

Update Propagation of Replicated Data in Distributed Spatial Databases

Jin Oh Choi¹, Young Sang Shin², Bong Hee Hong²

¹ Department of Computer Engineering, Kyungdong University, Pongpo Li, Tosung Myun, Gosung Goon, Kangwon Do, South Korea
jochoi@kyungdong.ac.kr

² Department of Computer Engineering, Pusan National University, Jangjun Dong, Kumjung Goo, Pusan, South Korea
{yosshin, bhhong}@hyowon.cc.pusan.ac.kr

Abstract. When spatial objects are replicated at several sites in the network, the updates of a long transaction in a specific site should be propagated to the other sites for maintaining the consistency of replicated spatial objects. If any two or more transactions at different sites concurrently update some spatial objects within a given region, two spatial objects having spatial relationships should be cooperatively updated even if there are no direct conflicts of locking for them. This paper deals with the problems of replication control of spatial objects. We present the concepts of *Region locking* and *Spatial Relationship-Bound Write locking* for enhancing parallelism of updating the replicated spatial objects. If there are no spatial relationships between the two objects that are concurrently being updated at different sites, parallel updates will be completely allowed. We argue that concurrent updates of two spatial objects having spatial relationships should be propagated and cooperated by using an extended two-phase commit protocol, called *Spatial Relationship-based 2PC protocol*.

1 Introduction

The update of replicated spatial objects is an interactive transaction that requires cooperative work with others. The update dependency between two interactive transactions cannot be modeled by the existing locking techniques for concurrency control, rather it is based on the user interaction with the two transactions.

The interactive updates of two replicated spatial objects at different sites should be synchronized for concurrency control. In the interactive transactions, it is very difficult to define the correctness criteria of concurrent transactions since the displayed spatial objects cannot be isolated due to their spatial relationships. We define a *distributed spatial relationship* as the binary spatial relation, for example, disjoint, meets, equals, inside, covers, or overlaps, between two spatial objects which are stored at different sites. Locks on two spatial objects do not conflict with each other, since they are not copies. However, concurrent updates of two spatial objects sometimes can make them inconsistent when they have a *distributed spatial relationship*.

In the replicated spatial database, the characteristics of interactive transactions make it difficult to exploit the traditional replication control approaches. As a pessimistic approach, the existing locking-based replication control approach has the following problems. First, locking objects in a long transaction makes other transactions wait for a long time. Second, an interactive transaction that updates spatial objects, has to lock the entire data set or at least a layer, because the replicated spatial objects should be able to be displayed for interactively updating any objects in a long transaction. Third, if two objects have a *distributed spatial relationship*, their concurrent update should be restricted. For example, if the boundary of a spatial object X is shared with the other spatial object Y, a transaction to update Y should be forced to wait until the update of X is completed.

An optimistic approach, like the multi-version control approach [13], allows concurrent updates on the replicated data at several sites, and then, merges the results together. Independent updates of their own data sets at each site will cause conflicts between them; therefore, resulting in the inconsistent states, when they are merged, calling for the need of rollback in the long transactions.

To deal with the issues of concurrent updates of replicated spatial data, we propose *region locking* and *spatial relationship-bound write locking*, as new locking concepts. We argue that new locking primitives should be introduced to achieve high concurrency and to control the consistency of replicated spatial data. *Region lock* is an extension of the shared lock, which provides a weak READ lock for a group of replicated spatial objects. The *region lock* allows a new long transaction to start at any time without waiting. The possible conflicts of concurrent updates of replicated data are filtered by *spatial relationship-bound write locking* during the execution of interactive transactions. The *spatial relationship-bound write lock* is an extension of the exclusive lock to model the update dependency between two interactive transactions due to *distributed spatial relationships*. The *spatial relationship-bound write locking* allows the objects not having any *distributed spatial relationships* to be concurrently updated.

We have introduced a new cooperative update protocol, which is designed on the basis of the existing two-phase commit protocol. The basic protocol of the extended 2PC is the same with that of the existing 2PC except that the decision on collaborative updates or independent updates is based on *distributed spatial relationships*. This protocol is named, *spatial relationship-based 2PC*.

This paper is organized as follows. In section 2, we will briefly describe related works. In section 3, we address the locking problems of spatial objects, which have a *distributed spatial relationship*. To deal with the issues of concurrent updates of replicated spatial data, new locking methods are introduced in section 4. Section 5 presents the update propagation protocol based on the distributed spatial relationship-based locking. Section 6 describes an overview of our system implemented on top of a GIS S/W, Gothic. Our conclusions are presented in section 7.

2 Related Work

In distributed databases, replication consistency can be maintained by the synchronous [11] or asynchronous [5] replica control scheme. Synchronous replica control keeps all replicas synchronized at all the sites by the 2PC protocol. Asynchronous replica control propagates replication updates asynchronously to the other sites after committing on a replica-server. In addition, there has been much research to release restriction of synchronous replica control, such as a quorum-based scheme, causality [2][9][12].

The optimistic approach, such as the lazy replication scheme [8], which belongs to the asynchronous replica control method, allows an object to be independently updated at each site. In this approach, locking is not used. Instead, the multi-version concept [13] is employed to control concurrency and ensure the serializability. Concurrent updates of two spatial objects having *distributed spatial relationships* may make them inconsistent even if their READ locks or WRITE locks do not conflict with the other. Thus, the traditional optimistic scheme is difficult to be applied on replica control of spatial objects.

