Seminar Report: Opty

Cegarra Dueñas, Gerard
Jabeen, Sana
Mosquera Dopico, Sergio

January 14, 2019

1 Introduction

Opty is a concurrency control method for transactional systems such as relational database management systems. What we want to achieve, is an optimistic concurrency control system with backward (and forward as an experiment) validation. As a resume, we want that our system is able to run concurrent transactions without getting lock, and also to allow rollbacks if any transaction cannot be validated.

2 Work done

The work consisted on developing several experiments to ensure we have understand correctly the algorithm (and also the code). This is developed fully in Erlang, taking advantage of the material provided by the teacher. The code is sent along with this report and there is a general code (the original one) and several folders with changes in the code that represent the different experiments.

3 Experiments

The results shown in the following tables represent the following information:

- Number of elements of the parameter whose value will be changed.
- Total number of transactions (round the summation of the transactions per client in subsections 3.1, 3.2 and 3.3, and average among clients in the other ones).
- Average number of OK values per client.
- Average percentage of OK transactions from the total.

This is done in that way because showing the total number of transactions gives a better perspective of how the data changes according to the value of the parameters. On the other hand, the change for each client in the OK messages per client and the average % is enough representative by itself so it is not necessary to compute the average of the summation per each value.

3.1 Different number of concurrent clients in the system

The following table represents the results according to giving different values for the different number of concurrent clients. There are a couple of things to be remarked:

- The total number of transactions increases up to a specific limit (around 300000) while we increase the number of processes, but then starts to decrease when the number of processes is too big.
- The values in the table represent an approximation of the real results i.e. the total number of transactions is rounded to the closer value ending in 0, the number of OKs represent an average of the real values obtained from every process.
- The experiments will be carried out using the following values:

  ```
  opty:start(X,10,3,2,3).
  ```

  Where X represents the number of concurrent clients.

# Clients	# Transactions	# OK	Avg %
3	220000	49000	66.67
5	260000	27250	52.75
10	305000	10750	35
20	300000	3100	21
50	300000	650	10.5
100	250000	165	6.25

Table 1: Different number of clients

As we can see with the results from the table, as we increase the number of concurrent clients, we will obtain worse results (with respect to the Avg %), because the number of OK values will be more or less the same for every client.

3.2 Different number of entries in the store

The following table represents the results according to giving different values for the number of entries in the store. There are a couple of things to be remarked:

- As we increase the number of entries in the store, the total number of transactions decreases as well as the number of OKs per process increases, such that the avg percentage also increases.
- The experiments will be carried out using the following values:

  ```
  opty:start(3,X,3,2,3).
  ```

  Where X represents the number of entries in the store.

# Entries	# Transactions	# OK	Avg %
3	250000	44000	52
5	237000	44700	56.3
10	220000	49000	66.6
20	210000	55000	77.75
50	204000	60000	88.9
100	192000	61000	93.9

Table 2: Different number of entries in the store

3.3 Different number of read operations per transaction

The following table represents the results according to giving different values for the number of read operations per transaction. There are a couple of things to be remarked:

- As we increase the number of read operations per transaction, the total number of transactions decreases as well as the number of OKs per process also decreases faster. This leads to a decrement of the average percentage.
- The experiments will be carried out using the following values:

  ```
  opty:start(3,10,X,2,3).
  ```

  Where X represents the number of read operations per transaction.

