Optimizing Concurrent Processing of Write-then-Read Transactions

© Alexander Kalinin

Institute for System Programming of the Russian Academy of Sciences
allex.kalinin@gmail.com

Abstract

Write-then-read transaction ($W|R$) is a transaction that consists of two consecutive phases: write phase containing write and read operations in random order, and second phase containing read operations and write operations only on data items previously updated in the first phase. $W|R$ transactions are of great practical importance, but the problem of efficient processing of such transactions has received only little attention of research community so far. In this paper, we present Dynamic Versioning Protocol (DVP), which optimizes $W|R$ transactions processing using versions on the second phase. DVP supports STEAL policy and incorporates dynamic approach to the problem of selecting most suitable versions for read operations of the read phase. We prove the correctness of our protocol, so it guarantees the serializability of all transactions. The conducted experimental evaluation shows significant benefits of DVP for processing of concurrent $W|R$ transactions.

1 Introduction

There is a well-known problem of concurrent processing of update transactions (usually called *updaters*) that update some data items more or less randomly and read-only transactions (usually called *queries*) that consist only of read operations. One of the most widely used protocols, two-phase locking (2PL [9]), does not efficiently support such processing [18], because it cannot efficiently handle data contention that may exist between updaters and queries. This problem is of great practical importance and, as a result, a variety of approaches have been proposed to address this issue.

The solution to the concurrency problem that avoids data contention between updaters and queries is to maintain multiple versions of data items. This is generally called *multiversioning* [20]. In multiversioning, data contention is reduced by allowing queries to read obsolete versions (that are, nonetheless, transaction-consistent), while updaters create new versions of data items.

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems, Saint-Petersburg, Russia, 2008

Most multiversion algorithms support only two transaction classes: updaters and queries. On the other hand, there exists such usable class as *write-then-read* transactions ($W|R$ transactions, for short). This transactions consist of two consecutive phases: write phase containing write and read operations in random order, and second phase containing read operations and write operations on data items only previously updated in the first phase. Integrity constraints (IC) checking implies using such $W|R$ pattern when such checking occurs at the very end of transaction. Of course, other types of IC's checking may yield different transaction patterns, which is not the goal of this paper.

Most concurrency control protocols process $W|R$ transactions as updaters (usually using 2PL), which leads to unnecessary delays of $W|R$ transactions. Let us consider an example.

Example 1. *Consider a system for items supplies management. This system uses two relations:*

`Supply(supp_id, date, item, item_number, ...),`

`ItemInfo(item_id, rating, ...)`

Manager can request supply of a particular item. The corresponding order is placed in `Supply` *relation. System checks reasonability of this request by checking* `ItemInfo` *relation, which stores some information for each item (rating, sale statistics, etc.). If some criterion (low sales for the item, for example) is not sufficient, request for supply will be denied.*

Thus, we could have two transactions here. Let T1 *be a $W|R$ transaction that inserts tuple s in* `Supply` *relation. Then this transaction reads tuple i from* `ItemInfo` *relation to check the criterion. Let* T2 *be a transaction that runs concurrently and updates* `ItemInfo` *relation by updating tuple i from it. Consider one of the possible histories:*

`W1(s) W2(i) R1(i) c(T2) c(T1)`

All concurrency control protocols that process updaters using 2PL rules would not allow such a history. For example, strict two-phase locking protocol (S2PL[3]) will block T1 on `R1(i)` *and will not allow it to read i before commit of T2.*

But transaction T2 could be serialized after T1 if T1 do not see version of tuple `i` *created by T2.*

Unfortunately, class of $W|R$ transactions has not received much attention in the database research community. As far as we know, there is only one protocol

that optimizes processing of $W|R$ transactions: an extension of the multiversion two-phase locking protocol (EMV2PL [13]).

However, the approach of [13] suffers from several important shortcomings. First, EMV2PL does not support STEAL policy [11], what implies that buffer manager cannot write noncommited updated pages back to the database before commit of transaction. STEAL policy is supported in most commercial database systems and we consider such support in our protocol as one of its major features. Second, EMV2PL maintains unrestricted number of versions, which leads to high storage overhead and complex version management. As shown in papers [15, 14] multiversion protocols, which bound the possible number of versions obtain serious performance advantages in comparison with protocols without restrictions on number of versions. Third, EMV2PL employs *static approach* to the problem of selecting appropriate versions for read operations on the second phase. That is, EMV2PL obtains timestamp at the end of the first phase and as a result on the second phase transaction is bound to read versions that precede obtained timestamp. This approach is simple, but in many cases more recent versions of data items could be read. Note, that the research conducted by Carey et al. [4] shows that reading younger versions of data items rather than older ones gives serious performance benefits.

Towards the goal of optimizing processing of $W|R$ transactions, this paper introduces a new Dynamic Versioning Protocol (DVP), which produces only serializable schedules and supports an efficient processing of $W|R$ transactions. Our protocol allows more efficient processing by employing following techniques. Firstly, it allows $W|R$ transactions to release all their read locks after executing their first phase and to execute second phase without acquiring new locks. Secondly, DVP supports STEAL policy, which is one of its major benefits. Thirdly, it maintains only fixed number of versions per each data item. At last, DVP introduces *dynamic approach* to the problem of selecting most suitable versions for read operations on the second phase, which allows reading of the most recent versions possible on the second phase without experiencing unnecessary delays. So, DVP ovecomes all aforementioned shortcomings of EMV2PL protocol.

To explain in more details the benefits of dynamic approach let us consider one more example.

Example 2. *Consider transactions T1 and T2 from previous example. Concurrent execution of this transactions might produce following history:*

`W1(s) W2(i) c(T2) R1(i) c(T1)`

A static approach obliges T1 to read old versions of tuple i. So, EMV2PL serializes T1 before *T2. On the other hand DVP dynamically (i.e. during T1 execution) determines that there are no existing conflicts between T1 and T2 transactions and serializes T1* after *T2. Therefore, it allows reading of more recent version of i.*

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 provides description of DVP. Section 4 discusses some possibilities of using our approach for query processing. Section 5 contains performance study and shows our protocol to be very efficient for processing concurrent write-then-read transactions. Finally, Section 6 concludes this paper.

2 Related work

To address the problem of data contention between concurrent transactions two-version protocols were proposed in [1, 17]. Two-version approach greatly simplifies version storage management, but it does not eliminate data contention, because only one old version is available at the time. Moreover, such protocols do not support processing of queries, because all transactions are processed as updaters.

Another approach is a multiversion two-phase locking (MV2PL) [3, 8, 6, 5]. It supports processing of two different type of transactions: queries and updaters. Updaters are processed according to S2PL. Query always reads the most recent versions that were committed before its start. However, to conduct such processing protocol must maintain unrestricted number of versions, because it cannot just simply discard versions that can be requested by queries. This leads to a considerable storage overhead. To address the problem of garbage collecting various approaches have been proposed. In [16, 6, 7] old versions can be discarded, and, as a result, active queries may have to be rolled back because of abscence of required versions for reading. Such rollbacks often affect long-running queries, and that complicates their processing. This problem has been eliminated in [19], but at the expense of complex initialization phase. During this phase queries must register all their future read actions, which prevents corresponding versions from discarding. Also, MV2PL uses timestamps to determine versions for reading. Every updater must write such timestamp on all the versions it has created at the time when it commits. That is why MV2PL does not support STEAL policy, which makes effective management of versions somewhat difficult.

