Exploiting Semantics and Speculation for Improving the Performance of Read-only Transactions

T. Ragunathan and P. Krishna Reddy

International Institute of Information Technology,
Hyderabad
India
ragunathan@students.iiit.ac.in, pkreddy@iiit.ac.in

Abstract

A read-only transaction (ROT) does not modify any data. Efforts are being made in the literature to improve the performance of ROTs without correctness and data currency issues. The widely used two-phase locking protocol (2PL) processes the transactions without any correctness and data currency issues. However, the performance of 2PL deteriorates with data contention. Snapshot isolation (SI)-based protocols proposed in the literature improve the performance of ROTs, but they compromise on correctness and data currency issues. Speculative locking (SL) protocols are proposed in the literature for improving the performance of ROTs by carrying out speculative executions only for ROTs and following 2PL for update transactions. In SL-based protocols, update transactions are blocked if they conflicting with ROTs. In this paper, we have proposed an improved approach to improve parallelism among update transactions and ROTs by exploiting a new notion called "compensatability". In this protocol, an ROT which can be "compensatable" can complete the execution and carry out compensating operation to incorporate the effect of conflicting update transactions. As a result, the parallelism is improved over SL protocols as the update transactions which are conflicting with 'compensatable' ROTs need not block. In this paper, we have proposed a protocol by exploiting both "compensatability" property of ROTs and speculation. The simulation results show that the proposed protocol improves the performance by carrying out less number of speculative executions. Further, the proposed protocol does not violate serializability criteria.

1 Introduction

Processing read-only transactions (ROTs) with high performance and without any correctness and data currency issues is an important problem in web-based information systems. A read-only transaction (ROT) does not modify any data. The main issues in processing ROTs are correctness (serializability), data currency and performance. The widely used two-phase locking (2PL) protocol [30] [14] processes the transactions with serializability as correctness criteria. 2PL performs poorly, as both ROTs and update transactions (UTs) are made to wait whenever conflicts occur. Snapshot isolation (SI)-based methods [3] are widely used to process ROTs. Note that, ROTs processed at SI violate serializability criteria [9]. A new mechanism is discussed in [9] which requires the analysis of transaction programs, without requiring any modification of the database management engine, to make SI serializable. In [27], automating the task of modifying the program logic is discussed.

Data currency refers to how current or up-to-date we can guarantee a data object to be, for a transaction. The definition for data currency in a data warehousing environment and in a replicated environment are discussed in [24] [16] respectively. We can define data currency for a general DBMS environment as follows. Let T_i and 't' denote a transaction and time duration, respectively. The data currency of the data object provided to T_i is the value of 't' which is the time difference between the commit time of the transaction which created the latest version of the data object and the commit time of the transaction which created the version of that data object that was read by T_i. If 't' is less/more, it means that transactions are provided with high/low data currency. It can be noted that 2PL provides high data currency and SI-based protocols provide low data currency for ROTs.

In the protocol proposed in [5], the execution of ROTs is completely independent of the underlying concurrency control and replica control mechanisms. An approach has been proposed in [11] for distributed environment in which ROTs are processed with a special algorithm that is different from the one used for update transactions (UTs). In [26], an approach has

International Conference on Management of Data
COMAD 2008, Mumbai, India, December 17–19, 2008
©*Computer Society of India, 2008*

been discussed for maintaining multiple versions of data objects. In this technique, based on the arrival time, ROTs read particular versions of the data.

In [21], speculation has been extended to improve the performance of distributed database systems by considering transactions which contain both the read and writes operations. Based on speculative locking protocol, two protocols have been proposed namely synchronous speculative locking protocol for ROTs [18] and asynchronous speculative locking protocol for ROTs [20]. In these approaches, ROTs are processed with speculation and UTs are processed using 2PL. The speculation-based approaches which require extra processing resources, improve the performance of the ROTs without any data currency and correctness issues.

In [13],the use of commutative property of arithmetic operations plus and minus is briefly discussed to reduce the lock contention for data objects. The use of commutative property of operations in increasing concurrency is also discussed in [28] [31] [32]. In [10], how commutative steps of distributed transactions can be interleaved to improve the performance is discussed. A multi-copy algorithm for replicated environment is discussed in [22], which exploits the application semantics. In [25], the authors identified a property known as recoverability which can be used to decrease the delay involved in processing non-commuting operations while still avoiding cascading aborts. In [15], the authors presented a locking protocol for object databases which uses method commutativity. A semantic locking protocol which supports referentially shared objects was proposed in [4]. Recently an experimental study [6] has been performed on an object-based industrial application used in telecommunications sector. This study shows that there is a potential in real-world applications for improving the performance through semantic-based approaches.

In this paper, we propose an approach which exploits the semantics of the applications and speculation to process the ROTs. We have identified that an ROT which performs compensatable computations can be executed without speculation and without waiting for the conflicting UTs. Also, we have observed that the UTs conflicting with such ROTs can be executed without blocking. Based on these observations, we have come up with a notion called "compensatability" for classifying the ROTs. Also, we have proposed a speculation-based approach which can process the ROTs effectively with less number of speculative executions and process the UTs satisfying certain conditions without blocking. The proposed protocol does not have any correctness and data currency problems. The simulation results show that the proposed protocol performs marginally better than SSLR and requires less number of speculative executions as compared to SSLR. Also, the UT throughput of the proposed protocol is better than 2PL, SI-based and SSLR protocols.

1.1 System Model and Notations

A database is a collection of data objects. Users interact with the database by invoking transactions. Transactions are represented with T_i, T_j,... A transaction is a sequence of read and write operations that are executed atomically on the data objects. A transaction can read a set of data objects from the database which forms the read set (RS) of the transaction and modify the values of another set of data objects which forms the write set (WS) of the transaction. An ROT contains only read operations. A UT consists of both read and write operations. The transactions T_i and T_j are said to have a conflict, if $\mathrm{RS}(T_i) \cap \mathrm{WS}(T_j) \neq \emptyset$, or $\mathrm{WS}(T_i) \cap \mathrm{RS}(T_j) \neq \emptyset$ or $\mathrm{WS}(T_i) \cap \mathrm{WS}(T_j) \neq \emptyset$. The execution of a transaction must be atomic [30] [14]; i.e., a transaction either commits or aborts. The commit of a transaction results in all of its changes being applied to the database, whereas the abort results in the changes being discarded.

The database management system consists of modules like transaction manager and a data manager [12]. Processing of transactions is managed by the transaction manager component of database management systems, while database is managed by the data manager.

