

# Implementation of SS2PL and OCC in Google AppEngine

Arijit Khan and Gaurav Mehta

Department of Computer Science, University of California, Santa Barbara

{arijitkhan, gaurav\_mehta}@cs.ucsb.edu

## ABSTRACT

In this report, we describe our implementation of two transaction scheduling protocols, namely SS2PL [3] and OCC [7] using Google AppEngine [8] as the back-end data store. We then use a customized version of the `http_load` tool [4] and a set of workload files [4] to test the performance of our implementations. The total number of read and write operations in these workload files are varied to measure the performance under different load conditions. We also perform a comparative analysis of both the protocols in terms of throughput, minimum, maximum and average response time for a transaction to complete, the number of transactions that are aborted (for OCC), and the amount of time an operation is blocked (for SS2PL).

## Keywords

Scheduling, Lock, Optimistic concurrency control, Commit, Serializability, Strictness, Recoverability, Validation, Abort.

## 1. INTRODUCTION

In databases and transaction processing, a schedule (transaction history) is called serializable if it has the Serializability property [1], i.e. its outcome is equal to the outcome of its transactions executed serially. Transactions are normally executed concurrently to achieve efficiency. Serializability, therefore, is the major correctness criterion for concurrent transactions' executions. It is considered the highest level of isolation between transactions, and plays an essential role in concurrency control.

In order to maintain this notion of correctness via serializability, scheduling of transactions can be done in both pessimistic and optimistic manner. Pessimistic approach makes use of locks. Locks can be either read (shared) or write (exclusive) locks. Write locks serialize both read and write access, whereas read-only locks only read access. When a write lock is held, no other write or read locks can be acquired. When a read lock is held, others can acquire read locks. However, to acquire write locks, one has to wait until all read locks have been released. When scheduled concurrently, write locks always have precedence over read locks. Note that (if enabled) read locks can be upgraded to write locks. However, locking mechanisms suffer from deadlocks [2]. Moreover, the Lock maintenance is an additional overhead that is not present in the sequential execution. For example, even for read-only queries, which cannot possibly affect the integrity of the data, we need locks in order to guarantee that the data being read are not modified by other transactions at the same time [7].

Strong strict two phase locking (SS2PL) [3] is a common mechanism utilized in database systems to enforce both conflict serializability [1] and strictness [5] (a special case of recoverability [6]) of a schedule. In this mechanism each datum is locked by a transaction before accessing it (any read or write

operation). As a result, access by another transaction may be blocked, typically upon conflict, depending on lock type and the other transaction's access operation type. Employing an SS2PL mechanism ensures that all locks on data on behalf of a transaction are released only after the transaction has ended (either committed or aborted).

Optimistic concurrency control (OCC) [7], on the other hand, is a concurrency control method used in relational databases without using locking. OCC is based on the assumption that most database transactions don't conflict with other transactions, allowing it to be as permissive as possible in allowing transactions to execute. There are three phases in an OCC transaction:

1. Read: The client reads values from the database, but stores all writes in the local storage.

2. Validate: When the client wants to commit the transaction, it enters into a validation phase. During validation, an algorithm checks if the changes to the data would conflict with either already-committed transactions in case of backward validation schemes, or with currently executing transactions in the case of forward validation schemes. If a conflict exists, the entire transaction is aborted and retried at some later time. Note that, this validation can be done in both atomic (serial) or non-atomic (parallel) manner.

3. Write: If the validation succeeds, the transaction commits; i.e. it makes permanent changes in the database from the local storage.

Optimistic concurrency is generally used in environments with a low contention for data. When conflicts are rare, validation can be done efficiently, resulting a higher throughput than other concurrency control methods. However, if conflicts happen often, the cost of repeatedly restarting transactions hurts performance significantly.

In this project, we designed a serializable read/write data store using the Google AppEngine [8]. AppEngine allows us run our own web applications on Google's infrastructure, and Google also provides development tools and APIs to help with application development. We have implemented a web application that provides a transactional back-end data store. All operations (read, write, abort, and commit) are exposed to the user via a web interface. From this web page, a user can read and write multiple items to the data store. Each read or write will issue a new http request that will be handled by the back-end. After some number of read and write operations, the user can then abort or commit the transaction. The data store must ensure that the transactions are executed in a conflict serializable manner. In order to achieve this, we used two different concurrency control protocols, SS2PL and OCC, which are described above. Note that, AppEngine is a very

new service, and so part of the challenge was to learn how to develop applications for this new environment.

