Modelling Snapshot Isolation Performance

© Dmitri Vasilik

Saint Petersburg State University
Dmitri.Vasilik@gmail.com

Abstract

Snapshot Isolation (SI) level is extensively used in commercial database systems. We developed a simple SI implementation protocol for distributed DBMS and implemented it in the Apache HBase. The work presents the performance evaluation of the protocol. We have measured the performance of a single-node system and modeled the performance of a distributed HBase cluster.

1 Introduction

In the last ten years the special class of data-intensive web-application has emerged. Youtube, Google Maps, social networks, etc. represent the applications of the class. This applications can be characterized by the following features: they work with large volumes of data and typically do not need ACID transactions.

The choice of the concurrency control to employ in the application is of great importance, since it affects the performance of the application as well as the complexity and cost of development. For example, choice of eventual consistency (EC) as an isolation level may result in increased complexity of programming model. Thus, application designer has to carefully look for the most suitable compromise between level of guarantees he gets and performance of isolation level protocols.

The Snapshot Isolation (SI) isolation level was introduced in [2]. It is widely adopted by the commercial and open-source systems thanks to its ability to cope with read intensive workloads and high degree of consistency guarantees. It is even used instead of serializable isolation level in Oracle and PostgreSQL. This decision can be justified by level of guarantees SI actually offers.

The transaction executed under SI reads data from a snapshot of the data as of the time the transaction started. Two transactions are called competitive if they have overlapping execution intervals and they have written the same data item. When the transaction is ready to commit, it is assigned the commit timestamp and in case if there is no competing transactions in the system it commits. This feature is called "First-Commiter-Wins".

We developed SI protocol for distributed DBMS. We have presented it in [12] along with its implementation prototype. We have chosen HBase to implement it in because it is open source distributed data storage which has natively supported only EC[1]. In [12] we have measured the performance of the prototype in a single-node mode and compared it with that of HBase, but we had not an opportunity to run the system in a distributed mode. Compared to [12], in this work we present performance model of the protocol and the comparison of the protocol simulation results with HBase simulation results for the distributed mode.

The remainder of this paper is organized as follows. In Section 2 we discuss literature works on the DBMS performance evaluation with concurrency control performance comparison being in focus. Section 3 contains the description of the protocol and its implementation. Performance model is provided in Section 4. Section 5 presents results of conducted simulation experiments. Section 6 contains discussion of protocol implementation performance along with model validation issues. In section 7 the results are summed up.

2 Related Work

A lot of concurrency control algorithms was introduced in the last 30 years. Considerable research was devoted to evaluating the performance of concurrency control algorithm in the 80s and the earlier 90s.

In [1] authors has examined the assumption set employed in concurrency control performance (CC) modelling. Authors has investigated the reasons of apparent contradictions in performance results of earlier studies. In particular, the common assumption of infinite resources was critically examined. The paper [11] lists the CC methods along with results gained from analytical and simulation studies. For a given CC method the models employed for its performance evaluation are briefly described. In [10] issues in modelling performance are discussed. The paper contains tens of questions to be solved by the researcher before he starts to implement his model along with some critique of using one's intuition in performance modelling.

Works [7, 6, 5] are focused on the distributed systems performance evaluation, however, this studies do not have the CC methods in the main focus. In [6] performance models classification based on the model assumption set is provided. The paper [7] contains a detailed comparison of Shared Disk and Shared Nothing (SN) architectures. In particular, the performance of Scan and Join operators is discussed. The paper [5] contains a comprehensive SN DBMS performance study, perfor-

Proceedings of the Spring Researcher's Colloquium on Database and Information Systems, Moscow, Russia, 2011

[1] Optimistic concurrency control was the only available alternative for HBase by the moment we started implementation.

mance is modeled using three different workloads, one of them being generated from database traces.

The work [8] presents an analytical SI performance model for a standalone server machine.

Recently there was a number of studies related to SI implementation in distributed systems [14, 4, 13]. The work [4] presents a technique to preserve the behaviour of centralized SI system in guaranteeing global SI in distributed, lazily replicated system. The technique leverages local SI concurrency control. A simulation model was developed to study the cost of using the technique. This model is rather simple in aspects of single-node performance, e.g. the service time per operation is specified as model parameter.