For the increase of the concurrency of long transactions, we have developed a new protocol, named, the *mid-commit protocol* based on the existing 2PC [1]. The main premise of our earlier work was to use the delta-merge protocol to resolve the update conflict problem of long transactions. The work on this paper is an extension of the *mid-commit protocol* for supporting replica control and concurrency of long transactions, which can guarantee the serializability of concurrent updates of spatial objects that are replicated.

3 The Locking Problems of Spatial Objects

We will describe the problems of concurrently updating replicated spatial objects, and identify update dependencies among interactive transactions to support the replication control and concurrency control of replicated data.

Fig. 1 shows a scenario of updating two spatial objects. Two interactive transactions, T_A and T_B , update replicated data, 'property' and 'road' respectively. If T_A and T_B are executed sequentially, as Fig. 1 (a) or (b), these serially scheduled transactions preserve a correct state.

Not all concurrent execution of long transactions result in a correct state. Consider the schedule of Fig. 1 (c). Since the WRITE locks (property 1, property 2) of T_A do not conflict with the WRITE lock (road 1) of T_B , the locking protocol does not delay any of two transactions. However, the schedule (c) leads to an incorrect state. The schedule leads to an undesirable result because two objects, property 1 and road 1, are independently updated in spite of having a spatial relationship, 'meets'.

Because of the possibility of giving an incorrect state, two spatial objects having a spatial relationship should not be updated concurrently. This is an update constraint of two different spatial objects, which have a spatial relationship. The traditional locking

protocol can not ensure the serializability of concurrent updates of two spatial objects with the dependency due to this spatial relationship.

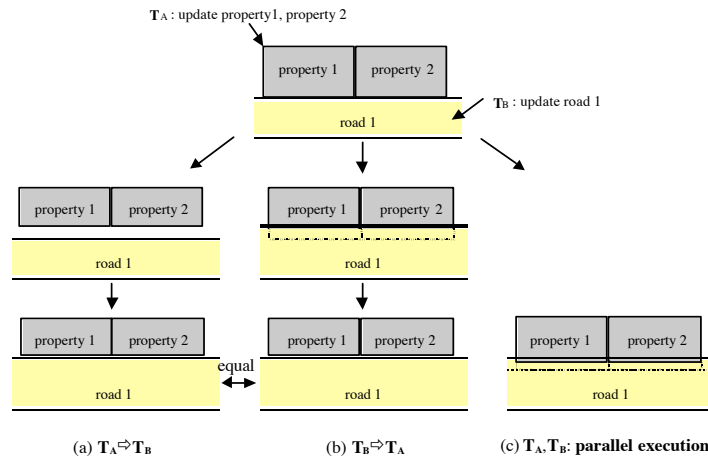


Fig. 1. Updating the objects that have spatial relationships

A spatial relationship is defined as a relationship between two spatial objects having Egenhofer's spatial relations. In [6], Egenhofer classified spatial relationships into 8 types, Disjoint, Meets, Equals, Inside1, Inside2, Covers1, Covers2, and Overlaps. Fig. 2 shows an example of a spatial relationship, 'Inside'.

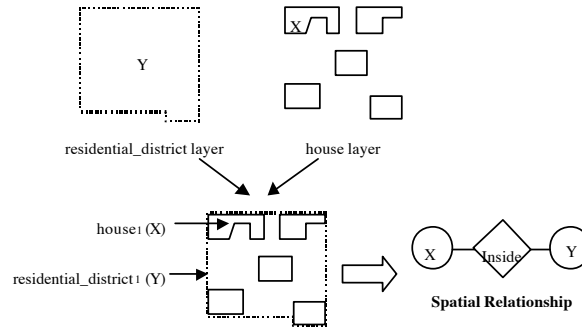


Fig. 2. An example of a spatial relationship 'Inside'

A spatial relationship dependency can also be defined for remote spatial objects. Now, we define *distributed-SR dependency* as follows:

Definition 1. Two objects, O_i and O_j are *distributed-SR dependent* if and only if two objects have a spatial relationship except 'Disjoint' relation, and where O_i and O_j are the objects to be updated by T_i and T_j at remote sites S_i and S_j respectively.

If two objects are *distributed-SR dependent*, concurrent update of them does not guarantee a correct state. For example, in Fig. 2, if X and Y are individually updated at different sites, they are *distributed-SR dependent*. Because X and Y are replicated across two sites, parallel updates of them also may produce an incorrect state shown in Fig. 1. Therefore, when two objects, O_i and O_j , are *distributed-SR dependent*, two transaction, T_i and T_j , that update two objects, must update them cooperatively.

4. Region Locking and SR-Bound Write Locking

We propose *region locking* and *SR-bound write locking*, which are extensions of two-phase locking. The key idea upon which the extensions are based, is to restrict the unit of concurrency control of interactive transactions to a window of spatial objects displayed on the screen.

4.1 The Definition of Region Locking and SR-Bound Write Locking

Region locking sets shared locks on all the objects within the region that is interactively defined by users. *Region lock* is a weak shared lock because the lock allows us to set WRITE locks on some objects in the region later on. This lock mode is similar to *share intention exclusive* lock (*SIX* lock) in multiple granularity locking. When a replicated spatial object is being updated at a remote site, it is desirable to allow interactive transactions to be able to display it at the same time in order to access or update some spatial objects by interacting with displayed objects. The mode of a *region lock* is a *weak SIX lock*, and define it as follows:

Definition 2. A *weak SIX lock* is a lock mode that holds locks on a set of objects in the shared mode and allows the other transactions to acquire exclusive locks on some of the objects. It tolerates shared locks and *weak SIX locks* of other transactions.