Data objects are denoted with 'x','y',.. For the data object 'x', 'x_i' (i = 0 to n) represents i_{th} version of 'x'. The notation $\mathrm{r}_i[\mathrm{x}_j]$ indicates that read operation is executed on 'x_j' by the transaction T_i and $\mathrm{w}_i[\mathrm{x}_j]$ denotes that the transaction T_i performs a write operation on a particular version of 'x' and produces 'x_j'. The notations 's_i', 'c_i', and 'a_i' denote the start, commit and abort of T_i respectively. T_{ij} indicates j_{th} speculative execution of T_i.

1.2 Paper Organization

The rest of the paper is organized as follows. In Section 2, we discuss 2PL, SI-based, speculative locking and SSLR protocols. In Section 3, we present the basic idea of the proposed protocol. Next, we present the overview of the protocol. Subsequently, we discuss the details of the protocol. In Section 4, we present the simulation results. We discuss the implementation and performance issues in Section 5. The last section contains conclusions and future work.

2 Concurrency control protocols

In this section, we discuss two-phase locking, snapshot isolation-based, basic speculative locking and SSLR protocols.

2.1 Two-phase locking

Under 2PL [30], a transaction requests "read-lock" (R-lock) to read an object and a "write-lock" (R-lock) to

Lock requested by T_j	**Lock held by T_i**	
	R	**W**
R	yes	no
W	no	no

Figure 1: Lock compatibility matrix for 2PL

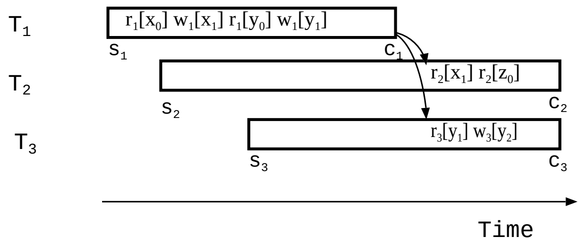


Figure 2: Depiction of transaction processing with 2PL

write/update the data object. In 2PL, a transaction should obtain all the required locks before performing any unlock operation. We have considered a variation of 2PL called "strict two-phase locking protocol" [12]. The lock compatibility matrix for 2PL is shown in Figure 1. The terms "yes" and "no" in the matrix means that the corresponding lock requests are compatible and incompatible, respectively. On any data object, the R-lock of the lock requesting transaction is compatible with the R-lock of the lock-holding transaction. If one of them is W-lock, the locks are not compatible.

We explain the processing of ROTs under 2PL with an example. Consider Figure 2. In this, both T_1 and T_3 are UTs and T_2 is an ROT. It can be observed that T_2 has to wait for a lock on the object 'x' until T_1 commits. Similarly, T_3 has to wait for a lock on 'y'. (The space between the last operation and 'c_i' notation in the transaction diagram depicts the time required to carry out logging and commit operations. Let 't_1' and 't_2' are time instances. The arrow mark between 't_1' and 't_2' indicates that the action at 't_2' starts only after the action at 't_1'). Due to waiting the performance of ROT suffers in 2PL as data contention increases.

2.2 Snapshot isolation-based protocol

A new isolation level called snapshot isolation (SI) is proposed in [3]. In SI-based techniques, an ROT (T_i) reads data from the snapshot of the (committed) data available when T_i has started or generated the first read operation. The modifications performed by other concurrent UTs, which have started their execution after T_i are unavailable to T_i. We consider one of the SI-based protocol "first committer wins rule"(FCWR) [3]. In FCWR, a transaction (T_i) commits if and only if no concurrent transaction (T_j) has already committed writes of data objects that T_i intends to write.

It can be noted that, SI-based protocols are not serializable [9] as an ROT ignores the updates of concurrent UTs which have committed after its start.

The processing of ROTs using FCWR is depicted in

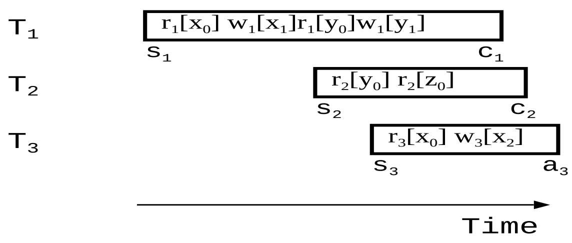


Figure 3: Depiction of transaction processing with FCWR

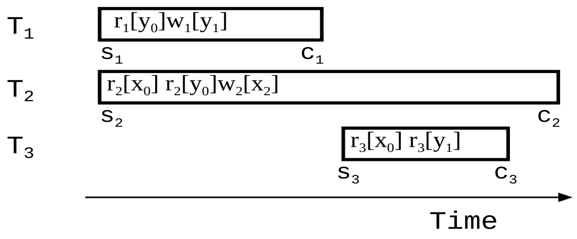


Figure 4: Serilizability violation under FCWR protocol

Figure 3. In this figure, both T_1 and T_3 are UTs, and T_2 is an ROT. It can be observed that T_2 reads the currently available values 'y_0' and 'z_0' and proceeds with the execution. Simultaneously, T_3 also reads 'x_0' and produces 'x_2'. Note that, FCWR allows only one of the conflicting UTs to commit. So, T_3 has to be aborted as T_1 commits. However, as per FCWR, T_2 commits with the old values and it has not accessed the updates produced by T_1 even though T_1 commits before its completion and therefore receives low data currency.

Figure 4 shows an example for serializability violation under FCWR protocol. In this figure, both T_1 and T_2 are UTs, and T_3 is an ROT. It can be observed that T_3 reads the currently available values 'x_0' and 'y_1' and proceeds with the execution. As T_1 and T_2 are not in conflict, they proceed their executions as per FCWR. We can observe that the execution of the transactions T_1, T_2 and T_3 is not equivalent to a serial order of executions and hence violates the serializability criteria. Note that, the execution of UTs T_1 and T_2 do not violate serializability criteria, But, the introduction of the ROT T_3, makes the execution of transactions T_1, T_2 and T_3 to violate serializability criteria [9].