The article [13] presents a cloud storage system named ecStore. The article contains a brief description of concurrency control protocols used, but it does not contain CC performance comparison.

In the paper [14] an approach to use HBase as a cloud database solution with global SI is described. The paper presents an optimistic protocol, which uses 4 special tables and several queries to provide SI gaurentees. The protocol is implemented on top of bare-bones HBase on the client side, it's imlementation is rather high-level, it does not introduce any changes to server code. Authors have conducted several experiments in a distributed environment (3-machine cluster) to evaluate the cost of employing the proposed protocol.

In this work we present performance models for our protocol and HBase and the comparison of simulation results. To the best of our knowledge the performance model of Key Value store like HBase has not been presented yet.

3 Implementation Protocol

3.1 Protocol Description

Let us suppose that current transaction state, its start timestamp and end timestamp (if a transaction has been already committed or aborted) are available at the execution time.

Read operation execution is straightforward. To read a data object x the transaction t performs a selection of all versions of x, committed before start time of t. Then it traverses the obtained list of versions to find the latest committed version. If the list contains version written by t, this version is read instead of the latest committed.

Before the value of x can be updated the transaction has to ensure, that no competitive transaction had already written x. If x was updated by transaction, which was committed after t had started or by active transaction, than t aborts.

We have formally proved the correctness of the protocol in [12], but the prove is not in the scope of this work.

3.2 Data Storage Design Issues

We have implemented our protocol in HBase. HBase is an open-source distributed data storage written in Java. It was made after Google Bigtable. In [3] Bigtable is presented along with design and implementation decisions made by developers.

Authors has called Bigtable "a sparse distributed persistent multidimensional sorted map". Bigtable API is very simple. HBase API is also restricted to `get`, `put`, `delete` and `scan` operations. The data processing necessary for a query can not be pushed as close as possible to data, as it is usually done in distributed RDBMS. If a client wants to update data according to a special rule, it requests the data, updates it locally and puts the updated data to the storage. That is, complex query execution is decomposed into two or more phases. As it was said before, initially only eventual consistency isolation guarantees were provided by HBase. HBase/Bigtable users who needed transactional execution, were suggested to implement it themselves using data timestamps.

HBase cluster consists of servers of two types: master nodes and region nodes. Every region server manages a number of regions. HBase uses horizontal partitioning, each region contains data for an interval of keys.

HBase is column oriented storage, that is, data is stored on a per column basis in main memory and in the file system. Every column store has its own set of files and a per file buffer pool employing the classical LRU discipline to manage pages. HBase use immutable files to store data. When the total space occupied by updates kept in main memory exceeds specified threshold a new file is written to the file system. This is done in a special thread. HBase uses write ahead log (WAL), that is, every update is written to log before it gets to main memory store.

3.3 Protocol Performance Discussion

The "First-Commiter-Wins" feature has an optimistic nature, thus SI definition naturally accepts the optimistic protocol. Our approach is not optimistic, it should be rather called "First-Updater-Wins".

Although the algorithm is nonlocking, it is more close to restart oriented locking methods such as immediate-restart. Immediate-restart algorithm description, its performance modelling and comparison with blocking and optimistic methods can be found in [1]. More sophisticated restart oriented methods exist such as wound-wait or wait-die, which could outperform the immediate-restart method in most cases ([11]). We did not attempt to build a protocol on top of this ideas.

Before a data object is written the version check is performed. All data object versions made after the transaction had started must be selected and traversed. It may take a considerable amount of time, because some of committed versions of data object may have been already written to the file system. In this case a number of page reads is to be performed.

Some tricks could be used to avoid unnecessary disk accesses or at least reduce their probability. The main memory store could be assigned a timestamp each time its contents are flushed to a new file. The version check performed in main memory is sufficient to ensure "First-Writer-Wins" feature if the transaction started after the main memory store had been written to disk last time. The second trick which may be used is to write only the oldest versions to file, keeping new ones in memory. Given average transaction execution time t_{avg}, we can keep versions with timestamps greater than $now() - t_{avg}$.