The definition of the *region lock* using definition 2 is as follows:

Definition 3. Let D be the whole data set of a map, R be the entire region of the map, R_i be sub-region of R viewed by users who are running a long transaction T_i , and D_{R_i} be all the objects which is totally contained in R_i . We define the *Region lock* of T_i as a set of *weak SIX locks* on D_{R_i} .

(** Fig. 3 shows an example of a *region lock*. D_{R_i} are the objects totally contained in R_i (that is, 1 road segment and 5 properties). The region, R_i , is defined by users for updating some objects of D_{R_i} . *Region locking* sets *weak SIX locks* on D_{R_i} .

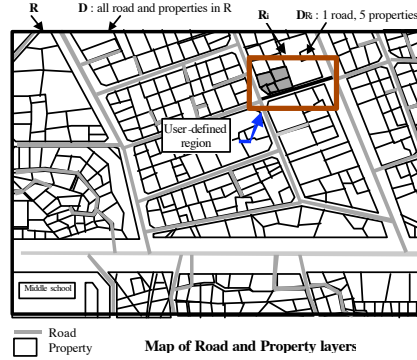


Fig. 3. An example of *region locking* (**)

If there exists a *distributed-SR dependency* between two objects at different sites, concurrent update of the two objects should not be allowed. We introduce a new WRITE lock mode based on *distributed-SR dependency*, and is defined as follows:

Definition 4. A *DSRX lock* (Distributed Spatial Relationship-bound eXclusive lock) is a lock mode that sets the exclusive lock to the remote objects being *distributed-SR dependent* on the locally updated object, and also holds an exclusive lock on it. It also tolerates shared locks and *weak SIX locks* of other transactions.

The definition of *SR-bound write lock* using *DSRX lock* is as follows:

Definition 5. Let X be a group of objects in a *region lock* of a transaction T_i . We define *Spatial Relationship-bound Write locks* of T_i as *DSRX locks* on X .

Table 1. Compatibility Matrix for two-phase locks, *weak SIX* and *DSRX*

	READ	WRITE	weak SIX	DSRX
READ	yes	no	yes	yes
WRITE	no	no	no	no
weak SIX	yes	no	yes	yes
DSRX	yes	no	yes	no

Lock compatibility matrix is extended to include *weak SIX lock* and *DSRX lock* as shown in Table 1. In the lock compatibility of Table 1, *weak SIX lock* and *DSRX lock* modes are compatible with the READ lock mode.

4.2 Concurrency Control by Region Locking

We name the *region lock* of a transaction T_A , as RGL_A . When RGL_A , the *region lock* of transaction T_A , and a RGL_B , the *region lock* of transaction T_B , are sent to remote sites, two *region locks* might have eight kinds of relations, like *distributed spatial relationships*. Except 'Disjoint' relationship, two *region locks* have non-disjoint area, denoted as NDJ_{AB} (Non-DisJoint area of T_A and T_B). In the case of 'Overlaps', NDJ_{AB} is the overlapped area ($RGL_A \cap RGL_B$) as shown in Fig. 5. For 'Meets' and 'Equals', NDJ_{AB} is the union of two regions ($RGL_A \cup RGL_B$). If two *region locks*, RGL_A and RGL_B , have NDJ_{AB} , we say, two transactions, T_A and T_B , are in the relation, region-NDJ. If two *region locks* don't have NDJ area, we say, two transactions, T_A and T_B , are in the relation, region-DJ.

(**)

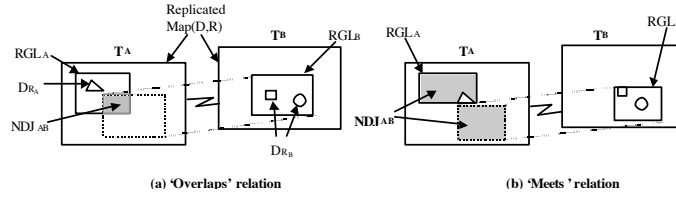


Fig. 4. Topology of *region locking* (**)

In this paper, we assume that T_A can only update the objects in D_{R_A} after acquiring SR (*Spatial Relationship*)-*bound write lock* and there is only one transaction at each site at a time. An object, which is updated by T_A and is set to SR -*bound write lock*, is called UDO_A (*UpDating Objects* of T_A , $UDO_A \subset D_{R_A}$).

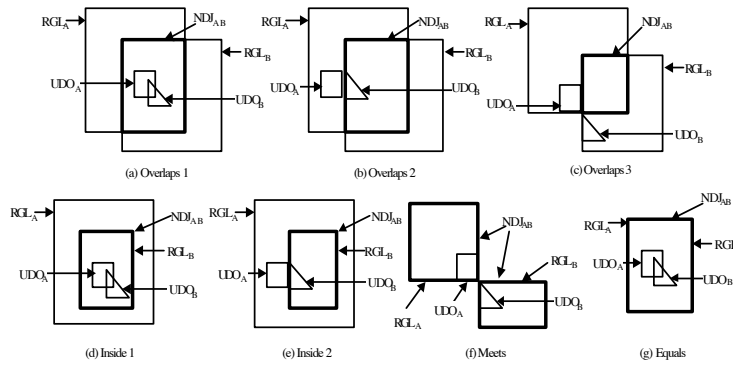


Fig. 5. NDJ_{AB} and update conflicts

If there exists a NDJ area between two *region locks*, update conflict can occur between the two interactive transactions while updating the two regions. Fig. 5 shows

the examples of *distributed-SR dependencies* of region locks, when two *region locks* are non-disjoint.