The rest of the report is organized as follow. In section 2, we summarize the schema design of the two scheduling protocols. The actual implementations are described in section 3. Additional implementations required for Performance testing are given in section 4. We analyze the results in section 5 and conclude in section 6.

## 2. DESIGN SCHEMA

In this section, we shall describe the schema used for our implementation of SS2PL and OCC respectively.

### 2.1 SS2PL

Our implementation of SS2PL uses the following classes:

A *Dataobj* : current value of the dataobj and the number of shared and exclusive locks it has..

Attribute	Function
i. dataobj	data identifier
ii. value	current data value
iii. shared_lock	Count for read locks on this dataobj
iv. exclusive_lock	Count fot write locks on this dataobj

B. *Lock*: lookup table for transaction-lock association

Attribute	Function
i. dataobj	data identifier
ii. txn_id	Transaction id that holds the lock on dataobj
iii. shared_lock	1 if it holds shared lock,otherwise 0
iv. exclusive_lock	1 if it holds shared lock,otherwise 0

C. *Blocked\_operation*:Repository for the currently blocked operation

Attribute	Function
i. Wctype	operationtype:1-write,2-commit,3-read
ii. txn_id	Transaction id
iii. value	Value to be written for Wctype=1
iv. dataobj	data object
v. order	Time of insertion into the queue

D. *Block\_Txn*: Transactions currently blocked.

Attribute	Function
i. txn_id	Transaction id

E. *Log*: Successful writes and successful blocked reads.

Attribute	Function
-----------	----------

i. txn_id	Transaction id
ii. dataobj	dataobject
iii. operation	The operation type
iv. order	Time of the completion
v. pre_value	Value of dataobj before the operation
vi. post_value	Value of dataobj after the operation

### 2.2 OCC

Our implementation of OCC with parallel validation is based on the algorithm described in [7]. We use the following classes:

A. *DataObj*: Parmanent Data Store.

Attribute	Function
i. dataobj	data identifier
ii. value	current data value

B. *Txn*: Attributes of individual transaction.

Attribute	Function
i. old_Txn_id	Transaction id as input by the user.
ii. new_Txn_id	System assigned transaction id after it validates and writes permanently.
iii. st	Value of tnc at the time of transaction starts
iv. fn	Value of tnc after transaction validates and before it starts writing permanently.
v. start_time	Transaction start time. used for performance analysis only.
vi. finish_time	Transaction finish time (if validation succeeds). used for performance testing only.

C. *Live\_Read*: keeps track of reads of all live (active) transactions.

Attribute	Function
i. old_txn_id	Same as B.i
ii. dataobj	Same as A.i

D. *Local\_Write*: stores local writes of all live transactions.

Attribute	Function
i. old_txn_id	Same as B.i
ii. dataobj	Same as A.i
iii. value	Same as A.ii

E. *Commit\_Txn\_Write*: stores the writes of all committed transactions.

Attribute	Function
i. new_txn_id	Same as B.ii
ii. dataobj	Same as A.i
iii. value	Same as A.ii

Note that, if a transaction is validated, its local\_writes are moved into Commit\_Txn\_Write. The mapping from old\_txn\_id to new\_txn\_id for each transaction is available in the Txn class. If the transaction is aborted, all its local\_writes are deleted.

#### F. Counter: Logical Time

Attribute	Function
i. tnc	tnc is incremented (atomically) by one each time a transaction is validated and finished writing permanently

### 3. IMPLEMENTATION PROCEDURE

We explain our implementation algorithms with the help of flow diagrams.