To ensure transactional execution of queries in distributed data storage we used the distributed commit protocol. We have chosen a protocol similar to Two Phase

Commit (2PC) for its implementation simplicity. The classical 2PC successfully commit scenario consists of the following steps. The transaction initiator waits for READY message from all the participated servers. When all the participants voted to commit, it sends them COMMIT message and waits for ACK messages. After all ACK messages were received, initiator finishes the transaction.

Since HBase uses WAL, when subtransaction sends READY message to the transaction home node (client machine), it has no additional work to do before it can commit. It can not be aborted by other local subtransaction. When the transaction initiator receives READY message from all the execution region servers, it just sends them COMMIT message and commits the transaction locally. It does not wait for ACK messages from executor servers as in the classical 2PC.

Let us summarize the overheads introduced by the protocol.

- Employing distributed commit protocol leads to additional communication overhead.
- Version check may lead to additional (local) disk accesses and may consume CPU.
- Certain resources are wasted due to additional aborts by protocol reasons.

4 Simulation Performance Model

The simulation model was written in Java using Simkit simulation package [9].

Our model is rather hardware resource bounded than data contention bounded. We believe that HBase was not intended to be used for the workloads with high data contention. Our model was not aided to solve some fundamental questions, it concentrates on answering the engineering questions only. We have modeled the particular distributed data storage performance, so the model is much less universal than models employed in mentioned related works. However, the model is rather simple. It abstracts from some details. For example, the primary copy replication used in HBase, may affect the performance, but it is not reflected in the model.

The model will be described in terms of queries, we will use the term transaction when we need to outline the differences.

4.1 Modelling Assumptions

We use a close system model. There is always the same number of queries in the system. After one query had completed its execution, the new one immediately replaces it. Since the concurrency control performance hardly depends on the active query count in the system, we decided to keep these number constant.

The cluster is modelled as a queueing network, each node is itself modeled as a queueing network.

We use quite common assumption called "homogeneity assumption" in [6].

- All database sites have the same structure and the same service capacity.
- The amounts of data allocated to each unit are equal.

Parameters		Settings
CPU:	cores per PE	2
	processor capacity (MIPS)	9000
#avg instr.:	BOE and EOE steps	5000
	per object reference	25000
	for message	5000
	for IO operation	3000
Disks:	count per PE	1
	access time mean (ms)	8
	page transmission time (ms)	0.2
Network:	throughput (Mbit/sec)	100
	min packet size	540
Database:	page size (Kb)	4
	count of pages per PE	53000000
	count of pages in buffer pool	500000

Table 1: System parameter settings.

- Data accesses are spread among nodes in uniform manner.

We do not use a sophisticated model of a communication network as well, because we believe that network should not be a bottleneck. In our model there is a virtual connection between every query initiator (client) and executor node with 100Mbit/sec throughput.

Table 1 contains the system settings used for experiments.

4.2 Workload Model

The workload model is homogeneous, i.e. it consists of queries of one type. As was mentioned above, the complex query is executed in a number of steps. We evaluate the performance for the query which has two execution steps.

1. Several regions are scanned. After the scan results are send to the client (query initiator machine).
2. Some of the objects obtained on the first step are updated. Updates take place in the same regions, and thus are performed by the same region servers.

The number of nodes participating in query execution is distributed according to Poisson distribution with mean M. The number of data objects query reads per region server is exponentially distributed with the mean N_r, writes count is distributed in a similar way with the mean N_w. Reads and writes are distributed among the nodes uniformly.

The 99%-1% data access pattern is used. 99% of transactions references 1% of data objects. In our model the data access pattern influences both the buffer pool hit rate and the data contention.

For the model simplicity we assumed that the data object version has a page size. It is reasonable assumption, because HBase was not designated to work with objects of small size. Thus, the granule for CC algorithm is represented by a page in our model.

Workload parameters can be found in the Table 2.

4.3 Query Execution Model

Each query has a home node. The home node is the client machine it has arrived to. After a query has arrived, it is

Parameters	Settings
M	7
N_r	1000
N_w	10
N_{vc}	10
data access pattern	99%-1%
percent of sequential IO	5%

Table 2: Workload parameter settings.

split to a number of subqueries, each of which is local to one of execution nodes. Each subquery is modeled as a list of data references (pages) to be processed. For each subquery the message is sent to the execution site to start up the execution. Each query joins the Net queue and consumes CPU time to send a message to execution sites.