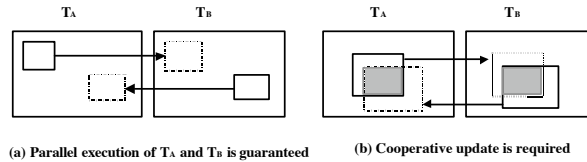


Fig. 6. Concurrency control by *region locking*

We use *region locks* as a means of synchronizing the read access of a group of replicated spatial objects. In Fig. 6 (a), the two transactions, T_A and T_B , can be executed concurrently, because there is no *distributed-SR dependency* between two *region locks*. However, in Fig. 6 (b), concurrent execution of the two transactions must be forbidden, because of the *distributed-SR dependency* between the two *region locks*.

(** *Region locking* does not always limit the concurrent execution of two transactions having *distributed-SR dependency*. Some transactions, which have a *NDJ* between their two *region locks*, can be executed at the same time without affecting each other. Fig. 7 (a) shows that if UDO_A and UDO_B are not *distributed-SR dependent*, update conflict may not occur even if these are updated concurrently. Only when there are the *region lock conflict* and *distributed-SR dependency* between two transactions, concurrent execution of them should be delayed until they obtain exclusive locks on the object to update.

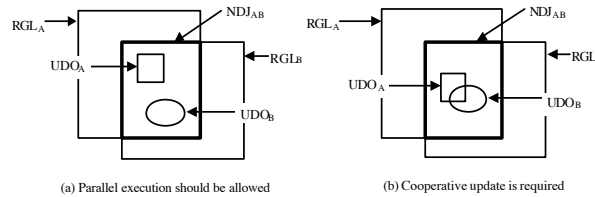


Fig. 7. Concurrency control based on *distributed-SR dependency* (**)

4.3 Concurrent Update Using *SR-Bound Write Locking*

We will describe an extension of the existing cooperative transaction model for increasing concurrency of updating spatial objects. A newly defined cooperative spatial transaction model introduces several new transaction operators as shown in Fig. 8. A transaction T_A sets *region lock* and sends it to remote sites (*set-region-lock* operation). When a user points or clicks an object to update it in the *region lock*, T_A sends a lock request message to remote sites to acquire *SR-Bound Write lock* (*set-*

SR-bound write lock on the object updated by T_A , whether T_B will ‘wait’ or ‘continue’, is determined by examining the *distributed-SR dependency* between UDO_A and UDO_B . Even if two *region locks* of T_A and T_B are non-disjoint (NDJ_{AB}), concurrent update can be allowed if there is no *distributed-SR dependency* between *Updating Objects* (UDO_A and UDO_B) (Fig. 9). When there exists *distributed-SR*

dependency, the waiting time of a transaction is limited to the duration of one update cycle (Fig. 9).**)

4.4 A Change in the Distributed-SR Dependency

During execution of a transaction, *distributed-SR dependency* between two spatial objects can be dynamically deleted or created according to the changes of geometry. Thus, the concurrency control using *SR-bound write locking*, should reflect the runtime change of *distributed-SR dependency*.

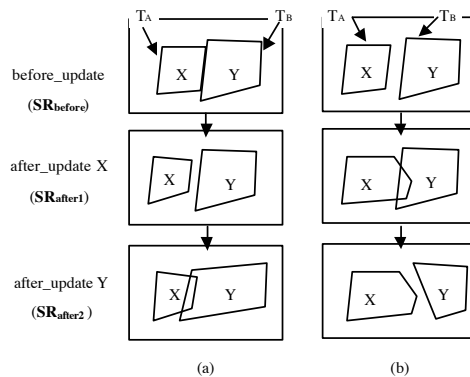


Fig. 10. The changes of *distributed-SR dependencies*

Here, we define SR_{before} as a *distributed spatial relationship* before update, and SR_{after} as a *distributed spatial relationship* after update. In Fig. 10 (a), the mode of *distributed-SR dependency* is set to *independent* from *dependent* by updating of the geometry of X. Fig. 10 (b) shows the reverse case of Fig. 10 (a). In these cases, the correctness of updated map cannot be automatically determined. An approach to address and find solutions to the issues of concurrent control of interactive transactions, is to use the 2PC protocol for supporting cooperative transactions.

(** We assume that T_A is executed before T_B , and also, when T_A is executed in parallel with T_B , T_A is committed before T_B . When a *distributed spatial relationship* at SR_{before} or SR_{after} is *dependent*, the simple merge of two transactions may lead to an incorrect state. Therefore, the *mid-commit* of a transaction should be done in the 2PC for merging. The *mid-commit* protocol also should be applied to temporary SR-independent state, as in Table 2 (b), (c), and (d). In addition, when a *distributed-SR dependency* is newly created after carrying out the *mid-commit* operation of a transaction, T_A , the second transaction, T_B , should cooperate with T_A in synchronizing the result of T_A with that of T_B , as in Table 2 (e), (f), (g).