Protocols supporting fixed number of versions were proposed to address some of the issues [21, 14, 15, 12]. In this protocols queries read transaction-consistent snapshots of the database. Since number of snapshots is restricted, snapshot advancement becomes a very important matter. [21, 14] present dynamic finite versioning (DFV) schemes, which support dynamically derived snapshots and effective query processing. It uses techniques for dynamic obtaining and advancement of snapshots without need to interrupt transaction processing. Another approach, presented in [15], deals with problem of versioning indexed data, but implementation described there enables advancement only for the oldest snapshot. In contrast, DFV (and DVP as well) enables advancement for any snapshot that is not in use by any query. The AVA3 protocol proposed in [12] needs at most three versions to guarantee non-interference of queries and updaters, but queries still may access out-of-date data even right after version advancement procedure. Protocols mentioned here do not address, however, another important issue: optimization of read operations for updaters. Updaters are processed according to S2PL protocol, and that makes difficult an efficient processing of, for example, $W|R$ transactions.

One of the solutions that makes possible efficient pro-

cessing of read operations in concurrent transactions is *Snapshot Isolation* [2, 16]. It processes all transactions in the same manner. Update operations work as usual, with acquiring of write locks, while read operations work with versions from some snapshot that was created before transaction's start. This method guarantees that read operations are never blocked, because there is no need in obtaining any read locks. However, *Snapshot Isolation* has its own drawbacks. First, old versions must be maintained in some way, and aforementioned problems with garbage collecting may occur. Second, it uses the principle called *First-commiter-wins*, which prevents lost updates, but at the same time can cause frequent rollbacks. And last, *Snapshot Isolation* can allow non-serializable executions.

Another approach, which supports processing of $W|R$ transactions, is EMV2PL [13]. EMV2PL is an extension of MV2PL protocol. As a result it inherits all of its drawbacks such as unrestricted number of versions and absence of support of STEAL policy. $W|R$ transactions are processed in a different way than in DVP. Transition between phases (*LockPoint* in EMV2PL) results in release of all read locks, but second phase is processed similar to as for query: every read operation reads the most recent version that was committed before *LockPoint* time. As a result, second phase proceedes without acquiring any locks, but at the same time it can lead to reading more obsolete versions.

[4] presents another protocol, called multiversion query locking (MVQL), which deals with problem of obsolence for queries. MVQL is based on MV2PL protocol, but allows weaker form of consistencies for queries [10]. MVQL supports four types of consistencies: strict, strong, weak and update. Rules for each type of consistency can be described in terms of relaxation of some serialization graph constraints. Strict and strong forms allow only serializable execution, while weak and update forms accept non-serializable execution, by allowing some types of cycles in serialization graph. The weaker form of consistency, the more recent versions can be read. As a result, MVQL allows benefits only at the cost of serialization.

3 Dynamic Versioning Protocol

In this section we will describe our protocol. DVP is an extension to DFV schemes and it uses some details of their design. The comprehensive description of DFV can be found in [21], but for the sake of clarity we will describe some of the details here along with our innovations. In the first section we will describe basic principles of version identification, which is the foundation of our protocol. The next three sections describe processing of different types of transactions. In the last two ones we will discuss some theoretical issues.

3.1 Basic principles of version identification

DVP is a multiversion protocol. For each data item it maintains several versions. The number of versions for each data item is limited and this limit is a parameter of our protocol. In this paper we assume data items to be pages. Data granulatity is not essential for our protocol, and it can be implemented to be used with records, for example.

Definition 1. *Logical snapshot of the database is a collection of versions, one per each data item, that represents transaction-consistent state of the database.*

Definition 2. *Committed version is a version that has been created by now committed transaction.*

Some of the versions are used as parts of logical snapshots of the database. Our protocol supports any number of snapshots, but for the sake of clarity we will describe the case of two snapshots. We will refer to them as "current snapshot" (CS) and "previous snapshot" (PS). The former will represent the most recently obtained logical snapshot, and the latter will represent logical snapshot that has been obtained at some moment before the current one.

Each data item can have several types of versions:

Definition 3. *Last-Committed Version (LCV) is the most recently committed version of a data item.*

Definition 4. *Working Version (WV) is a version of a data item that has been created by some not yet committed (i.e. active) transaction.*

Definition 5. *Previous Version (PV) is an obsolete committed version of a data item, which has been created before Last Commited Version.*

Each data item has at least one version – LCV. If it has some other committed versions, they can be considered as PV versions. Of course, PV versions may not exist at all. For example, when data item has been created and has never been updated since.

If WV version exists, there cannot be another one for the same data item, because active updaters are processed according to S2PL rules, so they cannot obtain Write-locks for the same data item at the same time. For some data items such version, of course, may not exist at all in the case of absence of active transactions that update them. When updater commits, all WV versions it has created are converted into LCV versions, and "old" LCV versions at the same time are converted to PV versions.

So, all versions of a data element can be divided into three categories: WV, LCV and PV. Belonging to a snapshot can be considered as an additional property of a version. For example, version from current snapshot is always labeled as LCV and CS or PV and CS. It cannot be labeled as WV and CS, because WV cannot be part of any snapshot, as such version has not been committed yet.

It is important to mention that these labels are not physically stored on versions. Our protocol identifies version labels using some auxillary structures described below.

To dynamically identify versions DVP uses following structures: list containing active (i.e. not committed) transactions (we will call it $ActList$), timestamp of version (which is obtained at the moment of creation of version) and version's creator identifier (which is an identifier of corresponding updater). Timestamp can be represented as some kind of ascending counter. So, when new version is created, its timestamp is assigned counter's

value, and counter itself increments. Creator identifier uniquely identifies transaction that created the version. Each transaction obtains such an identifier at the very start and gives it to every new version it creates.

$ActList$ is a global list, but timestamp and identifier are version-dependent and must be stored for each version (e.g. on a page itself). Below we will describe dynamic approach to identify LCV, WV and PV versions. Dynamic identification of snapshot versions will be described in Section 3.3.

Consider list V, containing metadata (timestamp and creator identifier) about all versions of some data item[1]. Let's assume that this list is sorted in descending order by timestamps. We identify versions for each data item as follows:

- **WV.** This version is always the most recent one. Thus, it must have the largest timestamp among other versions. So if it exists, it is the first one in V. And this version exists iff there exists an active transaction that updates this data item. Summarizing, if creator identifier of the first version in V lies in $ActList$ then it is a WV version. Otherwise, WV version does not exist.
- **LCV.** This version is the most recent one among committed versions. So only WV can have larger timestamp. Thus, we have two possibilities here. If WV version exists, then LCV is the second one in V. If WV version does not exist, then LCV is the first one in V, because in this case the data item has only committed versions.
- **PV.** Previous versions are all versions except LCV and WV.

3.2 $W|R$ transactions

Definition 6. *$W|R$-transaction is a transaction that consists of two consecutive phases: write phase containing write and read operations in random order, and second phase containing read operations and write operations on data items only previously updated in the first phase.*

On the first phase $W|R$ transaction is processed as an updater according to S2PL protocol. Such transactions always start as updaters and become $W|R$ ones only when they switch to the second phase. So transaction manager (TM) does not need to know in advance that new transaction is $W|R$ one. We will describe updaters processing in Section 3.4. This section and the next one describe design details of the second phase. Efficient proccesing of the second phase of $W|R$ transaction is our main technical contribution.