#### 3.1 SS2PL

a) Read

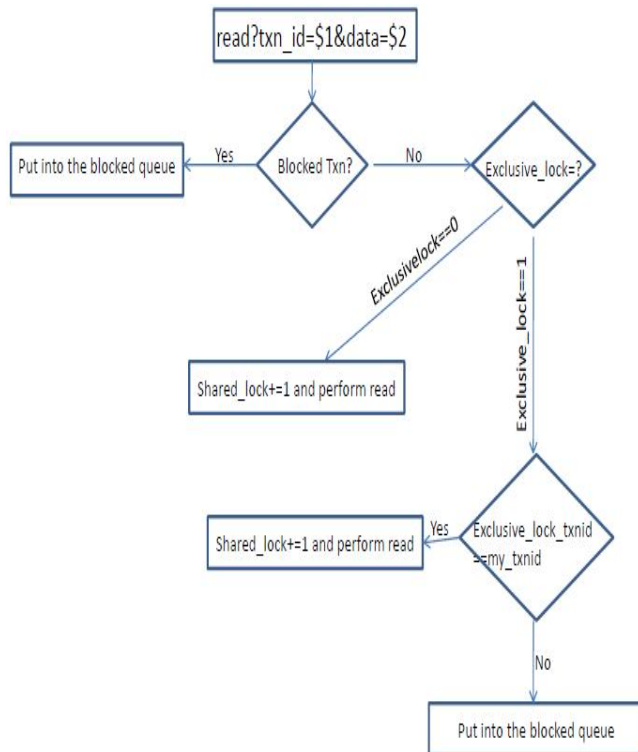


Figure 1: Flowchart for SS2PL Read

b) Write

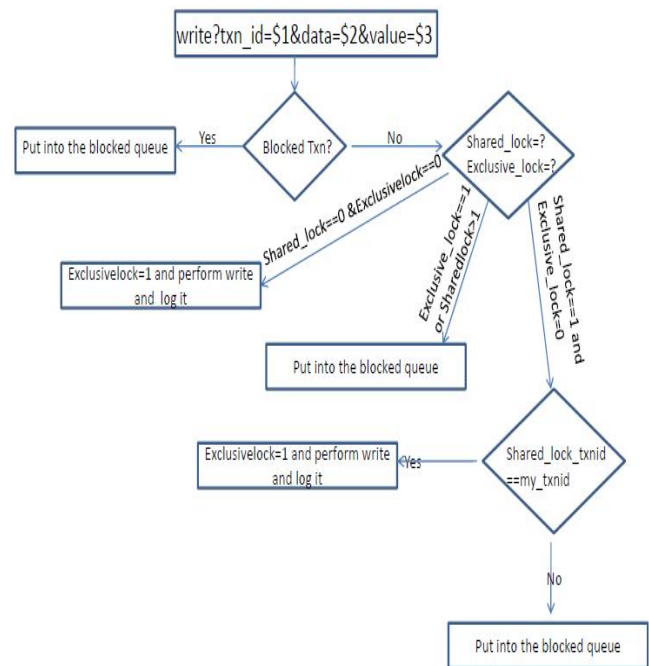


Figure 2: Flowchart for SS2PL Write

When a transaction gets blocked, the subsequent operations of that transaction go into the blocked queue and as a result the transaction is not able to see the results of the operations when they are performed at some later time. We provide a *Retry* functionality wherein a transaction can query to see the action of its previously blocked operations.

c) Commit and Abort:

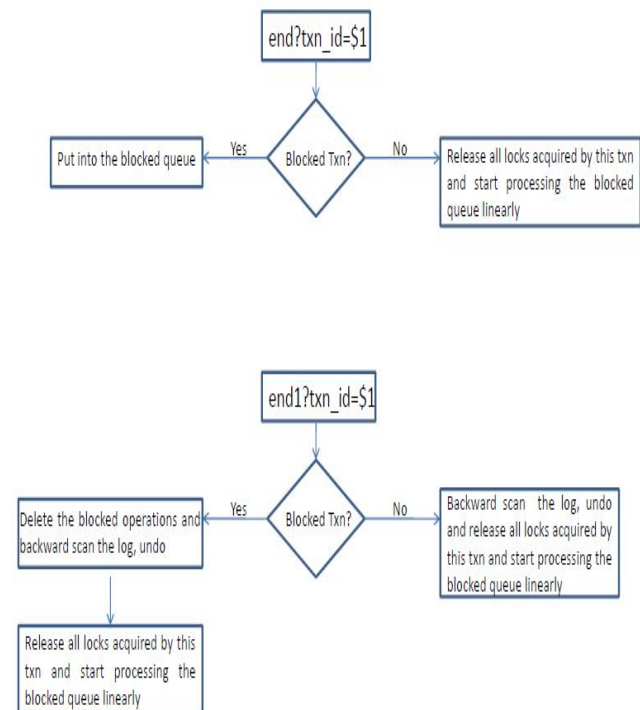


Figure 3: Flowchart for SS2PL Commit or Abort

After a commit or abort, we scan the blocked queue in the increasing order of time and check whether the operation is a candidate for consideration. An operation is a candidate for consideration if the blocked queue does not contain any operation from this operation's transaction before it. Once it becomes a candidate, then if the operation can be performed, it is performed and it gets deleted from the blocked queue and the scan moves forward. Note that in case the operation is a commit operation, we perform the commit operation (if it is a candidate for consideration), and stop the scan. Eventually the scan starts again from the starting as a result of the commit operation function call.

### 3.2 OCC

a) Read:

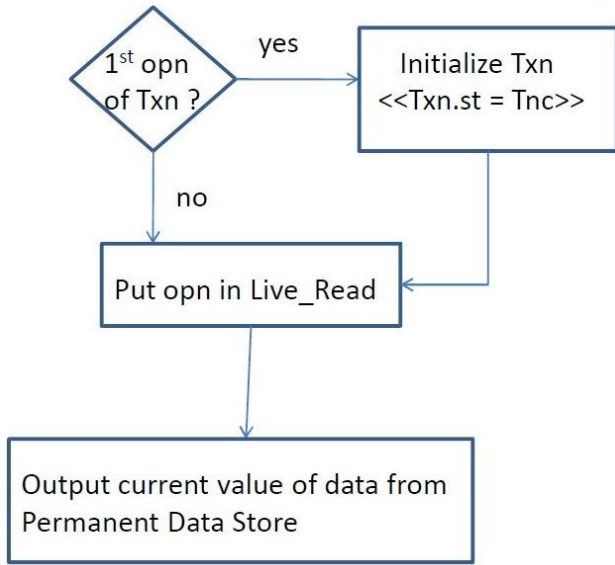


Figure 4: Flowchart for OCC Read

b) Write:

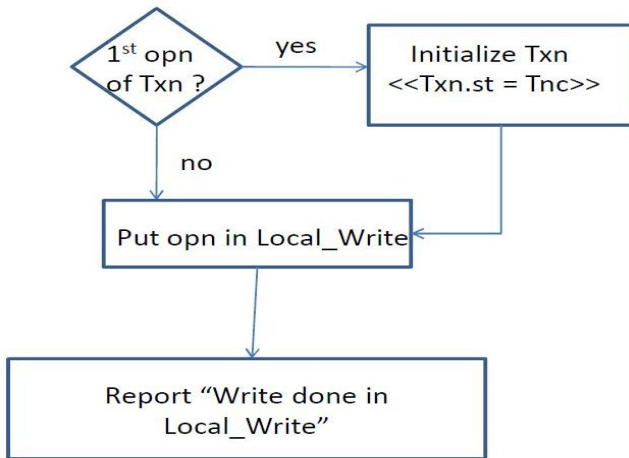


Figure 5: Flowchart for OCC Write

c) Validation:

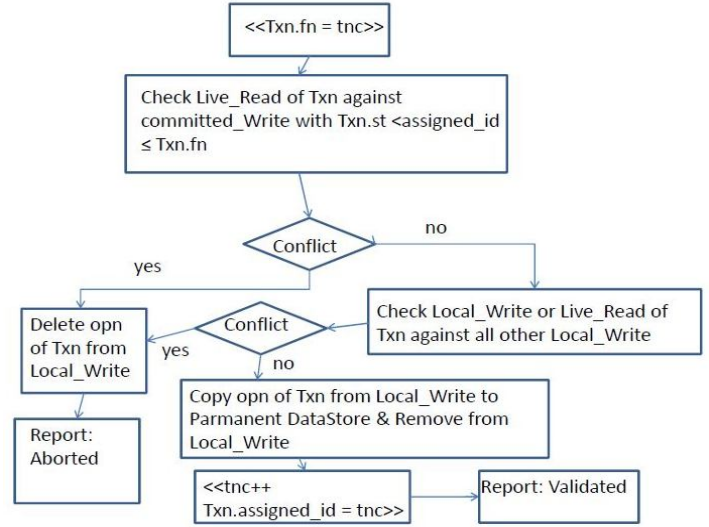


Figure 6: Flowchart for OCC Validation

Note that, we do not need to remove the live\_reads of an active transaction after its validation is done. This is because, we never check conflicts against the live\_reads of active transaction; rather for each live\_read of current transaction is being checked against the local\_writes of all other active transactions.

d) *Abort*: In case of user initiated aborts, we simply remove the local\_writes of that transaction.

## 4. SETUP FOR PERFORMANCE TESTING

We perform some additional setup required for performance testing.

### 4.1 SS2PL

We keep count of the total number of operations being blocked at some part of their execution. We also add up the blocking time for all these operations. Since we do not implement explicit deadlock detection techniques, we slightly change our implementation for performance testing; so that, a transaction is aborted if its commit cannot be performed immediately (commits are not stored in blocked queue for performance testing). Otherwise, for a large workload file consisting of many transactions and equivalent number of reads and writes generally create all-deadlock situation at the end and we do not get any significant result. Besides, the start time and finish time of each transaction is monitored to find out the maximum and minimum response time required for a transaction.

### 4.2 OCC

When a transaction is first encountered, we store its start time. When it is finished (either validated and performed its permanent writes or aborted), we check its finish time and thereby we get the total time for each validated transaction. This helps us to keep track of the maximum time and minimum time for a transaction. Moreover, we keep count of how many transactions are being validated and how many of them are aborted.

We use a customized version of the http\_load tool [4] and a set of ten workload files [4] to test the performance of our implementations. Five of the workloads (Mixed) have transactions

made of up a mix of reads and writes (approximately 65% read operations, 35% write operations) In the other five workloads (Read\_Skewed), 75% of the transactions are read only and 25% of the transactions are a mix of reads and writes. The keys accessed by each transaction are selected at random from the interval (1, 25). The total time to perform all the transactions in each workload can be found from the output of http\_load. We measure the average response time of a transaction = Total time taken / Total no of transactions; whereas, throughput = No of Successful transactions per unit time. We run all the workloads in Google AppEngine for both SS2PL and OCC.

## 5. ANALYSIS OF RESULT

### 5.1 SS2PL

Table 1 shows the number of successful and aborted transactions, as well as the throughput for different workload files in SS2PL Performance Testing. As we can expect, number of successful transactions and therefore, throughput both are less in case of Mixed workloads than those for Read\_Skewed workloads.

Workload Files	No of Successful Transactions	No of Aborted Transactions	Throughput (No of Successful Transactions per minute)
Mixed 1	313	687	4.01
Mixed 2	350	650	4.18
Mixed 3	352	648	4.36
Mixed 4	207	793	2.53
Mixed 5	311	689	4.16
Read_Skewed 1	757	243	11.38
Read_Skewed 2	739	261	11.51
Read_Skewed 3	727	273	11.42
Read_Skewed 4	747	253	10.70
Read_Skewed 5	746	254	11.01

Table 1: SS2PL Performance Testing Result of Throughput

Workload Files	Average Response Time (sec)	Minimum Time for a Transaction (sec)	Maximum Time for a Transaction (sec)
Mixed 1	25.48	7.61	57.69
Mixed 2	27.94	5.45	66.36
Mixed 3	25.97	7.02	66.96
Mixed 4	25.89	6.63	55.08
Mixed 5	24.42	5.75	56.68
Read_Skewed 1	20.30	4.86	47.06
Read_Skewed 2	19.56	5.71	73.52
Read_Skewed 3	20.42	5.58	47.09
Read_Skewed 4	21.30	5.89	49.31
Read_Skewed 5	20.69	5.4	51.87

Table 2: SS2PL Performance Testing Result of Responsiveness

The above table gives us the average response time and maximum and minimum response time for a successful transaction in SS2PL Performance Testing.