# RDxTR	# Transactions	# OK	Avg %
3	219000	48000	66.7
5	157500	30750	58.3
10	96000	15200	47.5
20	53400	7000	39.5
50	23300	2700	36.3
100	10000	1250	34.5

Table 3: Different number of read operations per transaction

3.4 Different number of write operations per transaction

The following table represents the results according to giving different values for the number of write operations per transaction. As in the previous sections, the experiments will be carried out using the following values:

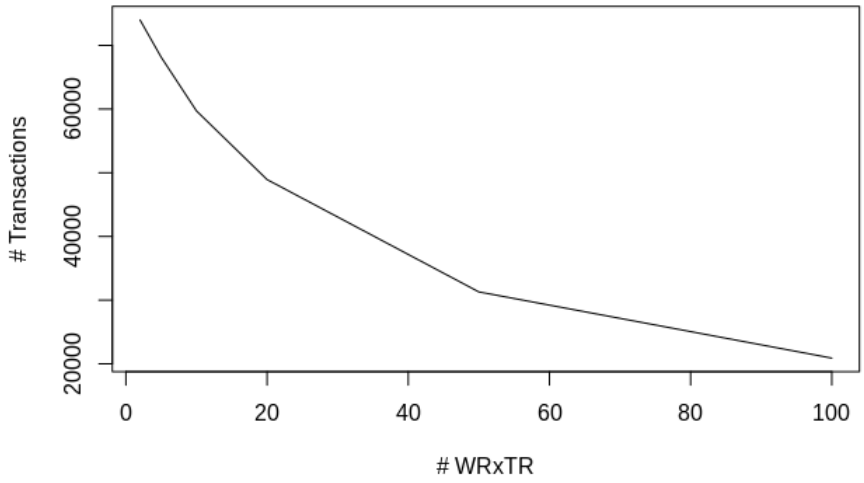
opty:start(3,10,X,2,3).

Where X represents the number of write operations per transaction.

# WRxTR	# Transactions	# OK	Avg %
2	74001	49345	66.7
5	68146	35129	51.5
10	59712	24988	41.8
20	48931	18073	36.9
50	31291	11225	35.9
100	20896	7595	36.3

Table 4: Different number of write operations per transaction

From these results, we can see that the number of transactions keeps decreasing as we increment the number of write operations per transactions. The decrement it follows looks exponentially shaped with limit 0, as we can see in the following figure. In the other hand, the average percentage of OK transactions has already reached a limit around 36% from 20 write operations per transaction.



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

For this experiment we have to change the read and write operations for that total five experiments have been done with varying read and write operations. The command used for this is as follow:

```
opty:start(3,10,RDxTR,WRxTR,3).
```

Here RDxTR represent the number of read operations per transaction and WRxTR represent the number of write operations per transaction.

# RDxTR	# WRxTR	# Transactions	# OK	Avg %
3	100	6455	2941	45.5
5	50	8409	3683	43.8
10	20	8800	4299	48.8
20	10	6788	2813	41.3
50	5	3042	1234	40.5
100	2	1363	580	42.4

Table 5: Ratio of read and write operations per transaction

From the above table it could be seen that by increasing the number of write operations with respect to read operations, increase the accuracy rate but the balance between read and write could result for best improvement that is 10-20 with accuracy 48.8.

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

The OPTY code has been modified in order to perform this experiment. This modification can be found in the directory src/experiment6 in the code delivered. It adds a new parameter to the start function signature which is a percentage. For each client, the result of multiplying this percentage by the total number of entries, is used as the size to create a random subset of entries. The clients will only use their own subset to request their transactions.

The different executions in this experiments have been run as:

```
opty:start(3,10,3,2,3,X).
```

Where X represents the percentage used to create the entry subsets.

%	# Transactions	# OK	Avg %
0.2	92178	78410	84.9
0.4	80296	56014	69.8
0.6	76873	53631	69.8
0.8	72723	49689	68.3
1.0	71742	47984	66.8

Table 6: Different percentages for the clients' entry subsets sizes

As expected we can see that the percentage of OK transactions decreases when the size of the clients' subsets increases. When the subsets are small, the clients have less chances to compete with the other clients for the entries.

3.7 Distributed execution

The OPTY algorithm has been modified for this section that is under src/distributedexecution folder. one node from the server side with the following command:

```
optyserver@localhostwith opty:start_server(12).
```

and the other node from the client side that is:

```
clients@localhostwith
opty:start_clients(5,3,1,1,2,5,optyserver@server,3).
```

Has been executed. The handler in the distributed Erlang network runs in the clients node as it performs all the operations done by clients.