On the second phase $W|R$ transaction can read any data items, but it can only update pages previously-locked on the first phase. So it cannot obtain new write locks. Write operations are processed in a usual way, but read ones require different approach. To get performance gains read operations are processed without acquiring any common locks. This could cause a serialization fault, e.g. when the same data item is already acquired for writing. To guarantee serializability we enable $W|R$ transaction to read one of the PV versions. This happens only when read operation of this $W|R$ transaction conflicts with some another operation on the same data item. The meaning and definitions of different types of conflicts will be exlained in the next section.

We use some kind of notifications of read operations, called *read-notification-locks* (RN-locks). RN-lock is not a lock in the common sense of this word. It is rather some kind of flag that indicates reading of a data item. But this flags can be easily thought of as some type of locks that are compatible with every other types of locks, even with the exclusive ones. So, they can be implemented as a part of a lock manager. That is why we call it this way. To keep information about read operations in the first phase, TM converts usual read locks obtained at the first phase into RN-locks (this operation, of course, can unlock some transactions that have been locked according to S2PL protocol). Before executing each read operation on the second phase, $W|R$ transaction also obtains RN-lock for the corresponding data item. In contrast to S2PL, $W|R$ transactions almost do not experience unnecessary delays, which is a benefit of our approach.

RN-locks provide information about read operations, but to maintain information about concurrent transactions, we need another structure, called $FollowSet$. For each $W|R$ transaction T, $FollowSet_T$ stores identifiers of transactions that must be serialized after T. Two main rules about $FollowSet$ are:

- All transactions that belong to $FollowSet_T$ must be able to see all versions created by T.
- T must not be able to see versions created by transactions from $FollowSet_T$.

Using this structure TM can easily find version for reading. It just selects the most recent committed version whose creator has not got into $FollowSet_T$. In this case $W|R$ transaction does not see versions created by transactions that have been serialized after it. Repetitive readings of the same data element by T will affect the same version, because creators of new ones also will be placed into $FollowSet_T$. Proper maintenance of $FollowSet$s is essential for efficient processing of $W|R$ transactions. In the next section we will describe this issue in more detail.

3.2.1 Maintenance of FollowSets

In this section we will describe design details for $FollowSet$s. The main idea is to dynamically determine conflicts between operations of concurrent transactions. Each concurrent transaction that conflicts with $W|R$ transaction T and is serialized after it must be placed in $FollowSet_T$. Conflicts can be divided into two classes: direct and indirect.

Definition 7. *Two operations conflict when they belong to different active transactions, they are on the same data element and one of them is write.*

Definition 8. *Direct conflict is a conflict between operations of two active concurrent transactions one of which is $W|R$ transaction on its second phase.*

[1] In our implementation we store list V in the header of the page that represents last version.

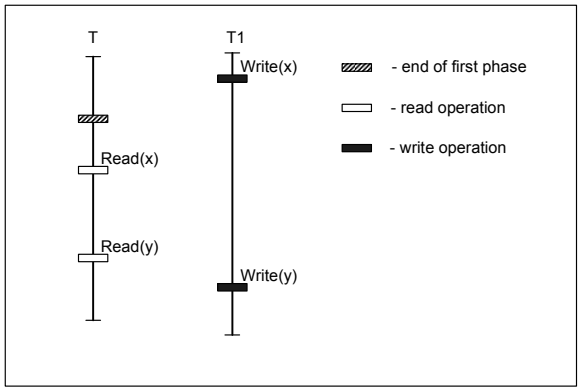

Figure 1: Example of direct conflicts

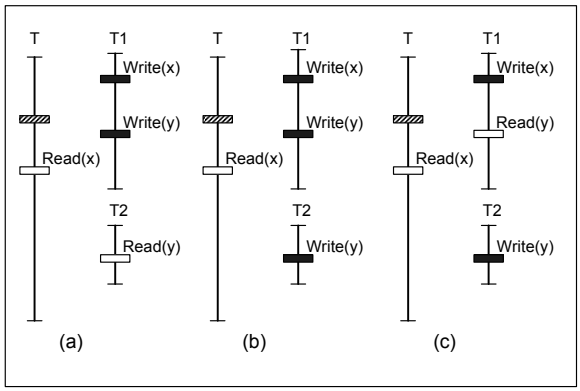

Figure 2: Indirect conflicts

Direct conflict occcurs only when some $W|R$ transaction reads a data element on the second phase while some other transaction updates it. Such conflicts can be easily recognized by TM at the time of locking of data items. This is illustrated by the following example:

Example 3. *Consider Figure 1. T is a simple $W|R$ transaction, which reads data elements x and y on the second phase, and $T1$ is a concurrent updater, which updates this elements. Let's look at the data item x. Before reading x, T obtains RN-lock. In this case TM determines that x is already locked by $T1$. Direct conflict between T and $T1$ is recognized and $T1$ is placed into $FollowSet_T$.*

Now consider element y. In this case $T1$ obtains Write-lock before updating y. Again, direct conflict is recognized, because T is already obtained RN-lock on y, and $T1$ is placed into $FollowSet_T$.

Direct conflict between two $W|R$ transactions can cause a delay at the second phase. Consider $W|R$ transactions $T1$ and $T2$. $T1$ updates some data element x and $T2$ reads it (on the second phase). But if at that moment $T2$ is already in $FollowSet_{T1}$, then it must read version created by not yet committed $T1$. To prevent reading uncommitted data TM delays $T2$ until $T1$ commits.

Definition 9. *$W|R$ transaction T and concurrent transaction $T1$ conflict indirectly, when they each conflict directly with some other concurrent transaction $T2$.*

To clarify this definition, consider for example, $W|R$ transaction T and updaters $T1$ and $T2$. When $T2$ conflicts directly with T and $T1$ then it causes indirect conflict between T and $T1$. Indirect conflicts can be of three types: W-R, W-W and R-W. We describe each type of conflict providing the only possible scenario for each of them to occur:

W-R: W-R conflicts occur due to read-after-write operations on the same element of data. Consider Figure 2(a). Transactions T and $T1$ conflict directly on data element x. So $T1$ is placed in $FollowSet_T$. During its execution $T2$ reads version of y created by $T1$. Hence, it is serialized after $T1$ and must be also placed in $FollowSet_T$. To prevent such conflict, TM obtains information about creator of the version selected for reading and uses it to put transaction in corresponding $FollowSet$. For example, when $T2$ reads y, TM puts it in $FollowSet_T$, because creator of the corresponding version $T1$ is already in it.

W-W: This type of conflict occurs due to precedence relation between versions [3]. Precedence relation in our protocol is determined by S2PL rules. As a result, order of precedence coincides with order of serialization of corresponding updaters. This corresponding updaters never conflict directly, but they can cause an indirect conflict. Consider Figure 2(b). This case is somewhat similar to the previous one. But this time $T2$ updates element y. $T2$ must be placed in $FollowSet_T$, because its version of y succeeds version of y created by $T1$. In this case, to prevent conflict, TM uses information about creator of LCV version. For example, $T2$ will be placed in $FollowSet_T$, because creator of LCV $T1$ is in it. We need to check creator of LCV version, because updated page always becomes LCV version after its creator commits.