Workload Files	Total Blocked operations	∑time spent by each blocked operation in the blocked queue (sec)	Average time spent by each blocked operation( in sec)
Mixed 1	3616	25574.93	7.07
Mixed 2	3711	27606.31	7.43
Mixed 3	3602	25457.02	7.07
Mixed 4	4538	43967.10	9.69
Mixed 5	3685	24688.67	6.70
Read_Skewed 1	1375	8382.56	6.09
Read_Skewed 2	1400	8501.47	6.07
Read_Skewed 3	1323	8028.81	6.07
Read_Skewed 4	1472	9991.15	6.79
Read_Skewed 5	1404	8956.66	6.38

Table 3: SS2PL Performance Testing Result of Blocked queue

The above table gives us the total number of blocked operations (successful and aborted transactions combined) and the summation of the time spent by each blocked operation in SS2PL Performance Testing

### 5.2 OCC

Table 4 shows the number of validated and aborted transactions, as well as the throughput for different workload files in OCC Performance Testing. As we can expect, number of validated transactions and therefore, throughput both are less in case of Mixed workloads than those for Read\_Skewed workloads.

Workload Files	No of Validated Transaction	No of Aborted Transaction	Throughput (No of Validated Transaction per minute)
Mixed 1	309	691	5.07
Mixed 2	293	703	4.80
Mixed 3	301	699	4.92
Mixed 4	300	700	4.90
Mixed 5	296	704	4.18
Read_Skewed 1	503	497	6.96
Read_Skewed 2	493	507	5.66
Read_Skewed 3	516	484	8.17
Read_Skewed 4	507	493	10.96
Read_Skewed 5	514	486	8.40

Table 4: OCC Performance Testing Result of Throughput

Workload Files	Average Response	Minimum Time for a	Maximum Time for a
----------------	------------------	--------------------	--------------------

	Time (sec)	Transaction (sec)	Transaction (sec)
Mixed 1	3.28	2.99	42.90
Mixed 2	3.66	3.23	35.17
Mixed 3	3.67	3.34	44.19
Mixed 4	3.57	3.02	30.57
Mixed 5	4.25	3.51	43.56
Read_Skewed 1	4.33	3.64	25.73
Read_Skewed 2	5.22	4.65	30.86
Read_Skewed 3	3.79	2.92	30.62
Read_Skewed 4	2.77	2.56	25.08
Read_Skewed 5	3.67	5.14	37.49

Table 5: OCC Performance Testing Result of Responsiveness

The above table gives us the average response time and maximum and minimum response time for a transaction in OCC Performance Testing. Note that, it is not always necessary that these values for Mixed workloads will always be higher than the corresponding values for Read\_Skewed workloads (see Mixed 1 and Mixed 4 in the above table). This is due to the fact that we consider both the validated and aborted transactions here and some transactions are aborted quickly.