3.8 Forward Validation

For this section we have modified the OPTY algorithm so it uses Forward Validation instead of Backward Validation. This modified code can be found under the src/forwardValidaion directory in the delivery.

A parameter *TransactionId* has been added to the Handler status to identify each transaction. The Handler module has been selected to store the transaction id because one Handler instance is created for every transaction, and all transaction operations are made through this module.

The Entry module has been modified so each entry keeps a list of the active transactions that read it. A transaction will validate only if all of its written entries have an empty list of active readers. We also included a function in this module that permits to delete an specific transaction from an entry's active readers list.

This function will be used from the Validator module when validating a transaction (as it is not active any more). The Validator module has also been modified to validate the transactions' writes instead of reads.

The following table shows the code results when executing it for different number of clients. The number of transaction and successfull transactions of this experiment has been averaged among all clients. Each execution has been ran with:

```
opty:start(X,100,3,2,3).
```

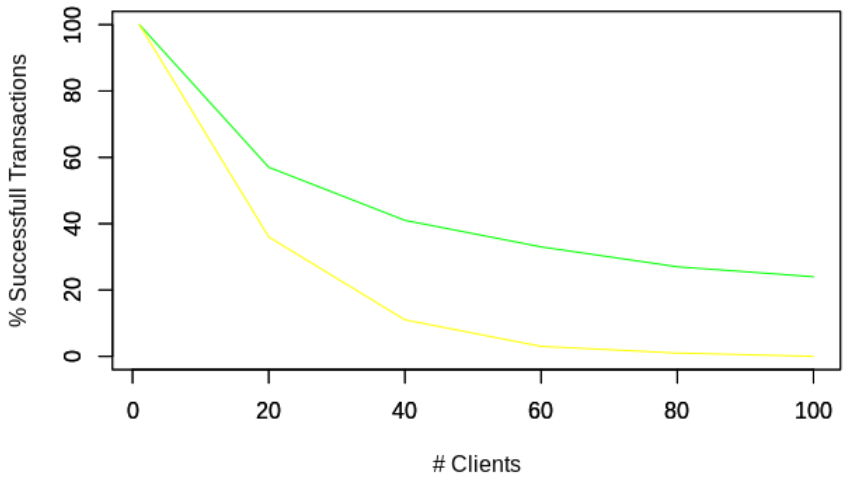
Where X is the number of clients.

# Clients	**# Transactions**	**# OK**	**Avg %**
1	111166	111166	100
20	10462	3739	36.5
40	5244	572	11.2
60	3445	115	3.4
80	2544	25	1.1
100	1890	8	0.3

Table 7: Results for different number of clients with Forward Validation

From the results we can see that the three metrics shown in the results decrease as the number of clients gets closer to the number of entries. Actually, the percentage of successfull transactions tends to 0 when the number of clients tends to the number of entries.

The following graph compares the average number of successfull transactions for different number of clients between the Backward Validation (green) and Forward Validation (yellow) implementations.



From the graphic we can observe that the percentage of successfull transactions follow similar shapes but with different boundaries. In the specific configuration used we ran the experiments (the parameters apart from number of clients) this percentage have a boundary around 50% for Forward Validation and 0% for Backward Validation. Backward Validation strategy seems to have a better performance in terms of transactions succesfulness.

4 Personal opinion

This seminar has an appropriate complexity to understand the OPTY algorithm but not getting stucked and lost. The way to implement the code filling the blanks helps to understand the really crucial part of the code without wasting time implementing the rest. Like in the previous seminar, the questions that required some modification of the code were the more challenging ones. Maybe a little bit more difficult this time, but not too difficult anyway. The general impression is good and we think it is worth to repeat it the next year.

5 References

- **GitHub Repo:** `https://github.com/SergioMD15/CPDS-OPTY`