R-W: This type of conflict is always due to creating new version of data element that has been read by another transaction. Consider Figure 2(c). Transaction $T2$ creates a new version of element y, but when $T1$ was reading y it did not see this new version. So $T2$ is serialized after $T1$ and it also must be placed in $FollowSet_T$, because $T1$ has been put in it. To handle such conflicts TM needs different approach. When some transaction updates data element, TM cannot determine if this element has been read by another already committed transaction. To keep information about read operations of updaters TM uses another technique called *Lock Inheritance*. When updater commits, TM converts all its read locks into RN-ones and passes them to $W|R$ transactions that contain committed updater in their $FollowSet$s. Then R-W conflicts can be handled as direct ones. In our scenario, when $T1$ commits TM converts read lock on y into RN-lock and passes it to T. When $T2$ updates element y direct conflict occurs between T and $T2$. TM handles this conflict by putting $T2$ into $FollowSet_T$.

There are some important issues here:

- $FollowSet$ is used by TM only at the second phase. So it is always empty at the start of the second phase.
- $FollowSet$ is used only for $W|R$ transactions and there is no need to maintain such a structure for another type of transactions.
- When TM places some transaction T into $FollowSet_{T1}$ it must also place it in each

other $FollowSet_{T2}$ that contains $T1$ itself. This happens because T must be serialized after $T1$, and $T1$ must be serialized after every $T2$. Thus, in this case T also must be serialized after every $T2$.

- When $W|R$ transaction T is placed in some $FollowSet_{T1}$ the contents of its own $FollowSet_T$ must be placed with it. This happens because $FollowSet_T$ contains transactions that must be serialized after T. So in this case they must be serialized after $T1$ too.

The last two actions enforce serializability and are in total consistence with the definition of $FollowSet_T$ as a set containing all transactions that must be serialized after T.

3.3 Logical snapshots and queries processing

Queries are transactions that contain only read operations. Since such transactions are processed in a special way, TM must know that a new transaction is indeed a query. For efficient processing of queries DVP maintains some number of logical snapshots. When a new query begins it just starts reading current snapshot, which it cannot change during execution. Moreover, it does not need to acquire any kind of locks, even notification (RN) ones. Since queries do not obtain any locks they also do not participate in any type of conflicts described earlier. So this transactions never experience delays of any kind during their execution.

Logical snapshot is a set of versions that represents transaction-consistent state of the database. This state corresponds to the moment of snapshot's creation. As a result, versions belonging to snapshot can be slightly obsolete. There exist different approaches to creation of snapshots. The naive one would be to wait for active transactions to finish and to not start new ones. Then, at the moment when there are no active transactions, snapshot can easily be obtained. Such approach is prohibitive in OLTP systems. Instead DVP uses diffirent, dynamic approach that does not require any manipulations with transactions. The main idea of this approach is that TM can dynamically determine versions belonging to snapshots using some simple auxillary structures. Thus, such versions do not differ from the usual ones and version's metadata (its timestamps and creator's identifier) can be used to determine if it belongs to some snapshot. For each snapshot TM only needs to store timestamp of its creation (T_{CS} and T_{PS} for our snapshots) and a copy of $ActList$ ($ActList_{CS}$ and $ActList_{PS}$). Timestamp of snapshot is a value of the same counter we use for the creation of versions.

Let V represent list of versions ordered by timestamps as described in Section 3.1. Then algorithm to find version belonging to a current snapshot would be:

1. Obtain sublist V' of V that contains versions with timestamps less than T_{CS}.

2. Select LCV version from V' as described in Section 3.1, using $ActList_{CS}$.

In fact, V' contains versions that existed at the moment of snapshot creation. The problem is that this sublist can contain WV version as well, because TM did not wait until it became committed. That is where $ActList_{CS}$ comes in. Since it is a copy of $ActList$ at the moment of snapshot's creation, all that TM needs is to select LCV version from V'. For any other snapshot (e.g. previous one) the procedure would be the same.

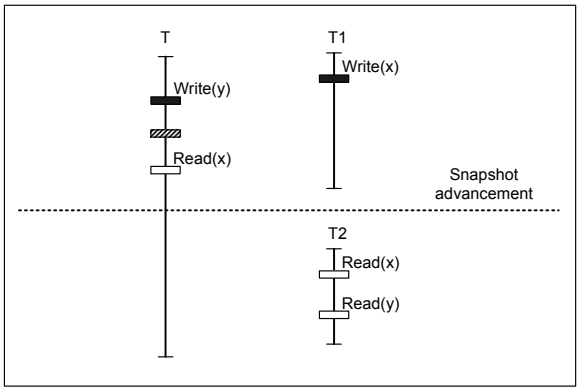

Figure 3: Scenario for snapshot creation failure

To take a new snapshot TM simply must obtain timestamp and a copy of $ActList$. In this case there is no need to use the naive approach described earlier. However, there are some restrictions here as well.

First of all, to be transaction-consistent snapshot must contain versions from all transactions that had been serialized at the moment of its creation. Since our protocol maintains serializability for $W|R$ transactions in a special way, there is some possibility that creation of a snapshot may be delayed. TM delays snapshot creation only when some $W|R$ transaction is active and its $FollowSet$ is not empty. Again, there is only one possible scenario for this:

Consider Figure 3. Updater $T1$ directly conflicts with $W|R$ transaction T. Hence, it is placed in $FollowSet_T$ and is serialized after T. If TM took snapshot as depicted, it would not contain version of y created by T. Thus, query $T2$ would see version created by $T1$, but not by T, which is serialized before $T1$. When query reads version of some data element, it must be able to read all the versions that have been serialized before this first one. So this snapshot is not transaction-consistent.

Second restriction is due to that the number of snapshots is fixed. If some snapshot is in use by any number of queries, it cannot be replaced by the new one. To be able to take a new snapshot TM must have a free one. New queries always start using current snapshot (i.e. CS version). So if current snapshot is free, TM just takes another one by obtaining timestamp in T_{CS} and copying $ActList$ into $ActList_{CS}$. But if current snapshot is used by other queries, then it must be kept intact for them. TM uses previous snapshot for this purpose. To backup current snapshot it just copies T_{CS} into T_{PS} and $ActList_{CS}$ into $ActList_{PS}$. However, if previous snapshot is also in use by other queries, then creation of a new snapshot must be delayed.

3.4 Updaters processing

Updaters are transactions that contain read and write operations in random order. They are processed according to S2PL protocol. That means that transactions must ac-

quire read and write locks. As we explained before, when updater commits TM converts all its read locks into notification ones (RN-locks) and passes it to the corresponding $W|R$ transactions. However, there is a restriction imposed on updates by versioning environment. When transaction updates some data element for the first time it creates new version of this element. That is not a problem when number of versions is unrestricted. In our case, though, this number is fixed and TM must choose a version for replacement.

LCV versions cannot be overwritten because they are the only source of previous versions. If TM replaced LCV versions, it would eliminate versioning at all and DVP would become similar to S2PL. PV versions, however, can be considered for replacement. But we have several limitations here as well:

1. PV version that belongs to the current snapshot cannot be replaced, because current snapshot is used by new queries.

2. PV version that is read now by some active $W|R$ trasaction cannot be replaced.

3. PV version that might be read by some active $W|R$ transaction in the future cannot be replaced by now. We call such situation a *potential reading*. If TM chose to replace such version, in the future $W|R$ transaction might find no version at all.