2.3 Speculative locking protocol

In the speculative locking (SL) protocol [21], it was assumed that a transaction produces after-image whenever it completes the work with that object. By accessing before- and after-images of conflicting active transactions, the waiting transaction carries out multiple speculative executions. In SL, a transaction commits only after the termination of preceding transactions with which it has formed commit dependencies. The SL approach improves the transaction processing

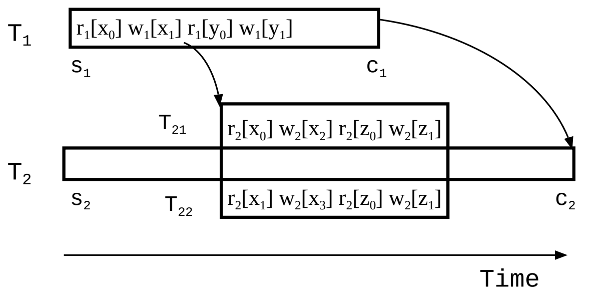

T1
r1[x0] w1[x1] r1[y0] w1[y1]
S1
C1
T21
r2[x0] w2[x2] r2[z0] w2[z1]
T2
S2
T22
r2[x1] w2[x3] r2[z0] w2[z1]
C2
Time

Figure 5: Depiction of transaction processing with SL

Lock requested by T_j	**Lock held by T_i**		
	R	**EW**	**SPW**
R	yes	no	sp_yes
EW	sp_yes	no	sp_yes

Figure 6: Lock compatibility matrix for SL

performance by increasing the parallelism and reducing waiting.

Figure 5 depicts the processing of transactions with SL. T_{ij} indicates j_{th} ($j > 0$) speculative execution of T_i. It can be observed that T_2 starts speculative executions T_{21} and T_{22}, once T_1 produces the after-image 'x_1'. T_2 accesses both 'x_0' and 'x_1' and starts speculative executions. Here T_2 forms commit a dependency with T_1. If T_1 commits, T_2 commits by retaining the execution T_{22}. Otherwise, if T_1 aborts, T_2 commits by retaining T_{21}. If these transactions would have processed under 2PL, T_2 can obtain lock only after the termination of T_1. So, it can be observed that SL improves parallelism by reducing lock waiting time.

Lock compatibility matrix of SL is shown in Figure 6. In SL, W-lock of 2PL is partitioned into two locks: exclusive write (EW)-lock and speculative write (SPW)-lock. Transactions request R-lock for read and EW-lock for write. When a transaction produces after-image for a data object, the EW-lock is converted into SPW-lock. Under SL, only one transaction holds an EW-lock on a data object at any time. However, note that multiple transactions can hold the R- and SPW-locks on a data object at the same time. The entry "sp_yes" indicates that the requesting transaction carries out speculative executions and forms commit dependency with the preceding transactions that hold R/SPW-locks. Note that the requesting transaction

Lock requested by T_j	**Lock held by T_i**			
	RR	**RU**	**EW**	**SPW**
RR	yes	yes	no	ssp_yes
RU	yes	yes	no	no
EW	no	no	no	no

Figure 7: Lock compatibility matrix for SSLR

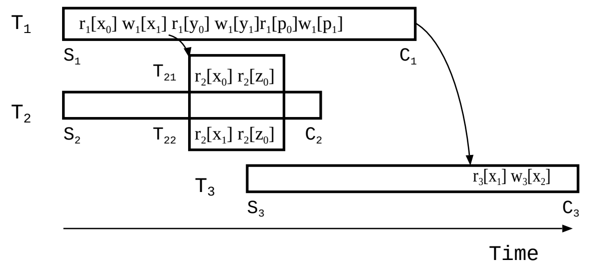


Figure 8: Depiction of transaction processing with SSLR

commits only after the termination of preceding transactions with which it has formed commit dependencies. SL protocol produces serializable executions.

2.4 The SSLR protocol

The SL protocol was proposed by considering UTs; i.e., the transactions that contain both read and write operations. Two speculation-based protocols are proposed in the literature for improving the performance of ROTs [18] [19] [20] [17]. One is synchronous speculative locking protocol (SSLR) and the other one is asynchronous speculative locking protocol (ASLR). These protocols process the ROTs using speculation and UTs with 2PL. In this paper, we consider SSLR protocol for performance comparison purpose.

The lock compatibility matrix of SSLR is shown in Figure 7. The W-lock is divided into EW-lock and SPW-lock. The EW-lock is requested by UTs for writing the data object. The RU-lock (Read lock for UT) is requested by UTs for reading a data object. The RR-lock (read lock for ROT) is requested by ROT for reading a data object. The entry "ssp_yes" (synchronous speculation yes) indicates that the requesting ROT carries out speculative executions and forms a commit dependency. Note that, the notion of commit dependency in SSLR is different as compared to the notion of commit dependency in SL. In SSLR, if T_i carries out speculative executions and forms a commit dependency with T_j, T_i commits whenever it is completed (say at time 't') by retaining appropriate speculative execution based on the termination status of T_j at time 't'. That is, even though T_j is not completed, T_i can retain one of the speculative execution and proceed to commit whenever T_i completes execution. This is possible, only if T_j is ROT. Whereas in SL, T_i has to wait for T_j's termination as SL is proposed for UTs.

The processing of ROTs under SSLR is illustrated in Figure 8. Here, T_1 and T_3 are UTs which are processed with 2PL and T_2 is an ROT which is processed with SSLR. T_1 obtains EW-lock on data object 'x'. It reads 'x_0' and produces 'x_1' and converts the EW-lock on 'x' to SPW-lock. T_3, being a UT, waits till T_1 releases the lock on 'x'. The ROT T_2 is processed as follows. Note that even though both T_1 and T_2 have arrived at the same instant, T_2 waits till T_1 produces after-image 'x_1', T_2 carries out two executions T_{21} and T_{22} by accessing 'x_0' and 'x_1' respectively. Note that, T_{21} and T_{22} are carried out synchronously. After T_2's completion, T_{21} is retained even though T_1 is not yet committed. We can observe that, T_2 is committed without waiting for the termination of T_1. Also, the transactions are serialized as per the order $T_2 \ll T_1 \ll T_3$.

3 The proposed protocol for ROTs

In this section, first we discuss the basic idea of proposed protocol for ROTs. Subsequently, we present the overview of the protocol. Next, the protocol is presented in detail.

3.1 Basic idea

It can be observed that the SSLR protocol improves the performance of ROTs and blocks conflicting UTs. Because, both ROT and UT are executed in parallel and access the same data objects, the correctness may be violated. However, in case of ROTs, there is an opportunity to improve parallelism, if it is possible to compensate the computation that is being missed by the ROTs.

So, the basic idea of the proposed protocol is as follows. The UTs are processed in parallel with ROTs without any blocking. However, the ROTs should compensate the computation. If we identify the ROTs which are "compensatable", by modifying the transaction code, the parallelism can be improved by allowing the execution of UTs in parallel with conflicting ROTs. The property of ROTs which allow compensation, is called "compensatability", which is defined as follows.

Definition: ***Compensatability:*** Let T_i be an ROT and T_j be a UT. Consider that T_i accesses data object 'x' at the time instant 't_s' and produces new data object 'y' at the time instant t_e ($t_e > t_s$). In parallel, T_j accesses 'x' and produces 'x'' at time instant t_u ($t_s < t_u < t_e$). Here 'x'' is the value of 'x' modified by T_j. Consider that if T_i would have accessed 'x'', it would have produced new data object 'z'. We say, the computation by T_i on 'x' is "compensatable", if there exists a function or computation 'g' such that $z=g(x')$.