When the subquery start message arrives to the execution site, the subquery is created and it joins the Net queue on the execution site and consumes CPU time needed to read a message. Than it consumes CPU to begin execution.

The subquery execution consists of Begin-Of-Execution (BOE) step, a number of data reference processing iterations and End-Of-Execution (EOE) step.

For each data object reference it goes to the buffer pool to get the data. If the required page is not already in memory, subquery consumes processor time to start IO and joins the disk queue. Than the subquery goes to CPU queue and consumes time to process data reference (to find appropriate version). In case there are some more references the subquery repeats a similar data reference processing iterations.

After the subquery has successfully finished its execution it sends requested data (if present) and the READY message to the home node. After that, the subquery is finished, so it goes to CPU queue and consumes time for EOE step.

The client receives the data from all the participated servers, process it and sends a message to each participant to perform updates of the second execution stage.

A write subquery is executed in the similar way with a read subquery. Its behaviour differs in data processing iteration. The write subquery firstly consumes CPU for reference processing and after that writes the log page to the file system and enqueues the page write (it does not wait until the page will be written).

4.4 Transaction Execution Model

Read subtransactions do not differ from read subqueries in their execution model. Although they may consume more CPU time for a data reference processing to find appropriate version, the performed steps are same.

Write subtransaction performs version check before each write. If the version check failed, the transaction is to be aborted, and subtransaction goes to the Net node to send an ABORT message. The probability of version check success will be discussed later.

The probability that versions needed for version check were not flushed to a file since the transaction had started is quite high for not write intensive workload. Therefore, the version check step is modeled as processing of N_{vc} pages, each of which is already in buffer pool, so disk IO is not needed.

After the write subtransaction has finished execution it sends READY to the home node and waits for COMMIT or ABORT message in reply. The transaction initiator waits until all READY messages (or one ABORT message) will be received. Than it sends the COMMIT (ABORT) message to all executors and the transaction is finished. After the subquery received the COMMIT (ABORT) message, it writes a log page and goes to the CPU queue to perform the EOE step.

4.5 Abort Probability

Let us consider the probability of version check failure. Suppose the transaction t which was started at the moment u_0 is going to update the value of data object x at the moment u.

The failure of version check may be caused by two types of transactions that had updated x recently.

1. x may been written by the transaction which was committed after t had started. Given the throughput of the server $T(u)$ for the moment u, the number of transactions committed in the time interval $[u_0, u]$ can be approximated by $N_w * T(u) * (u - u_0)$.

2. x may been written by the transaction, that has not already committed. Let $A(u)$ denote the number of write subtransaction active at the server as for the moment u. The average number of writes made by active subtransactions may be approximated by $N_w/2$. The number of writes made by all active transaction as for a moment u may be estimated with $N_w * A(u)/2$.

The count of updates written by transactions which execution interval overlaps with execution interval of t may be approximated by

$$K(u, u_0) = N_w * \frac{A(u) + 2 * T(u) * (u - u_0)}{2}$$

Let us consider the a-b data access pattern and the case when t belongs to majority of transactions. The number of updates made by transactions of major class is $a * K(t, u)$, while the number of data objects t can access is $D * b$. The probability of observing update made by a transaction of the same class is

$$\frac{a * K(u, u_0)}{D * b}$$

The probability of observing updates made by transactions of the other class is

$$\frac{(1 - a) * K(u, u_0)}{D}$$

Thus, for a transaction of the first type the probability of version check failure can be estimated by

$$P_1(u, u_0) = \frac{b * (1 - a) + a}{b} * \frac{K(u, u_0)}{D}$$

and for a transaction of the second type by

$$P_2(u, u_0) = \frac{K(u, u_0)}{D}$$

5 Simulation Results

The main goal of our performance study consists in evaluation of performance degradation caused by additional overheads introduced by the protocol. We measured the performance of the protocol in terms of the system throughput and average response time.

We evaluated the throughput degradation for a particular parameter settings as the difference between the EC and SI throughputs divided by EC throughput.