4. PV version that belongs to a previous snapshot can be replaced, if there are no queries that use this previous snapshot. Of course, in this case we must consider three aforementioned limitiations as well.

Thus, TM can only replace PV versions that satisfy this restrictions. In the case of potential reading it acts as if $W|R$ transaction is already reading this item. So, if it creates a new version for data item x, for every active $W|R$ transaction it must determine the version of x it is reading now or may be reading later using $FollowSet$ as described in Section 3.2.

If version for replacement exists then it is replaced by the new one. Otherwise updater must wait for some version to become free. It is important that updater will not be locked permanently, because new transactions do not use PV or PS versions and old transactions will eventually finish. In this case some version will become free and updater will be unblocked.

3.5 Correctness

Theorem 1. *DVP guarantees serializability of all transactions.*

We provide the proof for this theorem in Appendix A. Note, that serializability is the key issue for any protocol. Many applications rely on serializability and cannot simply sacrifice it for perfomance gains.

3.6 Deadlocks

Theorem 2. *$W|R$ transactions cannot experience deadlocks during the second phase.*

The proof for this theorem can also be found in Appendix A. This is a very important result. It can significantly increase transaction throughput in the situations with strong data contention between concurrent transactions. We will discuss this issue in the light of experimental results.

4 $W|R$ transaction as a query

In this section we discuss the possibility of using $W|R$ transactions as queries. $W|R$ transaction is similar to query when its first phase contains no operations. In this case it is transfered to the second phase right at the start, and because $W|R$ transaction cannot acquire write locks on the second phase, it can only read data. The main difference is that such $W|R$ transaction still has to obtain RN-locks. As a result, it can participate in some of the aforementioned conflicts. But at the same time it uses dynamic approach to find suitable version for reading. Query, on the other hand, uses snapshot, which is completely defined at the moment of querie's start. Moreover, such snapshot cannot be updated if some long-running queries are keeping busy all snapshots, because their number is fixed. Hence, new queries would have to read obsolete versions of the pages. Dynamic approach can yield some benefits in such a case, because it selects version based on current situation, and it is independent of concurrent long-running queries.

Using $W|R$ transactions as queries has its own drawbacks. As we mentioned before, if some version has been (or will be) used by some active $W|R$ transaction, it cannot be replaced by a new one. This creates a potential bottleneck for updaters. If number of such $W|R$-query transactions is high enough, they might cover all versions and updaters might experience delays. It may happen with usual $W|R$ transactions too, but their second phase's length generally is not big enough to cause any serious delays. However, in many cases even such delays might not be such a major drawback. In other multiversion protocols queries can be started as updaters. In this case they read the most recent versions. But at the same time they must acquire read locks, and, as a result, they experience delays. Such queries-updaters can even become deadlock victims. $W|R$-query transaction, on the other hand, does not acquire any read locks and cannot experience deadlocks. Thus, its performance can be significantly better than querie-updater's. So such $W|R$-query transactions represent some kind of trade-off between performance of updaters and reading of less obsolete versions.

5 Experimental Evaluation

We have developed prototype to conduct experimental evaluation of our protocol. This prototype simulates the work of the buffer manager, the version manager and the lock manager. Each transaction consists of write and read operations on the data elements, which in our experiments are pages. It is important to mention that since all protocols were tested on the same prototype, the possible implementation details of this prototype should not influence the results.

We compared DVP with two other protocols: multiversion DFV and nonversion S2PL. Comparision with

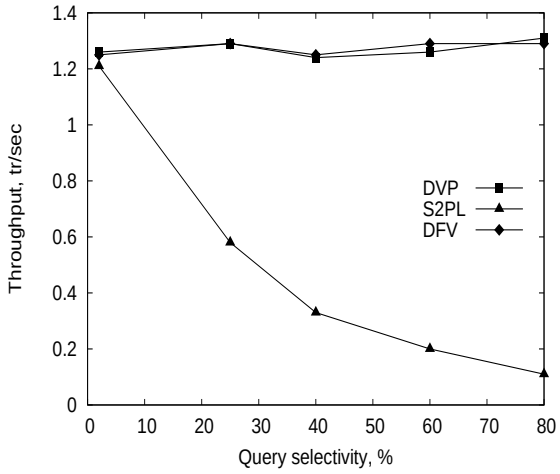

Figure 4: Updater throughput

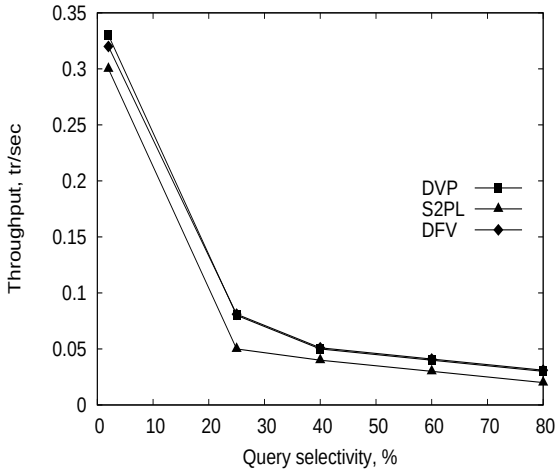

Figure 5: Query throughput

S2PL allows us to study benefits of versioning approach in processing of $W|R$ and other types of transactions. But our main goal is to test our protocol against DFV, which also supports queries, STEAL policy and limited number of versions per each data item. It does not support $W|R$ transactions, however, which allows us to study benefits of our approach.

5.1 Experiment 1: Benefits of multiversion environment

In this experiment we compare multiversion algorithms (DVP and DFV) with usual (S2PL) in the case when strong data contention between queries and updaters exists. Our workload contains 25% of queries and 75% of updaters running concurrently. Selectivity factor for queries is varied from 2% to 80% of the database, but within the same experimental run it stays fixed. Updaters, on the other hand, are short, containing only three write operations. This allows us to reduce data contention between them. In this case almost all conflicts are between queries and updaters. Write actions for updaters are generated randomly, while queries always read some continuous region of the database.

Figure 4 shows updater throughput. As we can see, for multiversional DVP and DFV queries selectivity is not an issue. Queries just read snapshots, while updaters write new versions. Updaters do not experience any unnecessary delays, because they do not intersect with queries. Any updaters delays would be because of data contention between updaters themselves, but we have reduced it to minimum by allowing them only three operations. Data contention becomes a problem for S2PL, though. In this case queries and updaters read the same pages. In fact, queries are executed as an updaters here, obtaining read lock before each read operation. It does not create any problems when query selectivity is about 2%, because in this case queries are relatievely small and data contention is not high enough to cause problems. However, increasing query selectivity also increases execution time of queries and data contention. Queries hold read locks for a considerable ammount of time, and, according to S2PL protocol, updaters that read the same pages have to wait for this locks to be released. At 80% selectivity almost all new updaters conflict with queries and, as a result, spend most of the time waiting for them. This results to a very poor performance of updaters under S2PL.