Suppose, T_i accesses 'n' data objects. The computation carried out by T_i is "compensatable", if the computation carried out by T_i is "compensatable" for each conflicting data object. The problem is to find the function 'g' for each data object accessed by ROT. It may be difficult for UTs. However, it is not that difficult for ROTs as they do not modify any data objects. In addition, for those ROTs which produce arithmetic results such as SUM, AVERAGE, PERCENTAGE and so on, we consider that it is easy to find the function 'g' and improve the parallelism.

We now explain the notion of "compensatability" with an example.

Let T_1 be an ROT and T_2 be a UT. Let 'x', 'y' and 'z' be the integer data objects. T_1 and T_2 are defined as given below.

T_1: r[x], r[y], $z = x + y$, d[z], commit.
T_2: r[y], $y = y+10$, w[y'], commit.

Here T_1 is an ROT and T_2 is a UT. Both T_1 and T_2 are concurrent transactions. We can assume that, while T_1 is performing computation ($z = x + y$), T_2 modified the 'y' value to 'y+10' (y'). In T_1, d[z] refers to displaying the value of 'z' to the terminal. Note that, the computation performed by T_1 (addition) satisfies the "compensatability" property. Here the function 'g' can be expressed as "$z+ (y'-y)$". The transaction T_1 has to perform the compensating function 'g', after its completion, if T_2 completes first. Otherwise, there is no need for the compensating computations.

So, if the ROTs are performing computations of "compensatable" type and if they conflict with UTs, then the ROTs and UTs can be processed in parallel without blocking, which results in increased performance. However, before committing, the ROTs have to perform compensating computations by reading the updated data which are produced by the conflicting committed UTs.

3.2 Overview of the proposed protocol

The proposed protocol exploits the "compensatability" property of the operations used in the applications for improving the performance of ROTs. In this approach, based on the "compensatability" property of the operations performed, we classify the ROTs into two types namely compensatable ROTs (CROTs) and non-compensatable (NCROTs). If all computing operations of an ROT satisfy the "compensatability" property, we call that ROT as CROT. Otherwise, we call the ROT as NCROT. We allow CROTs to execute without blocking and NCROTs to follow synchronous speculation as per SSLR. Note that, CROTs do not perform speculative executions, but they have to perform compensating computations during commit time. 2PL is chosen to process the UTs. However, the UTs conflicting with CROTs are processed without blocking. We call the proposed protocol as SSLR-S (synchronous speculative locking protocol for ROTs - which exploits semantics).

Compensating operations:

In the proposed protocol, CROTs are processed without blocking. However, when a UT conflicts with a CROT or a CROT conflicts with a UT, the transaction identification number of the UT and the identification number of the data object which is modified

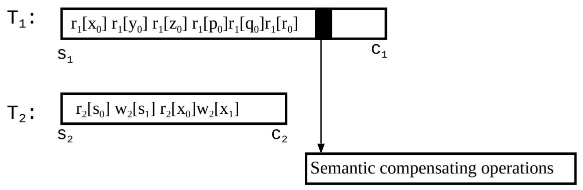
Lock requested	**Lock held by T_i**				
by T_j	**CR**	**NR**	**RU**	**EW**	**SPW**
CR	yes	yes	yes	sm_yes	sm_yes
NR	yes	yes	yes	no	sp_yes
RU	yes	yes	yes	no	no
EW	yes	no	no	no	no

Figure 9: Lock compatibility matrix for SSLR-S

by that UT, are recorded in a list. During its commit time, a CROT has to read the identification numbers of the conflicting UTs and data objects from this list and search for the same in the transaction log. The transaction log is searched in the reverse order by the CROTs. This is similar to the approach followed in [11]. After reading the up-to-date values of the data objects from the transaction log, the CROT can perform the compensating computations as per the procedure available in the transaction program of that CROT. The procedure to perform compensating computations has to be developed by the database programmers. We believe that only few lines of code have to be added to the transaction program for performing compensating computations. The software routine for searching the transaction log has to be available in the transaction manager for all CROTs.

Types of lock:

The CROTs request compensating read locks (CR-locks) for reading. The NCROTs request non-compensating read locks (NR-locks) for reading. The UTs request read update locks (RU-locks) for reading and exclusive write locks (EW-locks) for writing.



T_1:
$r_1[x_0]$ $r_1[y_0]$ $r_1[z_0]$ $r_1[p_0]r_1[q_0]r_1[r_0]$
s_1
c_1
T_2:
$r_2[s_0]$ $w_2[s_1]$ $r_2[x_0]w_2[x_1]$
s_2
c_2
Semantic compensating operations

Figure 10: Depiction of transaction processing with SSLR-S

The lock compatibility matrix of SSLR-S is shown in Figure 9. The entry "yes" indicates that the corresponding locks are compatible and "no" indicates that the corresponding locks are incompatible. The entry "sm_yes" (semantic yes) indicates that the requesting CROT is allowed to continue the execution. Note that, the UTs conflicting with CROTs are allowed to continue without blocking, which is different from the 2PL procedure. The entry "sp_yes" (speculation yes) indicates that the requesting NCROT carries out speculative executions with the after-image produced by the preceding UT and forms a commit dependency with that UT.

Transaction processing in SSLR-S:

Figure 10 depicts the processing under SSLR-S. Here T_1 is a CROT and T_2 is a UT. Note that, here T_2 accesses 'x_0' on which T_1 is already having CR-lock. This creates read-write conflict and as per SSLR-S rule T_2 is allowed to acquire EW-lock and to continue its execution. When T_1 completes its execution, it reads the modified data from the transaction log and performs the compensating operations as per the procedure given in its transaction program. Note that, this type of processing does not violate the serializability criteria.

3.3 SSLR-S protocol

For each data object, a *lockqueue* is maintained to store the lock requests. The CROTs use the transaction log to read the updated values produced by the conflicting UTs. The list *dependset*(T_{ij}) stores the commit dependency details of the j_{th} speculative execution of T_i. This list is maintained for each speculative execution of NCROTs. The list *dependset*(T_i) stores the details of the UTs with which the CROT T_i had conflicts, during its execution. This list is used by the CROTs for identifying conflicting UTs while performing compensating computations.

Protocol for UTs.

1. *Lock acquisition.* Let T_i be a UT and requests for RU-lock to read 'x' or EW-lock to write 'x'. The lock request is entered into the lockqueue.

