%	57.07%	61.48%	59.64%	70.31%	72.14%	59.68%	64.95%

Table 6: Average Accuracy Rate for different percentage of accessed entries

Plotting the obtained values (see Plot 6), we cannot see a strong dependence between the number of entries per subset and the accuracy rate. This can be explained by the randomness introduced in this experiment: the probability that a read/write operation conflicts between two or more clients should be directly proportional to the subset size. However, the random choice of these subsets from instance to instance breaks the relation between the different instances shown here, in contrast with the theoretical explanation.



Plot 6: Average Accuracy Rate vs percentage of accessed entries

4. Open questions

The success rate is affected by many parameters:

In experiment 3.1, by increasing the number of clients, the success rate decreases. As a consequence the number of possible conflict and the number of aborted transactions will increase too.

In experiment 3.2,it is clear that as we increase the number of entries, the success rate also increases. In this case the small number of entries will increase the possible conflicts because similar subsets of entries are shared between clients and it arise more conflicts in transactions. Then different subsets of entries will lead to less confecting or aborting transactions.

In experiment 3.3 and 3.4, with more writes and reads operations more conflicts appear and the success rate decreases.

In experiment 3.5 by increasing the number of reads to the number of writes or increasing the number of write to the number of reads, the success rate decreases. For experiment v the conflict will increase by increasing the size of the client's entry subset.

In experiment 3.6, by increasing the number of entries in subset, success rate decreases.

5. Distributed execution

For this section we use a new implementation of the *Opty* module located under *src/split*folder. This implementation allows us to split the server and the clients providing a start function for the server and a start function for the clients. For its execution two nodes are created:

1) A server node: optyserver@localhostwith `opty:start_server(10).`
2) A clients node: clients@localhost with `opty:start_clients(5, 4, 1, 1, 2,5, optyserver@server,3).`

When running a distributed *Erlang* network, the handler runs in the clients node because all the operations done by the clients have to be performed through the handler. This choice could be arbitrary, but locating all the handlers in the server would have some negative effects:

1) Increasing the computational effort of the central server.
2) Increasing the latency between the client and its handler.

6. Other concurrency control techniques

To compare the performance results of the transaction server when using concurrency control with backward validation with respect to timestamp ordering we use the timey implementation provided in the wiki course. For the experimentation with set as base: 10 entries, 3 read operations per transaction, 2 write operations per transaction and 5 seconds as duration time. The number of clients is varied along the experiments. The command executed is `timey:start(1,10,3,2,5)` and *Figure 7* shows an execution example.

```
16> timey:start(8,1,3,2,5).
Starting: 8 CLIENTS, 1 ENTRIES, 3 RDxTR, 2 WRxTR, DURATION 5 s
Stopping...
6: Transactions TOTAL:89259, OK:1, -> 0.0011203352042931246 %
1: Transactions TOTAL:89074, OK:1, -> 0.0011226620562678222 %
5: Transactions TOTAL:89100, OK:1, -> 0.001122334455667789 %
2: Transactions TOTAL:89111, OK:1, -> 0.00112219591296248 5 %
7: Transactions TOTAL:88978, OK:1, -> 0.0011238733169997078 %
8: Transactions TOTAL:88911, OK:1, -> 0.0011247202258438214 %
4: Transactions TOTAL:89265, OK:1, -> 0.0011202599002968 69 %
3: Transactions TOTAL:89036, OK:1, -> 0.0011231412013118 29 %
Stopped
ok
```

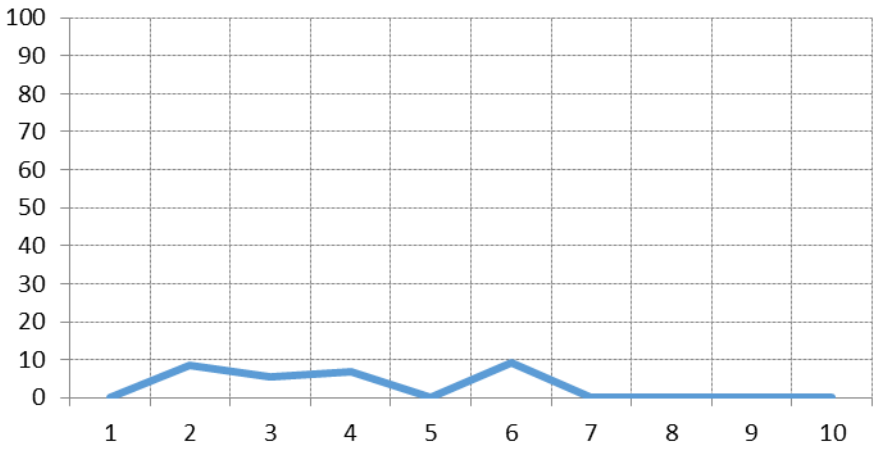
Figure 7: Example execution of timey implementation

The set of experiments varies the number of clients from 1 to 10 providing the results shown in *Table 7*.

Number of Clients	1	2	3	4	5	6	7	8	9	10
Average Accuracy Rate (%)	0.001%	8.41%	5.34%	6.83%	0.001%	9.22%	0%	0.001%	0.001%	0.002%

Table 7: Average Accuracy Rate of timey implementation for different number of clients

Plotting the obtained values (see *Plot 7*), we can see, a worse performance than in backward validation. In this particular case the timestamp ordering does not provide an admissible way to run our transactional server.



Plot 7: Average Accuracy Rate of timey implementation vs. number of clients

7. Personal Opinion

Working with backward validation in Optimistic Concurrency Control algorithm is the most important aspect of this seminar. We have discovered the central elements of a transactional server, along with the particular functionality of each of them.

We have worked with *Erlang* in a very particular way: using the capability of processes to act as data containers. This has been really useful for future work with *Erlang*.

We are very satisfied with this seminar, since it has covered more or less everything that we could have asked for while we were working on it, and has offered us an overview of database validation in distributed environments.