Table 2. Transaction action table based on *Distributed-SR dependency*

	before T_A updates (SR_{before})	T_B acts	after T_A updates (SR_{after1})	T_B acts	after T_B updates (SR_{after2})	T_B acts
(a)	dependent	wait	dependent	2PC with T_A	dependent	2PC with T_A
(b)					independent	2PC with T_A
(c)			independent	2PC with T_A	dependent	2PC with T_A
(d)					independent	2PC with T_A
(e)	independent	parallel update	dependent	2PC with T_A	dependent	2PC with T_A
(f)					independent	2PC with T_A
(g)			independent	parallel update	dependent	2PC with T_A
(h)					independent	No 2PC (**)

5. Update Propagation Scheme

In this section, we will discuss an *SR-based 2PC protocol* to propagate an update of any given object to all secondary copies. We will illustrate an example of update propagation scenarios.

(** 5.1 Transaction Operations

We extend the locking-based distributed transaction operations by including the *SR-based locking*. The transaction operations can be issued manually by users, or issued automatically by a system.

? *set-region-lock* to set a group of READ locks to all the objects contained within a user-specified area and notify the *region lock* to relevant remote sites.

- reply-region-NDJ to reply that a receiver has a relation, region-NDJ, over its source object.

- reply-region-DJ to reply that a receiver has a relation, region-DJ, over its source object.

? *set-SRBW-lock* to set an *SR-bound write lock* to an object to be updated within the area of the *region lock* and notify the *SR-bound write lock* to all the remote sites having a relation, region-NDJ, with its source site.

- reply-SRBW-conflict to reply that a receiver already has the *SR-bound write lock*, which conflicts with its source object, in response to *set-SRBW-lock*.
- reply-SRBW-ok to reply that a receiver doesn't have the *SR-bound write lock*, which conflicts with its source object, in response to *set-SRBW-lock*.
- ? *mid-commit* to propagate DELTA to all of the sites and start *SR-based 2PC*.
 - *reply-mid-accept* to reply "OK" to the sender.
 - *reply-mid-reject* to reply "Not OK" to the sender.
 - *send-global-commit* to send a global commit to all the sites, if all replies are "OK".
 - *send-global-abort* to send a global abort to all the sites, if any reply is "Not OK".
- ? *mid-rollback* to roll the most recent *mid-commit* back and to propagate it to all of the sites.
- ? *release-region-lock* to release the *region lock* and commit a transaction.
 - *pre-released* to inform of the start of the *release-region-lock* to the message sender.
 - *post-released* to inform of the end of the *release-region-lock* to the message sender.
- ? *release-SRBW-lock* to release the *SR-bound write lock* and awake the blocked transactions.
- ? *rcv-"any-operation"* when "any-operation" operation is propagated to remote sites, the receiver executes the *rcv-"any-operation"* operation. **)

5.2 The Algorithm of Transaction Operations

In this section, we describe the notification and release of *region locks* and *SR-bound write locks*, the propagation of mid-update (*mid-commit*), and the rollback of committed mid-update (*mid-rollback*). We define some terminology used for describing these algorithms as follows:

- ? *Coordinator* : a site to notify or propagate any transaction operation to participating sites.
- ? *all-sites* : all participating sites except a *Coordinator*.
- ? *Participants* : any participating site having a relation, region-NDJ, with a *Coordinator*.
- ? *Others* : any participating sites having a relation, region-DJ, with a *Coordinator*.

5.2.1 Notification and Release of *Region Lock*

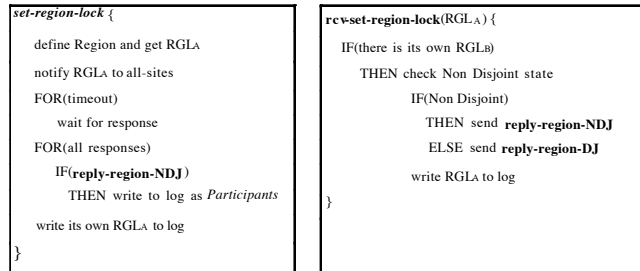


Fig. 11. The algorithm of *region locking*

The *set-region-lock* operation defines a *region lock*, notifying it to all-sites in order to get *Participants*, and writes these to a log, as shown in Fig. 11. The *rcv-set-region-lock* operation is automatically executed at the sites that receive the message *set-region-lock*. *rcv-set-region-lock* checks if there is a relation, region-NDJ, between a sender and a receiver, and returns an answer *reply-region-NDJ* or *reply-region-DJ* to the sender.

When there are no more objects to be updated under a *region lock*, the *release-region-lock*, shown in Fig. 12, will be invoked to release the *region lock* and finish the long transaction. However, if the transaction had any *Participants* after notifying the *set-region-lock*, the transaction should not be terminated independently. Some transactions having a relation, region-NDJ, should be coordinated by the extended 2PC, because their *distributed spatial relationships* may be dynamically changed according to their updates (note that, in Table 2 (g), the mode of SR_{after2} can be changed from 'independent' to 'dependent'). Therefore, the termination of the *release-region-lock* should be suspended until all *Participants* are finished.