    1.1 T_i obtains RU-lock if no transaction holds EW-lock or SPW-lock. Step (2) is followed.

    1.2 T_i obtains EW-lock on 'x', if no transaction holds RU-, NR-, EW-, and SPW-locks.

2. *Execution.* During execution, whenever T_i produces the after-image for a data object, EW-lock on the data object is converted into SPW-lock. If T_i obtains all the locks, step (3) is followed. Otherwise step (1) is followed.

3. *Commit/Abort Rule.* Whenever T_i commits, the speculative executions of NCROTs that have been carried out with before-images of T_i are terminated. Whenever T_i aborts, the speculative executions of NCROTs which have been carried out

with after-images of T_i are terminated. Commit status of T_i is added into the *dependset* of CROTs which are having dependency with T_i. Whenever T_i commits or aborts, the information regarding T_i is deleted from the *dependset* maintained for each of the speculative execution of NCROTs which are dependent on T_i. Also, all the related lock entries of T_i are deleted.

Protocol for CROTs:

4. *Lock acquisition.* Let T_j be CROT and requests for CR-lock to read 'x'. The lock request is allocated to T_j. The details of the preceding conflicting UTs are added to *dependset*(T_j).

5. *Execution.* T_j continues the execution by accessing 'x'. If T_j obtains all the locks then step (6) is followed. Otherwise, step (4) is followed.

6. *Commit/Abort Rule.* Whenever T_j commits, necessary compensating computations specified in its transaction program are performed using updated values available in the transaction log, which have already been produced by the conflicting committed UTs. The details of conflicting committed UTs are available in dependset(T_j). All the related lock entries of T_j are deleted. If T_j aborts, then also the lock entries of T_j are deleted.

Protocol for NCROTs:

7. *Lock acquisition.* Let T_k be a NCROT and requests for NR-lock to read 'x'. The lock request is entered into the *lockqueue*.

   7.1 If no transaction holds EW- or SPW-locks, the NR-lock is allocated to T_k. The step (8.1) is followed.

   7.2 If a preceding transaction holds SPW- lock, the NR-lock is granted. The identifier of preceding transaction that holds SPW-lock on 'x' is included in the T_k's *dependset.* The step (8.2) is followed.

8. *Execution.*

   8.1 T_k continues with the current executions by accessing 'x'. Step (8.3) is followed.

   8.2 Each execution of T_k is split into two speculative executions: one is with the before-image and the other one is with the after-image.

   8.3 If T_k obtains all the locks, step (9) is followed. Otherwise, step (7) is followed.

9. *Commit/Abort Rule.* Suppose one of the speculative executions T_{kj} of T_k has completed at time 't'. If the read set of T_{kj} contains the effect of all the conflicting transactions that have committed before 't', T_{kj} is retained and T_k's other speculative executions are aborted. Otherwise, T_{kj} is aborted. (Note that one of the speculative execution will be committed). If T_k is aborted, then all of its speculative executions are also aborted. Also the lock entries of T_k are deleted.

4 Simulation Results

In this section, we first explain the simulation model. Next, we present simulation results.

4.1 Simulation model

We have developed a discrete event simulator based on a closed-queuing model. We have a pool of CPU servers, all having identical capabilities and are serving one global queue of transactions. Each CPU manages two I/O servers. A CPU server serves the requests placed in the CPU queue in FCFS order. The I/O model is a probabilistic model of a database that is spread out across all the disks. A separate queue is maintained for each I/O server. Whenever a transaction needs service, it randomly (uniform) chooses a disk and waits in the I/O queue of the selected I/O server [23].

The description of parameters with values is shown in Table 1. The database size is assumed to be "dbSize". The parameters "cpuTime" and "ioTime" denote the I/O and CPU time associated with reading and writing an object (equivalent to an operating system page). The parameters "rotMaxTranSize" and "rotMinTranSize" are the maximum and minimum number of objects in ROT respectively. The maximum and minimum number of objects in UT is represented by the parameters "utMaxTranSize" and "utMinTranSize" respectively. Each resource unit (RU) constitutes 1 CPU and 2 I/O servers by considering that one CPU can drive two I/O servers. The parameter "noResUnits" represents the number of resource units. The parameter "MPL" denotes the number of active transactions exist in the system. The parameter "logOverhead" denotes the CPU time for reading the log in the reverse chronological order.

The parameter "% of UTs" denotes the percentage of UTs currently active in the system. The parameter "% of CROTs" means the percentage of CROTs active in the system. Let 'u' indicates the "% of UTs", which means that at any point of time, there are 'u' percent UTs active in the system. Let 'c' indicates "% of CROTs", which indicates that at any point of time, there are 'c' percent CROTs are active in the system. Note that, there are (100-u-c) percent NCROTs active in the system.

Table 1: Simulation Parameters, Meaning and Values

Parameter	**Meaning**	**Value**
dbSize	Number of objects in the database	1000
cpuTime	Time to carry out CPU request	5ms
ioTime	Time to carry out I/O request	10ms
rotMaxTranSize	Size of largest ROT transaction	20 objects
rotMinTranSize	Size of smallest ROT transaction	15 objects
utMaxTranSize	Size of largest UT transaction	15 objects
utMinTranSize	Size of smallest UT transaction	5 objects
noResUnits	Number of RUs (1 CPU, 2 I/O)	8
MPL	Multiprogramming Level	20
% of UTs	Percentage of UTs currently active	30% and 50%
% of CROTs	Percentage of CROTs currently active (10 to 50)	Simulation variable
logOverhead	Time to search transaction log	5ms

The value for "dbSize" is chosen as 1000 data objects [23]. This value is chosen to create a situation in which conflicts are more frequent. The value for "cpuTime" is chosen as 5 ms by considering the speed of modern processors [7]. The value for "logOverhead" is chosen as 5 ms by considering recent developments in log maintenance [8]. The value for "ioTime" is fixed as 10 ms by considering the speed range of current hard disk drives [1]. Regarding transaction size, we have chosen different parameter values for ROTs and UTs by considering the load character in modern information systems.The values for "rotMaxTranSize" and "rotMinTranSize" are fixed at 20 and 15 respectively and the values for "utMaxTranSize" and "utMinTranSize" are 15 and 5 objects, respectively [29]. The size of a ROT is a random number between 15 and 20 (both inclusive) and UT is a random number between 5 and 15 (both inclusive). We conducted the experiments by varying "% of CROTs" from 10 to 50.

Performance Metrics. We have employed the following performance metrics: throughput, UT throughput, ROT throughput and average number of speculative executions per transaction. Throughput is the number of transactions completed per second. UT throughput is the number of UTs completed per second. ROT throughput is the number of ROTs completed per second. Let 'e' denotes total number of speculative executions and 'n' denotes total number of transactions, then average number of speculative executions per transaction is equal to e/n.