Figure 5 shows query throughput. DVP and DFV process queries in a similar manner, and, as a result, query throughputs for this two algorithms are almost equal. Again, S2PL loses to multiversion algorithms. In fact, this experiment compares overhead of query processing for S2PL and multiversion algorithms. For S2PL overhead consists of locking needed pages and possible waiting for concurrent updaters. For multiversion algorithms overhead includes one possible additional read operation on each basic read. This additional operation occurs when query reads old version of a page. In this case query first reads LCV version, which contains information about all remaining versions, determines version for reading and reads it. If LCV version is a suitable version itself, no additional read occurs. As we can see on Figure 5, S2PL overhead exceeds multiversion one, and, as a result, query throughput is lower for S2PL.

Our first experiment shows that multiversion environment has benefits in both query and updater processing. It offers better query throughput, which can be an important issue for some applications. More importantly, it offers significantly better updater throughput, which is important for efficient OLTP.

5.2 Experiment 2: $W|R$ transactions

This experiment shows benefits of our protocol in $W|R$ transactions processing. The whole workload entirely consists of $W|R$ transations. Each transaction executes randomly generated read and write operations. Data contention between transactions is very high in this experiment, and rollbacks are relatively common. If some transaction is rolled back, it must be restarted. As a result, $W|R$ transaction may start several times before it is committed. We vary relative second phase length in this experiment, which is measured as a ratio of number of operations at the second phase to the total number of operations. As in the previous experiment, it stays fixed on each run.

$W|R$ transactions are processed as usual updaters according to DFV and S2PL, and relative second phase length does not have an impact on the throughput. Be-

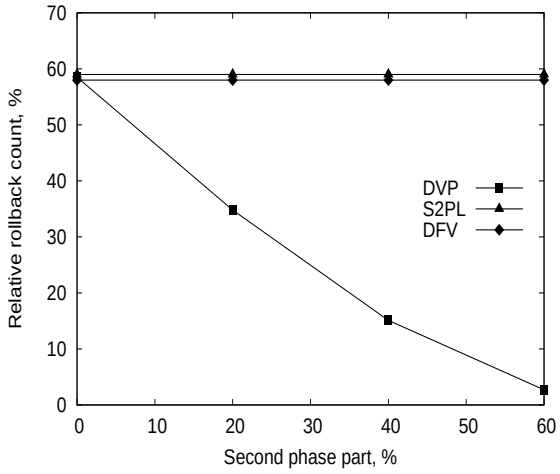

Figure 6: Relative rollback count

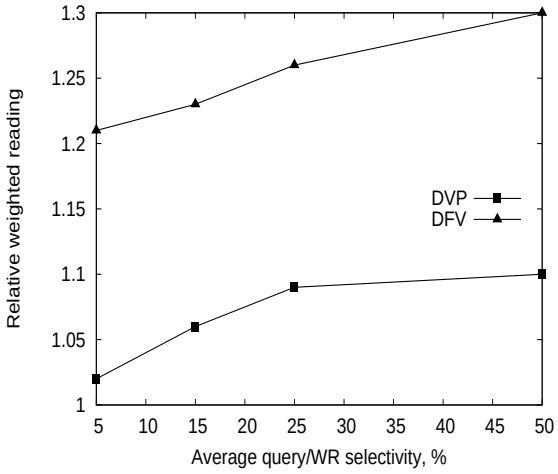

Figure 7: Weighted reading

cause of severe data contention, transactions experience frequent rollbacks and delays under S2PL and DFV. As a result, throughput of $W|R$ transactions for this protocols is not high. That is not an issue for DVP. First of all, as on the second phase $W|R$ transactions do not acquire any read locks, they would not experience any unnecessary delays. Second, as we stated in first theorem of Section 3.5, $W|R$ transactions do not experience deadlocks at the second phase and, as a result, number of rollbacks dramatically decreases.

Relative number of rollbacks is shown in Figure 6. It is measured as a ratio of number of rollbacks to total number of transactions. If some transaction is rolled back several times, only one rollback counts. As we can see, under S2PL and DFV transactions are indeed rolled back often. On the other hand, under DVP number of rollbacks is significantly lower, and at 60% length rollbacks are almost eliminated.

5.3 Experiment 3: $W|R$ and queries

In this section we evaluate the possibility of using $W|R$ transactions as queries. The main idea was described in Section 4. Workload consists 40% of read-only transactions and 60% of updaters. In this experiment we vary average selectivity of read-only transactions. In contrast to the first experiment, for each run selectivities of different transactions are uniformly distributed around some fixed point. For each such point we conduct two tests using DVP protocol. In the first one read-only transactions are represented as $W|R$ transactions, and in the second they are executed as queries. In this experiment we also study the effect that high number of $W|R$ transactions has on throughput of updaters. This effect was described in Section 4. Again, updaters in this experiment are relatievely short (with several write operations). This allows us to avoid conflicts between them, and we can more accurately evaluate relationships between updaters and queries/$W|R$ transactions.

First of all we want to discuss the weighted reading of versions. This parameter allows us to evaluate the novelty of read versions. Weighted reading is a ratio of weighted version count (WVC) to total number of read pages. Weighted version count increases with each reading of version:

$$WVC = WVC + \begin{cases} 1 & \text{for LCV version} \\ 2 & \text{for version next to LCV} \\ \dots & \\ n & \text{for } n_{th} \text{ version} \end{cases}$$

In some protocols information about versions is stored as some sort of distributed list: version keeps information about the next version. In this case WVC would define real number of read pages, and weighted reading would measure an average number of readings per one read operation. Again, for updater value of this parameter is equal to 1, because it always reads LCV versions. Figure 7 shows us the results. In this case, due to the nature of WVC, the smaller the value, the better. We can see here that dynamic approach leads during this experiment. The cause is that queries are bound to snapshots, while dynamic approach takes into consideration current situation. Also, we can see that after 25% mark, weighted access for $W|R$ transactions grows much slower than for queries. This is caused by slow update rates of snapshots: long queries with large selectivities keep snapshots busy and prevent them from update. As a result, queries read much more obsolete versions. For mentioned above protocols with distributed version information storing using queries would yield an additional readings per operation to find suitable version.

The next parameter we want to discuss is throughput of updaters. In Section 4 we mentioned the main problem that can occur when $W|R$ transactions are used as queries: updater's delays. Consider Figure 8, which shows throughput results. As we can see, queries have no impact on updaters processing. With queries, updaters always have versions for replacement. So they experince almost no delays, except that are caused by conflicts with another updaters. $W|R$ transactions are a different matter though. When average selectivity is not very high, updater throughput is almost equal to the throughput with queries. This is because length of $W|R$ transactions does not cause any significant delays yet. But then situation changes. With the increase of average selectivity, $W|R$ transactions become longer. As a result they can keep busy all the versions of some data items for some amount of time, and updaters have to wait for replacement versions. Similar situation would occur with any nonversion

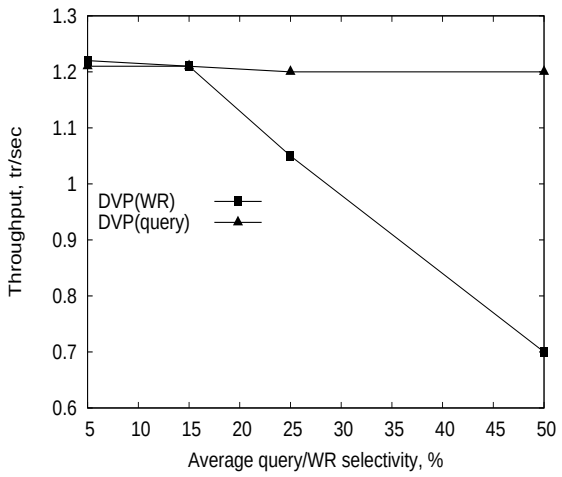

Figure 8: Updater throughput

protocol, such as S2PL, when updaters would wait for other transactions. Here, we must choose between novelty of read versions and possible deterioration of concurrent updaters. Different systems can require different approaches.