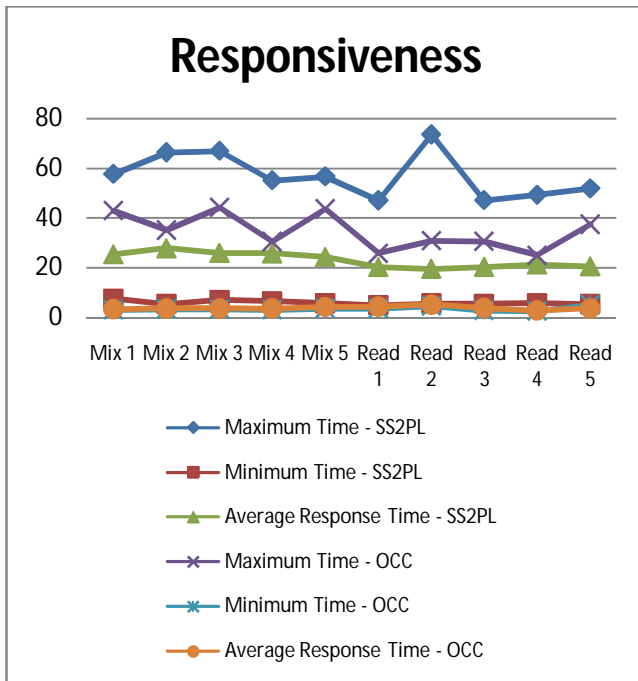


Figure 7. Responsiveness in Performance Testing

From Fig. 7, we can see that the value of Maximum time, minimum time and average response time for each workload file is higher than these values for OCC respectively. This is because OCC has no locking. Fig. 8 shows that the no of aborted transaction is less in case of SS2PL, because they are blocked and performed later; whereas in case of OCC, they are aborted if there were conflicts.

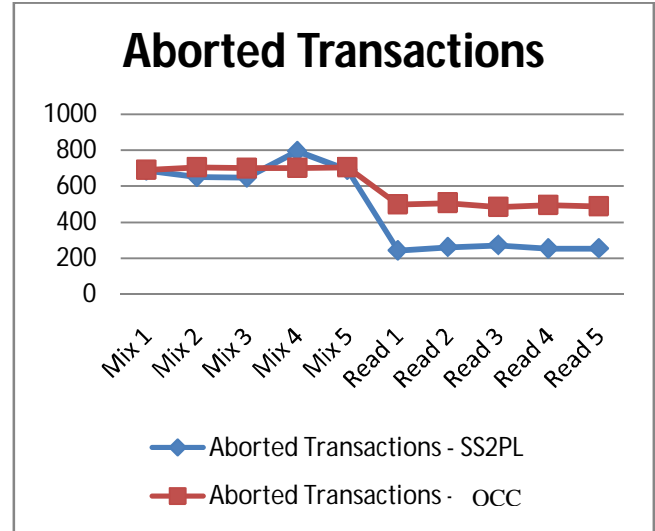


Figure 8: No of Aborted Transactions in Performance Testing

## 6. CONCLUSION

In this project, we have implemented two transaction scheduling algorithms, e.g. SS2PL and OCC using Google AppEngine as the back-end data store. We use AppEngine only for data storage and do not take help of its any inherent scheduling properties. We have also analyzed the results of our performance testing using http\_load and ten different workload files. Note that, http\_load goes through the workload file, one line at a time, and issues an http request for each URL. It issues the next request only after it receives a response for the current request. Considering this fact, we have simplified the blocked\_queue scanning algorithm for SS2PL as mentioned in section 3.1 It might be interesting to modify this algorithm in case the requests come concurrently. However, our OCC implementation works fine even if the requests are given concurrently.

## 7. REFERENCES

- [1] Philip A. Bernstein , Nathan Goodman, "Concurrency Control in Distributed Database Systems", ACM Computing Surveys (CSUR), v.13 n.2, p.185-221, June 1981
- [2] J.C. Corbett., "An Empirical Evaluation of Three Methods for Deadlock Analysis of Ada Tasking Programs," Proc Int'l Symp. Software Testing and Analysis (ISSTA), T. Ostrand, ed., pp. 204-215. ACM Press, Aug. 1994.
- [3] Shapour Joudi Begdillo, Fariborz Mahmoudi, Mehdi Asadi, "Improving Strict 2 Phase Locking (S2PL) in Transactions Concurrency Control", Proc Convergence Information Technology, p. 1468-1473, Nov 2007.
- [4] [http://cs.ucsb.edu/~cs274/project\\_performance.html](http://cs.ucsb.edu/~cs274/project_performance.html)
- [5] Udo Kelter, "Strictness and Serializability", 3rd annual symposium on theoretical aspects of computer science (STACS 86), p. 252-261, 1986.
- [6] Hassan M. Najadat, "A New Approach for Recoverable Timestamp Ordering Schedule", Proc World Academy of Science, Engineering and Technology, v. 13 may 2006.
- [7] Kung, H.T. and Robinson, J.T., "On optimistic methods for concurrency control", ACM Transactions on Database Systems 6 2, pp. 213-226, 1981.
- [8] <http://code.google.com/appengine/>