Protocols. We have compared SSLR-S, with 2PL, FCWR and SSLR protocols. In all these protocols, we have assumed that aborted transactions are resubmitted again after the time duration equals to average response time in order to reduce repeated aborts. For SSLR and SSLR-S, we have assumed that all the speculative executions are carried out in parallel. We believe that with the availability of multi-core CPUs, parallel processing of speculative threads is feasible. Also, we have not taken into account the cost of deadlock detection as it is same for all locking-based protocols.

In the experiments, the graphs show the mean results of 20 experiments; each experiment was carried out for 10,000 transactions. The results are plotted with a mean of 95 percent confidence intervals. These confidence intervals are omitted from the graphs.

4.2 Experiments under 30% and 50% UTs

In the following experiments, we have reported the results by simulating environments in which 30% and 50% UTs are kept. Note that, the performance of 2PL, FCWR and SSLR protocols is not affected because of change in the "% of CROTs". This is because, these protocols consider both the CROTs and NCROTs as simple ROTs.

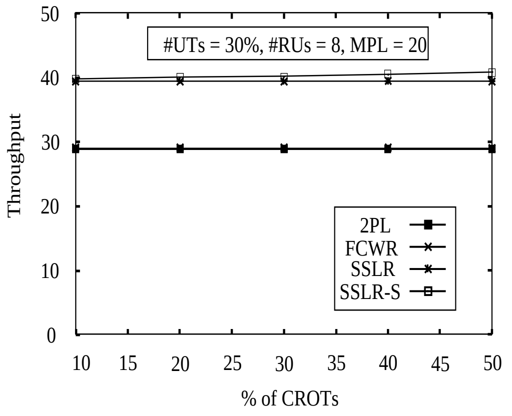


Figure 11: % of CROTs vs Throughput (30% UTs)

Figure 11 shows how throughput performance for 2PL, FCWR, SSLR and SSLR-S vary with "% of CROTs". We can observe that, SSLR-S protocol performs marginally better than SSLR in 30% UTs environment. In SSLR-S, UTs conflicting with CROTs are not blocked. However, UTs conflicting with UTs are blocked. So, the performance of SSLR-S is only marginally better than SSLR. In Figure 12, we can observe that difference in performance between SSLR

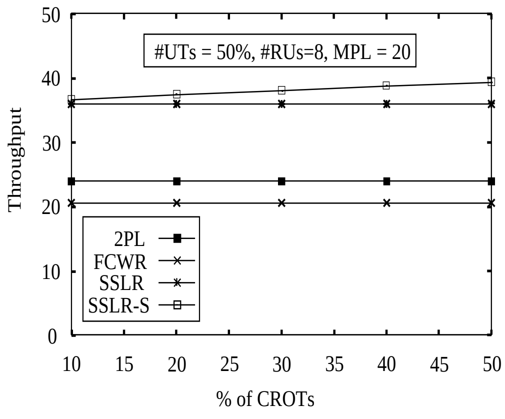


Figure 12: % of CROTs vs Throughput (50% UTs)

and SSLR-S is more than that of 30% UTs environment. As more CROTs are added in to the system, more UTs will complete their executions. So, the throughput performance has been improved in 50% UTs environment.

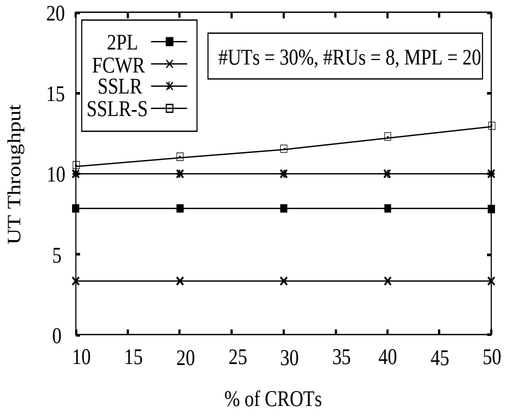


Figure 13: % of CROTs vs UT throughput (30% UTs)

Figure 13, shows how UT throughput performance of 2PL, FCWR, SSLR and SSLR-S vary with "% of CROTs". It can be observed that, SSLR-S protocol performs better than the remaining protocols in 30% UTs environment. SSLR-S protocol allows the UTs conflicting with CROTs to continue their executions without blocking. So, the UT throughput of SSLR-S is higher than the remaining protocols. From the Figure 14, we can observe that UT throughput performance of SSLR-S is better that of 30% UTs environment. As more CROTs are added in to the system, more UTs complete their executions. So, the UT throughput performance has been improved in 50% UTs environment.

Figure 15, shows how ROT throughput performance of 2PL, FCWR, SSLR and SSLR-S vary with "% of CROTs". We can observe that, the performance of SSLR-S decreases slightly as we increase "% of CROTs" under 30% UTs environment. The policy of allowing UTs conflicting with CROTs to continue their executions make more UTs to actively compete for resources than the situation of SSLR. In SSLR-S,

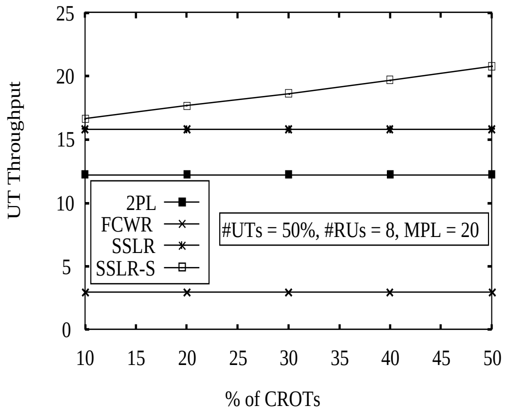


Figure 14: % of CROTs vs UT throughput (50% UTs)

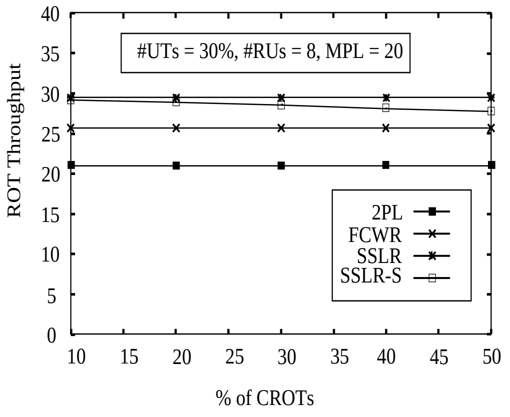


Figure 15: % of CROTs vs ROT throughput (30% UTs)

NCROTs have to wait in the queue for the generation of after-images by preceding UTs. So, the ROT performance of SSLR-S is slightly less than SSLR. Similar trend is observed in Figure 16 for 50% UTs environment.