This last experiment clearly shows benefits of our dynamic approach. $W|R$ transactions read more recent versions, because they are not bound to some statical structures, like snapshots. Hence, they can yield significant benefits to read-only transactions. If performance of updaters is an important issue (i.e. in OLTP systems), our protocol offers an alternative: snapshot queries.

6 Conclusion

In this paper we presented a new multiversion approach, called dynamic versioning protocol (DVP), which supports efficient processing of $W|R$ transactions. DVP allows $W|R$ transactions to release all their read locks before start of the second phase, and, as a result, during this phase they execute read operations without taking new read locks and cannot experience deadlocks. Our protocol uses dynamic approach to determine which versions to read on the second phase. This approach is based on evaluation of current situation and discovering existing conflicts, which results in reading much more recent versions. For queries, DVP uses dynamically derived snapshots, which are obtained and advanced without need to interrupt transaction processing. One of the important features of our protocol is support of STEAL policy, which allows dirty pages to be written back on disk without need to wait for commit. Also, DVP maintains fixed number of versions, which greatly simplifies version management and reduces storage overhead.

Then we discussed some interesting application of $W|R$ transactions to query processing. The main idea is that $W|R$ transactions with empty first phase can represent read-only transactions. The main benefit of such approach is that $W|R$ transactions generally read more recent versions than queries. However, such processing may result in slight decrease of performance, because $W|R$ transaction can experince rare delays during the second phase. Thus, DVP allows choosing between performance with queries and reading more recent data with $W|R$ transactions.

We conducted experimental evaluation of DVP. First of all, processing of second phase of $W|R$ transactions without read locks can yield significant benefits in $W|R$ throughput. Furthermore, the third experiment shows that our dynamic approach results in selecting much more recent versions for reading. This confirms that DVP can provide an efficient solution to the problem of processing of $W|R$ transactions.

Finally, we proved the correctness of our protocol. Correctness means that DVP allows serializable execution of concurrent transactions, which is a very important feature of our protocol.

References

[1] R. Bayer, H. Heller, and A. Reiser. Parallelism and recovery in database systems. *ACM Trans. Database Syst.*, 5(2):139–156, 1980.

[2] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD Conference*, pages 1–10, 1995.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[4] P. M. Bober and M. J. Carey. Multiversion query locking. In *VLDB*, pages 497–510, 1992.

[5] P. M. Bober and M. J. Carey. On mixing queries and transactions via multiversion locking. In *In ICDE 1992, Tempe, Arizona*, pages 535–545. IEEE Computer Society, 1992.

[6] A. Chan, S. Fox, W.-T. K. Lin, A. Nori, and D. R. Ries. The implementation of an integrated concurrency control and recovery scheme. In *SIGMOD Conference*, pages 184–191, 1982.

[7] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Trans. Software Eng.*, 11(2):205–212, 1985.

[8] D. DuBourdieux. Implementation of distributed transactions. In *Berkeley Workshop*, pages 81–94, 1982.

[9] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[10] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Trans. Database Syst.*, 7(2):209–234, 1982.

[11] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[12] H. V. Jagadish, I. S. Mumick, and M. Rabinovich. Asynchronous version advancement in a distributed three-version database. In *In ICDE 1998, Orlando, Florida, USA*, pages 424–435. IEEE Computer Society, 1998.

[13] F. Llirbat, E. Simon, and D. Tombroff. Using versions in update transactions: Application to integrity checking. In *VLDB*, pages 96–105, 1997.

[14] A. Merchant, K.-L. Wu, P. S. Yu, and M.-S. Chen. Performance analysis of dynamic finite versioning for concurrency transaction and query processing. In *SIGMETRICS*, pages 103–114, 1992.

[15] C. Mohan, H. Pirahesh, and R. A. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *SIGMOD Conference*, pages 124–133, 1992.

[16] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23, 1983.

[17] R. E. Stearns and D. J. Rosenkrantz. Distributed database concurrency controls using before-values. In *SIGMOD Conference*, pages 74–83, 1981.

[18] A. Thomasian. Performance limits of two-phase locking. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*, pages 426–435. IEEE Computer Society, 1991.

[19] W. E. Weihl. Distributed version management for read-only actions. *IEEE Trans. Software Eng.*, 13(1):55–64, 1987.

[20] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[21] K.-L. Wu, P. S. Yu, and M.-S. Chen. Dynamic finite versioning: An effective versioning approach to concurrent transaction and query processing. In *ICDE*, pages 577–586, 1993.

A Proof of theorems

In this section we will prove two theorems from Sections 3.5 and 3.6. We will be using elements of standard multiversioning theory, which can be found in [3]. First, we will prove correctness of our protocol by showing that all possible multiversion serialization graphs (MVSG) are acyclic. Then, we will prove the fact, that $W|R$ transactions cannot experience deadlocks under DVP.

First of all, we can define precedence relation between every two versions of some data item x. x_i precedes x_j ($x_i \prec x_j$) iff x_i was created before x_j. This order is well-defined because DVP processes updaters according to the S2PL rules. In this case x_j can be created only when x_i has already been committed.

Consider MVSG G. Its vertices represent transactions of three different types: $W|R$, W (updater) and R (query). It also can contain edges of three types:

1. $T_i \xrightarrow{1} T_j$. That means that T_j reads version of some data item, created by T_i.
2. Consider T_i, T_j and T_k, where $i \neq j \neq k$. T_i creates x_i that precedes x_j created by T_j. Let T_k read x_i. In this case this edge is drawn: $T_k \xrightarrow{2} T_j$.
3. Consider T_i, T_j and T_k, where $i \neq j \neq k$. T_i creates x_i that precedes x_j created by T_j. Let T_k read x_j. In this case this edge is drawn: $T_i \xrightarrow{3} T_j$.

First we examine some of the properties of MVSG under DVP. We formulate them in the form of lemmas. In this lemmas, FS_i denotes $FollowSet$ of transaction T_i. o_i defines commit operation for transactions of W-type and $W|R$-type. For transactions of R-type it defines its first read operation. Operations are processed in well-defined order. This implies that for each pair of operations o_i and o_j, either $o_i < o_j$ or $o_i > o_j$.