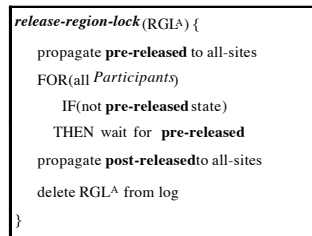


Fig. 12. The algorithm of releasing a *region lock*

5.2.2 Notification and Release of *SR-Bound Write Lock*

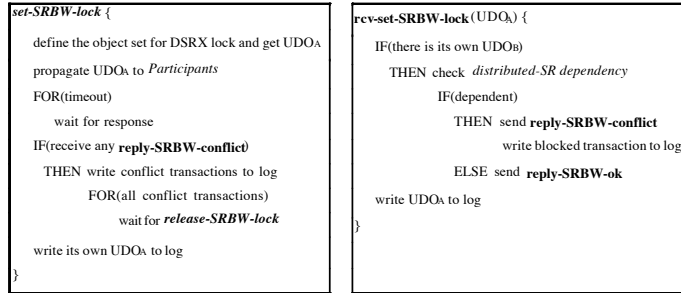


Fig. 13. The algorithm of *SR-bound write locking*

The *set-SRBW-lock* operation defines *UpDating Objects*(UDO), propagates them to all the *Participants* of which *region locks* may conflict with their sources, as shown in Fig. 13. Lock conflicts between a *Coordinator* and its *Participants* mean that *Participants* already hold the *SR-bound write lock* on some of UDO. In such cases, the *Coordinator* writes *Participants* to log, and should wait until the *SR-bound write lock* is released. *rcv-set-SRBW-lock* operation returns the message, *reply-SRBW-conflict* or *reply-SRBW-ok*, according to lock compatibility.

The release of the *SR-bound write lock* is done by the *release-SRBW-lock* operation. This operation is invoked after executing the *mid-commit*, and the lock release signal is notified to *Participants*.

5.2.3 SR-based 2PC

We have introduced the *mid-commit* operation to accomplish two purposes: replication control and collaborative work. First, we have to deal with the update propagation problem of replicated data in two or more long transactions. The idea is to decompose an interactive transaction into sub-transactions which contain only one update cycle each, and then perform update propagation incrementally. Second, the collaborative work is required to guarantee the correctness of long transactions. The collaborative work is performed by propagating the DELTA of an updated object to others.

For these purposes, we extend the traditional 2PC for implementing the collaborative work. The basic protocol of the extended 2PC is same with that of existing 2PC. However, the scope of participant sites communicating with the sender is determined by identifying *distributed-SR dependencies*. The extended 2PC, named as *SR-based 2PC*, is operated between a *Coordinator* and its SR-based *Participants*. In this paper, we assume that there is no communication failure and all the sites are always available to limit the scope of this paper.

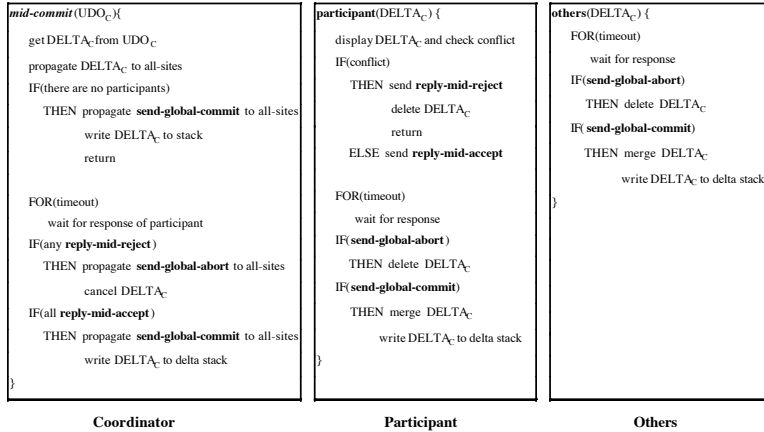


Fig. 14. The algorithm of *SR-based 2PC*

The *SR-based 2PC* operates among the *Coordinator*, *Participants*, and *Others*. The *Coordinator* is a site who issues the *mid-commit*. *Participants* are those updating the remote objects which are *distributed-SR dependent* or could be potentially *distributed-SR dependent* on the object being updated by the *Coordinator*. *Others* are the sites who are neither the *Coordinator* nor *Participants*. The algorithm of *SR-based 2PC* is shown in Fig. 14.

(5.2.4 mid-rollback**

After the updated object is propagated to all the remote sites, they may be canceled because of errors found later. We don't take the recovery problem caused by the failure of a communication link or a site into consideration, and concentrate on the cancellation of updates after propagation.

At first, we must determine how far the update transaction should be rolled back. It is undesirable to roll a whole long transaction back, since the cost of rollback is too large. Because the updated object is propagated after finishing 'an update cycle', the 'an update cycle' is not only the unit of *mid-commit* but also one of rollback.

Now, we focus on how the *mid-rollback* operation cancels the last update committed by the *mid-commit* operation. To ensure the global consistency, the *mid-rollback* operation must satisfy following requirements, to roll the most recent *mid-commit* back:

- ? all sites must maintain their own DELTA stacks to return to the previous state.
- ? If a transaction T_A issues the message *mid-rollback*, all the transactions that receive the message *mid-rollback* from T_A must also be rolled back.
- ? Once T_A issues the message *mid-rollback*, any transaction T_B that has updated the value of some object updated by T_A , must also be rolled back. This is called a cascade-rollback.

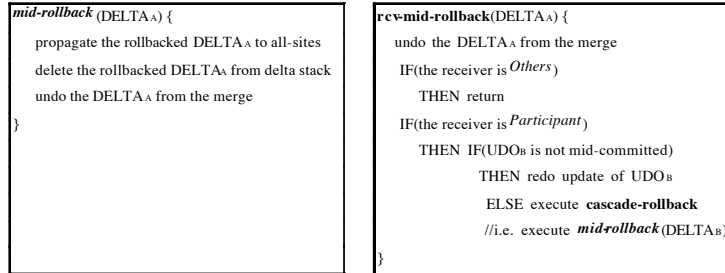


Fig. 15. The algorithm of *mid-rollback*

The algorithm of the *mid-rollback* operation is shown in Fig. 15. The *mid-rollback* operation propagates the DELTA to all-sites. Then, the DELTA is removed from the delta stack, and then the merge is cancelled. Any site that receives the message *mid-rollback* executes the *rcv-mid-rollback* operation. First, the *rcv-mid-rollback* rolls the propagated DELTA back. Then, the *rcv-mid-rollback* checks whether it is a *Participant* or one of the *Others* in the *mid-commit* of UDO_A. If the site is a *Participant*, undoing current updates or cascading the rollback may be required according to the *mid-commit* of UDO_B.