In this section we use the abbreviation EC to denote HBase version, which uses eventual consistency, and SI for HBase, which uses our protocol. For most of the experiments we present two diagrams: one depicts normal workload results, while the other reflects the situation when all the available resources are already saturated. We have payed a special attention to the saturated system performance to examine the influence of transaction aborts on the throughput.

Fig. 1 and 2 plot throughput results against the active queries count in the system. The maximum degradation obtained for normal workload was 3.3% and 7.7% for saturated system. In the latter case transaction aborts make significant contribution to the throughput degradation. As the number of queries in the system increase, the throughput degradation raises to the mentioned maximum of 7.7% for 1400 queries. When the number of queries is equal to 100, aborts number per second is less than 30% of difference between EC and SI throughputs. The rate of aborts count per second to difference of throughputs reaches the maximum of 98.8% at the point of 700 queries.

Fig. 3 and 4 give the average response time against the active queries count in the system. Relative response time increase is quite small, it does not exceed 3%. After the queries count reaches 35 the relative response time increase does not change significantly. It remains about 3% for the highly saturated workload too.

As expected the use of distributed commit protocol does not make considerable contribution to the average response time for the long-running transactions. The disk system appears to be the bottleneck for this kind of workload. The performance of the system is substantially influenced by the buffer pool hit rate.

Fig. 5 and 6 shows the throughput results for EC and SI against buffer pool hit rate. These experiments were conducted with the systems running 70 (Fig. 5) and 400 (Fig. 6) transactions. The partition size was varied to provide needed buffer pool hit rate. The results obtained from both experiments are similar. In both experiments systems show significant speedup with buffer pool hit rate verging towards 1. EC outperforms SI by 0.51-6.3% in case of normal workload and by 2.9-6.7% otherwise.

Fig. 7 gives response time against buffer pool hit rate for the system running 70 queries. Response time is not as sensible to buffer pool hit rate as throughput. The graph is close to linear. However, relative response time increase grows rapidly with buffer pool hit rate getting closer to 1 and reaches the maximum of 6.25% for normal workload and 3.87% for saturated workload.

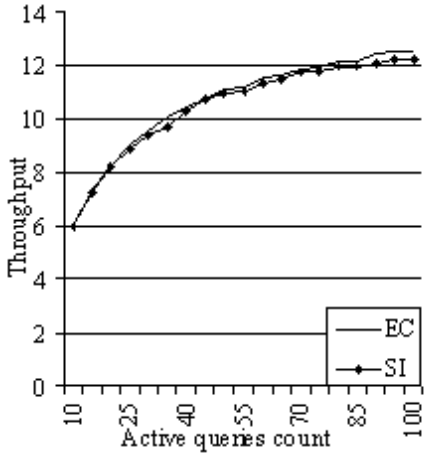

Figure 1: Normal workload throughput against the active queries count.

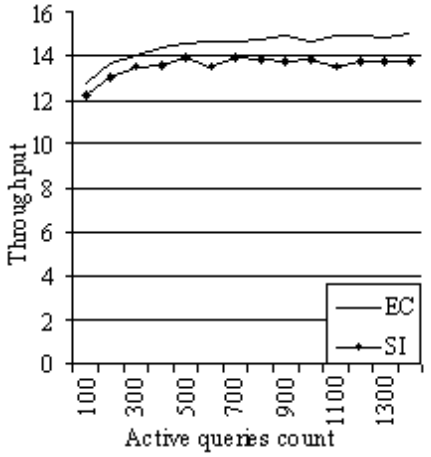

Figure 2: Saturated workload throughput against the active queries count.

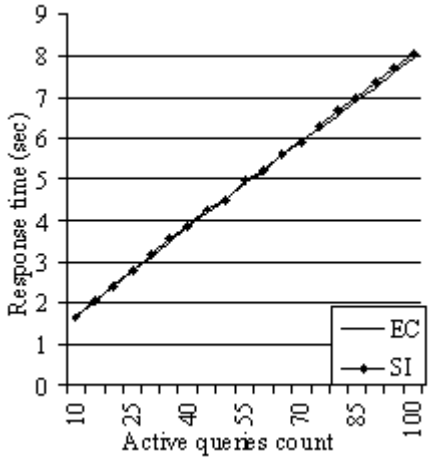

Figure 3: Normal workload response time against the active queries count.