Figure 17, shows average number of speculative executions per transaction required for SSLR and SSLR-S by varying "% of CROTs" under 30% UTs environment. We can observe that for SSLR-S, the average number of speculative executions per transactions has been less. This is because, CROTs are processed without speculation in SSLR-S. SSLR-S performs less number of speculative executions in 50% UTs environment than in 30% UTs environment, which can be observed in Figure 18. In 50% UTs environment, more UTs are allowed to enter into the system. Note that, UTs are processed without speculation. So, the average number of speculative executions per transaction decreases as "% of CROTs" increases.

5 Discussion and Implementation issues

Classification of ROTs: It is possible to classify the ROT as a compensatable-ROT if the computations

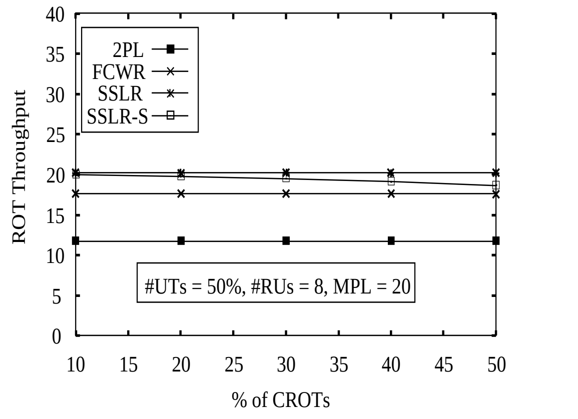


Figure 16: % of CROTs vs ROT throughput (50% UTs)

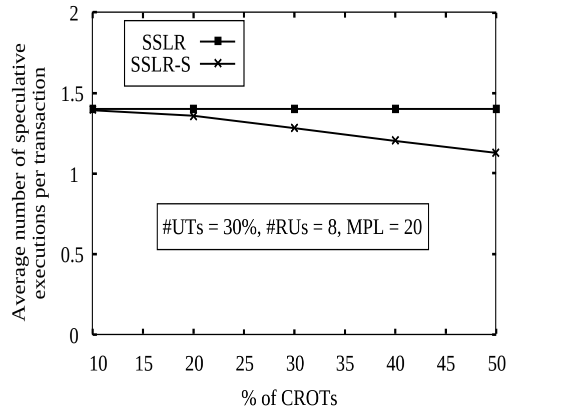


Figure 17: % of CROTs vs Average number of speculative executions (30% UTs)

performed by that ROT are compensatable. This classification information can be recorded in the transaction program itself. The problem is to identify the "compensatability" property of the operations. If the ROTs perform computations of arithmetic type, it is easy to identify the "compensatability" property of the computations. But for other computations, checking for "compensatability" property is difficult. We have to investigate this issue further.

Data currency issues: The data currency provided to the ROTs are very important for some web-based information systems. For example, let us consider the on-line stock exchange system, which provides the facilities like purchase and sale of company shares for its registered users. In this system, for the read only queries issued by the users, the SI-based protocols provide the data which is available in the database before the start of execution of current query, to the users and the decision taken based on this data may affect the financial benefits of the users. 2PL protocol provides recent data by including the effects of concurrent updates, but response time of 2PL is more. Whereas, the proposed approach would provide recent data by including the effects of concurrent updates. Moreover,

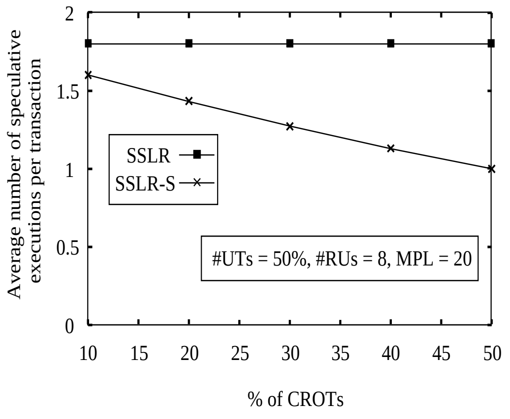


Figure 18: % of CROTs vs Average number of speculative executions (50% UTs)

the response time of proposed approach would be less than that of 2PL. So, the users would get recent data in a quick time and can take a better decision.

Performance enhancement of transactions specified in TPC-C benchmark: The TPC-C [2] is an online transaction processing workload. It is a mixture of read-only and update intensive transactions that simulate the activities found in complex OLTP application environments. TPC-C uses many tables and transactions. We consider only the STOCK-LEVEL table and NEW-ORDER and STOCK-LEVEL transactions for our discussion. The NEW-ORDER transaction updates S_QUANTITY column of STOCK table, for each item specified in the order. The STOCK-LEVEL transaction determines the number of recently sold items that have a stock level below a specified threshold. It reads the S_QUANTITY column of the STOCK table for each comparison. So from the above discussion, we can classify that STOCK-LEVEL as a compensatable-ROT and NEW-ORDER as a short UT. The proposed approach, can execute the transactions NEW-ORDER and STOCK-LEVEL without blocking. At the commit time, STOCK-LEVEL has to perform some compensating operations to include the updates of NEW-ORDER transaction. Thus the proposed approach improves the performance. Also, it provides high data currency to ROTs.

Performance comparison of 2PL, SSLR, SSLR-S and SI-based protocols: The performance comparison of 2PL, snapshot isolation-based, SSLR and proposed approaches are listed in the table 2. The proposed protocol provides a better performance than 2PL, SI-based and SSLR protocols. Note that, SI-based approaches provide less data currency to ROTs, whereas the proposed approach provides high data currency to ROTs. We can observe that, speculation-based protocols performs better than 2PL and SI-based protocols at the cost of extra processing resources. The throughput performance of the proposed approach is marginally better than SSLR. The UT throughput performance of the proposed SSLR-S protocol is better than the remaining protocols. We can observe that, SSLR-S protocol requires less average number of speculative executions per transaction than SSLR protocol. The proposed SSLR-S protocol and SSLR protocols require extra processing resources. Whereas 2PL and FCWR do not require extra processing resources. Note that, 2PL, SSLR and the proposed protocols satisfy serializability criteria, whereas SI-based protocols are not serializable.