5.3 An Example of Update Propagation Scenarios

In this section, we will illustrate an example where *SR-based locking* is required. Fig. 16 shows an example where two transaction, T_A and T_B, update two spatial objects, a polygon and a line, at the same time. We assume that the 'property' layer and 'road' layer are updated by T_A and T_B at sites A and B, respectively, and the update of the 'property 1' is followed by the update of the 'road 1'.

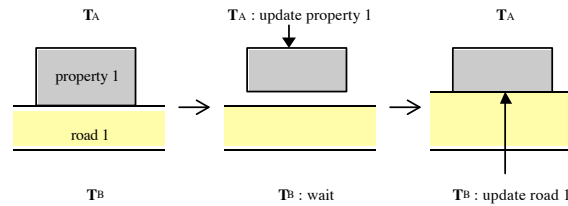


Fig. 16. Concurrent updates of two spatial objects, a polygon and a line

An example of update propagation scenarios is shown in Fig. 17. We divide the update propagation protocol into three phases for simplifying the scenarios: locking phase, mid-commit phase, and lock releasing phase. Each phase of a transaction T_B is as follows:

Phase 1 : locking phase (denoted as \langle in Fig. 17)

1. Set *region lock*, RGL_B , which contains a spatial object 'road 1', and notifies it to T_A . The return message of T_A informs that two *region locks* RGL_A and RGL_B are non-disjoint (reply-region-NDJ).
2. Set *SRBW lock* on 'road 1', and notifies it to T_A . The return message of T_A signals that T_A have the same lock on 'road 1' or on the *distributed-SR dependent* object. T_B is blocked until the lock is released.

Phase 2 : mid-commit phase (denoted as \langle in Fig. 17)

1. Propagate the $DELTA_B$ of 'road 1' to the *Participant*, T_A , and start the *SR-based 2PC*. The *Participant* T_A votes 'accept' because the $DELTA_B$ includes no errors (phase 1 of *SR-based 2PC*).
2. After the *Participant* (T_A) replies 'accept', the *Coordinator* sends a signal send-global-commit (phase 2 of *SR-based 2PC*).
3. After receiving the send-global-commit, T_A merges the $DELTA_B$ into its own data set.

Phase 3 : lock-releasing phase (denoted as \rangle in Fig. 17)

1. Propagate the message release-SRBW-lock on 'road 1' to T_A , after completing the update of 'road 1'.
2. Propagate the message *release-region-lock* of T_B to T_A , when there are no more objects to be updated in RGL_B .

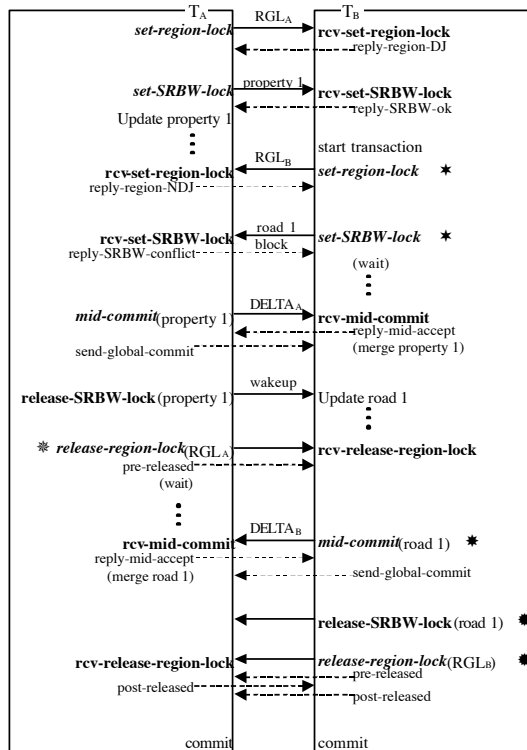


Fig. 17. SR-based 2PC for Participants voting accept (**)

6. Implementation

For building the replicated spatial database, we have decided to use an object-oriented spatial database system, called 'Gothic 3.0' [14], being operated on two workstations (HP C200 and DEC 500/400). The Gothic system still provides no concurrency control scheme for updating spatial objects. The system sets a WRITE lock on the entire data set before starting any update transactions. We assume that two sites can store and update replicated spatial objects.

We have developed the Replication Manager (R-manager in Fig. 18) on top of the Gothic. The Replication Manager is composed of 5 modules. User Interface, Lock Manager, Protocol Processor, Message Processor, and Catalog Manager.