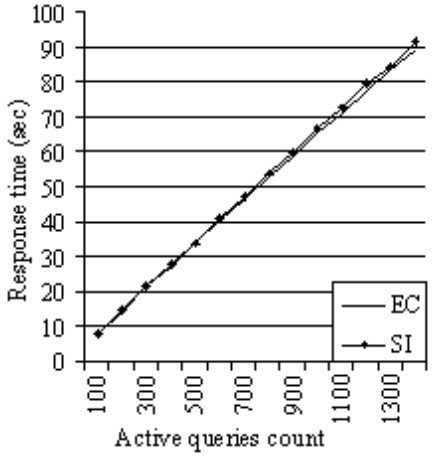

Figure 4: Saturated workload response time against the active queries count.

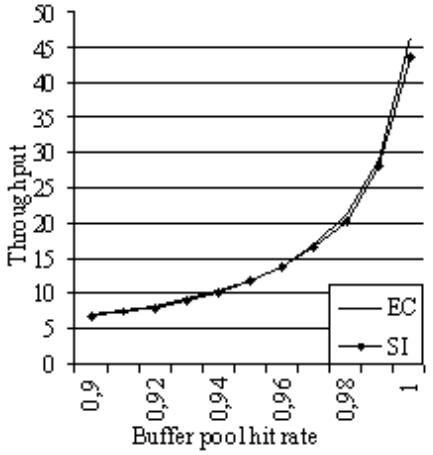

Figure 5: Normal workload throughput against buffer pool hit rate.

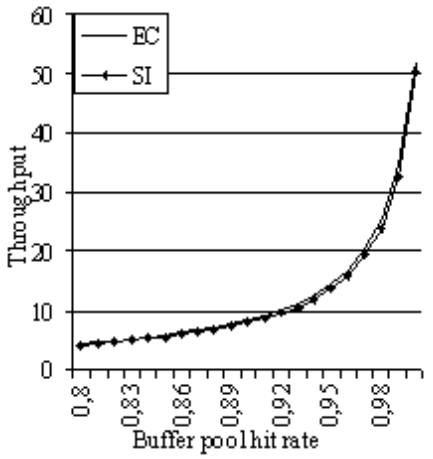

Figure 6: Saturated workload throughput against buffer pool hit rate.

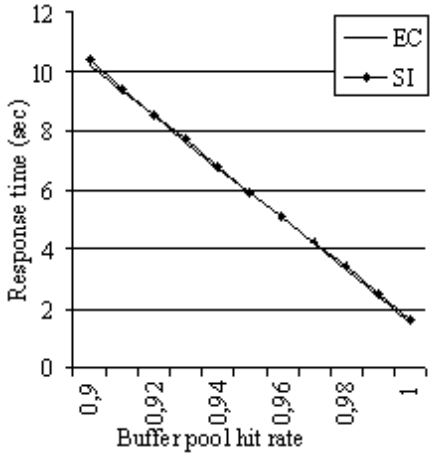

Figure 7: Normal workload response time against buffer pool hit rate.

6 Model Validation

We have implemented the protocol in HBase. Initially we have evaluated the protocol performance using the prototype. We have measured the prototype performance for a workload, which consisted of short-running queries. Each query read and updated 3 values of `double` type. The prototype has shown relatively good results for such workload: the throughput was 3 times lower than that of initial HBase version.

To validate our model mentioned test set was changed. If each query writes 3 `double` values, size of log entries to be flushed to disk before query finishes is small. So we replaced `double` values with 4Kb sized byte arrays. After the changes had been made we discovered HBase throughput being ten times greater than that of the prototype.

We suppose the prototype implementation showed such a poor performance, because it copies data objects for internal purposes. For every single `get` and `put` operation at least one data object is copied. We suppose that despite the fact that main memory reads and writes are extremely fast, copying of data objects may affect the performance in case of no disk accesses.

The values of model parameters for EC were selected in such a way, that EC throughput was as close as possible to HBase throughput. The experiment conducted with the same parameters using SI model has shown the throughput 1.5 times lower than for EC (1200 vs. 1800).