6 Conclusions and future work

In this paper we have proposed an improved concurrency control protocol for ROTs by exploiting the notion called "compensatability". In this protocol, a UT need not block, if it conflicts with compensatable-ROTs. After completing its execution, the compensatable-ROT is able to compensate for the updates made by concurrent UTs. In this paper, we have defined the notion of "compensatability" and proposed an integrated protocol to improve the performance of ROTs by exploiting the notion of "compensatability" of ROTs and speculation. The simulation results show that the proposed protocol improves the performance over existing protocols. The results also show that the proposed protocol requires less number of speculative executions as compared to speculative protocol for ROTs which has been proposed in the literature.

As a part of future work, we are planning to investigate more about how an ROT can be made compensable by analyzing transactions specified in the benchmarks. We are also planning to investigate the performance by applying proposed protocol in a data warehousing environments.

Many web-based information systems often have to display information to the client requests by reading the data from the database and by performing some simple computations of "compensatable" type. In data warehousing applications, long running ROTs may perform some statistical computations of "compensatable" type. We have proposed our protocol by considering such applications. In the emerging e-commerce scenario, the proposed SSLR-S protocol provides the scope for improving the performance of ROTs without compromising data currency and correctness.

References

[1] Barracuda ES, The highest-capacity drives for the enterprise. *http://www.seagate.com/docs/pdf/marketing /po_barracuda_es.pdf*, 2006.

[2] TPC-C Benchmark. *http://www.tpc.org/tpcc/*, May 2008.

[3] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil and Patrick O'Neil. A crtique of ANSI SQL isolation levels. *ACM SIGMOD*, 1995.

[4] Resende, R., Agrawal, D. and Abbadi, A.E. Semantic locking in object-oriented database systems. *Proceedings of International Conference on Object-Oriented Programming Sytems Languages and Applications, OOPSLA '94*, pages 388–402, 1994.

[5] Satyanarayanan, O. T. and Agrawal, D. Efficient execution of read-only transactions in replicated multiversion databases. *IEEE transactions on Knowledge and Data Engineering*, 5:859–871, 1993.

[6] Paul Wu and Alan Fekete. The efficacy of commutatative-based semantic locking in a real-world application. *IEEE Transactions on data and Knowledge Engineering*, 20, 2008.

[7] Kam-Yiu Lam, Rei-Wei Kuo, Ben Kao,Tony S.H. Lee andReynold Cheng. Evaluation of concurrency control strategies for mixed soft real-time database systems. *Information Systems Journal*, 27:123–149, 2002.

[8] Sang-Won Lee and Bongki Moon. Design of flash-based DBMS: An in-page logging approach. *ACM SIGMOD 2007*, 2007.

[9] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'neil, Patrick O'neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30:492–528, 2005.

[10] Hector Garcia-Molina. Using semantic knowledge for transaction processing in distributed database. *ACM Transactions on Database Systems*, 8:186–213, 1983.

[11] Hector Garcia-Molina and Gio Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems*, pages 209–234, 1982.

[12] Bernstein, P. A., Hadzilacos, V. and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[13] J. N. Gray. The transaction concepts virtues and limitations. *Proceedings of VLDB conference*, 1981.

[14] Gray, J. and Reuter, A. *Transaction Processing: Concepts and techniques*. Morgan Kauf-mann, 1993.

[15] Muth, P., Rakow, T., Weikum, G., Brossler., P. and Hasse, C. Semantic concurrency control in object-oriented database systems. *Proceedings of International Conference on Data Engineering, ICDE '93*, pages 233–242, 1993.

Table 2: Comparison of 2PL, FCWR, SSLR and SSLR-S protocols

Parameter	**2PL**	**FCWR**	**SSLR**	**SSLR-S**
Throughput	Low	Medium	High	High
Correctness	Serializable	Not Serializable	Serializable	Serializable
Data currency	Maximum	Minimum	Maximum	Maximum
UT Throughput	Medium	Low	High	Higher than SSLR
Extra processing resources	Not Required	Not Required	Required	Required
Average number of speculative executions per transaction	Not applicable	Not applicable	1.4	Less than SSLR

[16] Hongfei Guo, Per-Ake Larson, Raghu Ramakrishnan and Jonathan Goldstein. Relaxed currency and consistency: How to say "Good Enough " in SQL. *ACM SIGMOD*, pages 815–826, 2004.

[17] Ragunathan, T. and Krishna Reddy, P. Speculation-based protocols for improving the performance of read-only transactions. *to appear in IJCSE.*

[18] Ragunathan, T. and Krishna Reddy, P. Improving the performance of read-only transactions through speculation. *DNIS 2007,LNCS*, 4777:203–221, 2007.

[19] Ragunathan, T. and Krishna Reddy, P. Extending specualtion for improving the performance of read-only transactions. *Ph.d. Workshop, EDBT 2008*, 2008.

[20] Ragunathan, T. and Krishna Reddy, P. Improving the performance of read-only transactions through asynchronous speculation. *HPCS 2008*, pages 467–474, 2008.

[21] Krishna Reddy, P. and Masaru Kitusuregawa. Speculative locking protocols to improve performance for distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 16:154–169, 2004.

[22] Akhil Kumar and Michael Stonebraker. Semantics based transaction management techniques for replicated data. *ACM SIGMOD Record*, 17:117–125, 1988.

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

[24] Dimitri Theodoratos and Mokrane Bouzeghoub. Data currency quality factors in data warehouse design. *Proceedings of the International Workshop on Design and Management of Data Warehouses*, 15:1–15, 1999.

[25] Badrinath., B. R. and Ramamritham. Semantic-based concurrency control: Beyond commutativity. *ACM Trans. Database Systems*, 17:163–199, 1992.

[26] Mohan, C., Hamid Pirahesh and Raymond Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. *ACM SIGMOD*, 1992.

[27] Sudhir Jorwekar, Alan Fekete, Krithi Ramamrithamand S. Sudarshan. Automating the detection of snapshot isolation anomalies. *VLDB 2007*, pages 1263–1274, 2007.

[28] Beeri, C., Bernstein, P. A., Goodman, N., Lai, M. Y. and Shasha, D. E. A concurrency control theory for nested transactions. *Proceedings of Second Symposium on Principles of Distributed Computing*, 1983.

[29] Kwok-Wa Lam, Sang H. Son, Victor C.S. Lee and Sheung-Lun Hung. Using separate algorithms to process read-only transactions in real-time systems. *Proceedings of the IEEE Real-Time Systems Symposium*, pages 50–59, 1998.

[30] Eswaran, K., Gray, J., Lorie, R. and Traiger, I. The notions of consistency and predicate locks in database systems. *Communications of the ACM*, 11:624–633, 1976.

[31] W. Weihl. Specification and implementation of atomic data types. *Ph.D. Thesis, MIT*, 1984.

[32] W Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37:1488–1505, 1988.