(** The implementation issues are how to realize *region locking*, *SR-bound write locking*, and *SR-based 2PC*. The detail implementation techniques and processing mechanisms are as follows:

- ? *region locking* and *SR-bound write locking*: The User Interface module allows users to define the region to set READ locks to a set of display objects, and passes it to the Protocol Processor module. The lock compatibility, for example, whether *SR-bound write lock* could be allowed or not, is checked by the Lock Manager module. The Lock Manager module represents and manages locking information.
- ? *SR-based 2PC*: The user's decision in the *SR-based 2PC* is caught by the User Interface module, and then passed to the Protocol Processor module. The *Coordinator* in the Protocol Processor module propagates DELTA of the updated object to remote sites and waits for the *Participants*' votes. The Message Processor module at a remote site receives the DELTA and passes it to the *Participant* in the Protocol Processor module. The *Participant* gets a user's decision from the User Interface module, and returns it to the *Coordinator*. After the *Coordinator* gathers all the *Participants*' responses, it notifies the global decision to remote sites. The *Coordinator* and *Participants* perform the process of delta-merge on the local data set, when the global decision is 'accept'. (**)

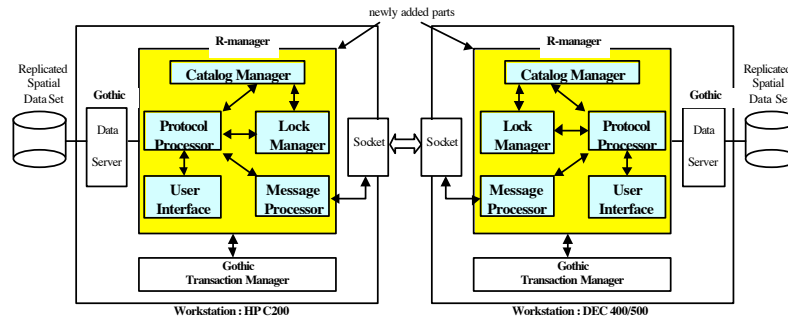


Fig. 18. Architecture for Replicated Spatial Databases

The main implementation problem was how to forbid users' attempts to update objects on which the other site already holds *SR-bound write locks*, without affecting the lock manager of Gothic. We resolved this problem by using event-processing function of Gothic. A 'reflex' of Gothic is a method to be invoked when an object is created, updated, or destroyed. Before updating an object, the 'reflex' method is invoked. Thus, our Lock Manager can catch the control and decide whether the update could be allowed or not.

7. Conclusions

In this paper, our goal was the development of new techniques for guaranteeing the concurrency and replication control in replicated spatial databases. To achieve these goals, we proposed new lock modes and an extended update propagation protocol. Because *region locking* does not block the entire data set, it can maximize the concurrency of long transactions. *SR-bound write locking* ensures the correct update

of spatial objects. We discovered that the *distributed spatial relationships* between spatial objects are a new dependency factor in the environment of concurrently updating replicated spatial data. *SR-bound write locking* could control the consistency of replicated spatial data having the *distributed spatial relationships*.

The contributions of this paper are as follows: First, we discovered that replicated spatial objects have *distributed spatial relationships*, and the objects are dependent on each other when they are updated at the same time, although they are not the same objects. Second, we proposed new distributed spatial relationship-based locking (*region locking* and *SR-bound write locking*), which support concurrency and replication consistency of long transactions. Third, we developed the extended update propagation protocol for supporting cooperative work and replication control of spatial data. The cooperative work is achieved by *SR-based 2PC protocol*.

Our implementation results showed that high concurrency could be achieved with little overhead. The update conflict caused by *distributed spatial relationships*, could be solved using *SR-bound write locking* and *SR-based 2PC protocol*.

References

1. Am-suk Oh, Jin-oh Choi, Bong-hee Hong: An Incremental Update Propagation Scheme for a Cooperative Transaction Model. Int. Workshop on DEXA (1996) 353-362
2. P. Chundi, D.J. Rosenkrantz, S.S. Ravi: Deferred Updates and Data Placement in Distributed Databases. Proc. Int. Conf. on Data Engineering (1996) 469-476
3. K. Stathatos, S. Kelly, N. Roussopoulos, I.S. Baras: Consistency and Performance of Concurrent Interactive Database Applications. Proc. Int. Conf. on Data Engineering (1996) 602-608
4. A. Kemper, G. Moerkotte: Object-Oriented Database Management : Applications in Engineering and Computer Science. Prentice Hall Press (1994)
5. G. Coulouris, J. Dollimore, T. Kindberg: Distributed Systems : Concepts and Design, 2ED. Addison-Wesley Publishing (1994)
6. M.J. Egenhofer: Reasoning about binary topological relations. 2th Int. Symposium, SSD '91 (1991)
7. H.F. Korth, G.D. Speegle: Long-Duration Transaction in Software. Proc. Int. Conf. on Data Engineering (1990) 568-574
8. J.Gray, P.Helland, D.Shasha: The Dangers of Replication and a Solution. Proc. of the 1996 ACM SIGMOD (1996) 173-182
9. E.Pitoura: A Replication Schema to Support Weak Connectivity in Mobile Information Systems. Int. Workshop on DEXA (1996) 708-717
10. M.H. Nodine, S.B. Zdonic: Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications. Proc. Int. Conf. on VLDB (1990) 83-94
11. P.A. Bernstein, N. Goodman: An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. ACM Tran. Database Systems, vol.9, no.4 (1984) 596-615
12. D. Agrawal, A.E. Abbadi: The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data. Proc. Int. Conf. on VLDB (1990) 243-254
13. H. berenson, P.Nernstein, J.Gray, J.Melton, E.O'Neil, P.O'Neil: A Critique of ANSI SQL Isolation Levels. Proc. of the 1995 ACM SIGMOD (1995) 1-10
14. The Gothic Object Server Module Reference Manual. Laser Scan Ltd. (1998)