Lemma 1. *Let $T_i \xrightarrow{1} T_j \xrightarrow{2} T_k$, where T_j is of R-type. Then, $o_i < o_k$.*

Proof. This follows from procedure of snapshot's creation. According to this procedure T_i must be committed before snapshot's creation, and T_k is committed after that. Otherwise, its versions would have belonged to the snapshot, and $T_j \xrightarrow{2} T_k$ wouldn't have been possible. □

Lemma 2. *This lemma consists of two parts:*

1. *Let $T_i \xrightarrow{1,3} T_j$. Then, $o_i < o_j$.*
2. *Let $T_i \xrightarrow{2} T_j$, where T_i is not of $W|R$-type or R-type. Then, also, $o_i < o_j$.*

Proof. This property follows strictly from S2PL locking rules. □

Lemma 3. *Let $T_i \rightarrow T_j \rightarrow T_k$, $T_j \in FS_i$ and T_k is not of R-type. Then, if $o_k < o_i$, then $T_k \in FS_i$.*

Proof. T_k conflicts indirectly with T_i through T_j. DVP handles such conflicts by putting T_k in FS_i. □

Lemma 4. *Let $T_i \in FS_j$. Then situation $T_i \rightarrow T_j$ is not possible.*

Proof. Examine the possibility of edges of different types:

1. $T_i \xrightarrow{1} T_j$ is not possible, because transaction cannot read version, created by transaction from its $FollowSet$.
2. $T_i \xrightarrow{2} T_j$ is not possible. If T_i is of W-type, then it would have committed before corresponding write operation of T_j and could not have belonged to FS_j. If T_i is of $W|R$-type, then, according to DVP rules, it would have been blocked until completion of T_j. In this case it would have read version created by T_j.
3. $T_i \xrightarrow{3} T_j$ is similar to the first part of the previous case: T_i would have committed before corresponding write operation of T_j.

□

Lemma 5. *Let $T_i \xrightarrow{1} T_j$, where T_j is of R-type and $T_i \in FS_k$. Then, situation $T_j \xrightarrow{2} T_k$ is not possible.*

Proof. In this case we have active $W|R$ transaction T_k with some transaction in its $FollowSet$. In this case snapshot could not have been obtained, according to DVP rules. So situation $T_j \xrightarrow{2} T_k$ is not possible. □

Lemma 6. *Let $T_i \to \ldots \to T_j \to T_k \to T_l$, where T_k is of R-type and $T_j \in FS_i$. Then $o_i < o_l$.*

Proof. Since T_k reads snapshot, and versions from T_k are in this snapshot, then T_i would have committed before snapshot's creation. This is ensured by DVP rules for snapshot creation. But versions from T_l are not in this snapshot, and that means T_l would have committed after snapshot's creation. This implies $o_i < o_l$. □

Lemma 7. *If there is a cycle in MVSG, it cannot contain any $W|R$ transactions.*

Proof. Let's assume that cycle with $W|R$ transactions exists: $T1 \to T2 \to \ldots \to T_{i_1} \to \ldots \to T_{i_2} \to \ldots \to T_{i_k} \to \ldots \to T_n \to T1$. Consider three general cases:

1. $o_1 < o_2 < \ldots < o_{i_1}$; T_{i_1} is a $W|R$ transaction and $T_{i_1+1}, \ldots, T_{i_2-1} \in FS_{i_1}$. Then, again, $o_{i_1} < o_{i_2}$ (notice, that we also have $o_1 < o_{i_2}$ here), and $o_{i_2} < o_{i_2+1}$ and so on (we use Lemmas 1-3 here). Similar to $o_1 < o_{i_2}$, we can derive that $o_1 < o_{i_1} < \ldots < o_{i_k}$. Then we have following possibilities:
   (a) $T_n, \ldots, T_{i_k+1} \in FS_{i_k}$. In this case we also have $o_1 < o_{i_k}, o_2 < o_{i_k}, \ldots, o_{i_1} < o_{i_k}$. Then, $T_1, T_2, \ldots, T_{i_1} \in FS_{i_k}$ (Lemma 3). But this implies that $T_{i_1}, \ldots, T_{i_2-1} \in FS_{i_k}$, and so on. We can receive only situations, described in Lemmas 4-6 here: Lemma 4 ($T_{i_k-1} \in FS_{i_k}$), Lemma 5($T_{i_k-2} \in FS_{i_k}$ and T_{i_k-1} is of R-type) and Lemma 6(for example, for some R transaction T_k where $k \in [1; i_1]$; in this case we would receive $o_{i_k} < o_{k+1}$, but at the same time $o_{k+1} < o_{i_k}$ in our general case).
   (b) $o_{i_k} < o_{i_k+1} < \ldots < o_n$ and T_n is of W-type or $W|R$-type. In this case we can derive that either $o_n < o_1$ (T_n is of W-type; or T_n is $W|R$ transaction and $T_n \xrightarrow{1,3} T_1$) or $T_1 \in FS_n$ (T_n is $W|R$ transaction and $T_n \xrightarrow{2} T_1$). In the first case we already have $o_1 < o_n$, which contradicts to $o_n < o_1$. The second case is equal to 1a, where $i_k = n$.
   (c) $o_{i_k} < o_{i_k+1} < \ldots < o_{n-1}$ and T_n is of R-type. In this case we can derive that $o_{n-1} < o_1$ (Lemma 1). But at the same time we have $o_1 < o_{n-1}$ - contradiction.
   (d) $T_{i_k+1}, \ldots, T_{n-1} \in FS_{i_k}$ and T_n is of R-type. In this case $o_{i_k} < o_1$. But we already have $o_1 < o_{i_k}$ - contradiction.
2. $T_2, \ldots, T_{i_1-1} \in FS_1$, $o_1 < o_{i_1} < o_{i_1+1} < \ldots < o_{i_2}$ and so on. This case is similar to the first general one with the same four possibilities. In fact, we can consider it as the first general case with $T_{i_1} = T_1$.
3. $T_2, T_3, \ldots, T_n \in FS_1$; or $T_2, T_3, \ldots, T_{n-1} \in FS_1$ and T_n is of R-type. First case contradicts with Lemma 4. Second case implies $o_1 < o_1$ (Lemma 6), which is also impossible.

Not one of this three general cases is possible, and they include all possible cycles with $W|R$ transactions. So we can derive that such cycles are prohibited. □

Lemma 8. *If there is a cycle in MVSG, it cannot contain any W transactions.*

Proof. Consider possible cycle in MVSG: $T_1 \to T_2 \to \ldots \to T_n \to T_1$, where T_1 is of W-type. First of all, according to the previous lemma, it cannot contain any $W|R$ transactions. So this cycle can contain only transactions of W-type and R-type. According to Lemmas 1 and 2, we can only derive that $o_1 < o_1$, which is impossible. So cycles with W transactions are prohibited. □

Theorem 3. *DVP guarantees serializability of all transactions.*

Proof. According to the previous two lemmas, if there existed any cycle in MVSG, then it could not contain W and $W|R$ transactions. But in this case we could receive cycle containing only R-type transactions, which is impossible, because R transactions cannot be adjacent to each other in MVSG. □

Next, we will prove that $W|R$ transactions cannot experience deadlock situations on the second phase.

Lemma 9. *Let $T_i \in FS_j$. Then $T_j \notin FS_i$.*

Proof. If $T_i \in FS_j$, then, according to DVP, in the situation that could lead to $T_j \in FS_i$, T_i would be blocked until completion of T_j. So it cannot occur that $T_j \in FS_i$. □

Theorem 4. *$W|R$ transactions cannot experience deadlocks during the second phase.*

Proof. Since $W|R$ transactions, according to DVP, can wait for only another $W|R$ transactions, deadlock cycle can contain only $W|R$ transactions. Consider such cycle: $T1 \to T2 \to \ldots \to T_n \to T1$. If T_i waits for T_j, then $T_i \in FS_j$. But for our cycle we would receive that $T_n \in FS_1$ and $T_1 \in FS_n$, which contradicts with Lemma 9. So such deadlock cycles are prohibited. □