To have the SI model performance close to that of the prototype, we adjusted the mean page processing time parameter to be 270000 instruction instead of default 25000. The primary goal of this experiment was to validate the abort probability model. Fig. 8 shows the prototype throughput and SI model throughput, and 9 shows abort rates obtained. The difference between the aborts rates obtained from the model and the prototype has not exceeded 3% being about 2% at average. We conclude that our model captures the system behaviour adequate.

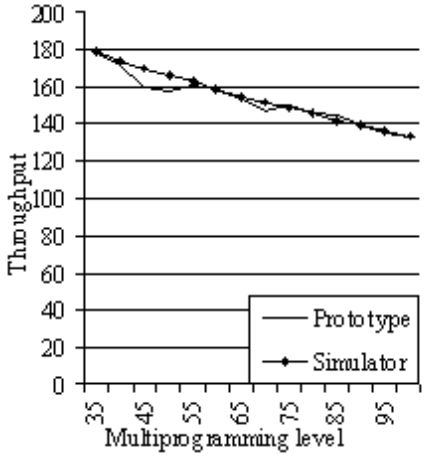

Figure 8: Model validation: SI throughput.

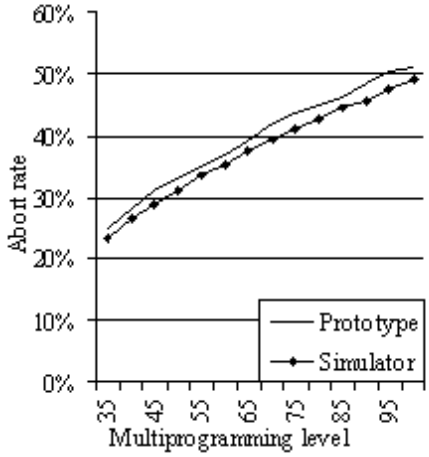

Figure 9: Model validation: SI abort rate.

7 Summary

In this work we presented SI protocol and evaluated its performance. The advantage of the protocol under EC is obvious: the data storage user is provided with SI consistency guarantees. However, the protocol introduces several additional overheads, which may affect the performance. We have modeled the performance for the particular kind of workload and validated the model using the prototype we have presented in [12]. Although, the protocol implementation prototype has shown quite poor performance, it does not imply that the protocol is ineffective itself. The simulation results has shown that use of the protocol does not lead to significant performance degradation for this workload type.

References

[1] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, 1987.

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, volume 24, pages 1–10, New York, 1995.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, volume 7, pages 295–310, 2006.

[4] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with Snapshot Isolation. In *Proceedings of the 32nd international conference on Very Large Databases*, 2006.

[5] Robert Marek and Erhard Rahm. Performance evaluation of parallel transaction processing in Shared Nothing database systems. In *Proceedings of the 4th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 295–310, Paris, 1992.

[6] Matthias Nicola and Matthias Jarke. Performance modeling of distributed and replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 12(4), 2000.

[7] Erhard Rahm. Parallel query processing in Shared Disk database systems. In *Proceedings of 5th Int. Workshop on High Performance Transaction Systems*, Asilomar, 1993.

[8] Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia Sapienza. A performance model of multi-version concurrency control. In *Proceedings of IEEE International Symposium on Modeling, Analysis and Simulation of Computers*

and Telecommunication Systems, pages 1–10, 2008.

[9] Naval Postgraduate School, March 2010. Simkit.

[10] Y.C. Tay. Issues in modelling locking performance. In Hideaki Takagi, editor, *Stochastic Analysis of Computer and Communication*. Elsevier Science Publishers B.V. (North-Holland), New York, 1990.

[11] Alexander Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Computing Surveys*, 30(1), 1998.

[12] Dmitri Vasilik. Implementing Snapshot Isolation in HBase. Diploma thesis, Saint Petersburg State University, 2010.

[13] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. Towards elastic transactional cloud storage with range query support. *Proceedings of the VLDB Endowment*, 3(1-2), 2010.

[14] Chen Zhang and Hans De Sterck. Supporting multi-row distributed transactions with Global Snapshot Isolation using bare-bones HBase. In *Proceedings of the 11th ACM/IEEE International Conference on Grid Computing (Grid 2010)*, pages 295–310, Brussels, 2010.