Concurrency control in distributed geographical database systems

Gjermund Hanssen

Department of Mapping Sciences, Agricultural University of Norway,
PO Box 5034, N-1432 Ås, Norway**
gjermund.hanssen@itek.norut.no

Abstract. This paper discusses concurrency control in distributed geographical database systems. A geographic data server providing transactional services must be designed to handle both long- and short-duration transactions. The main challenges are identified as: (1) preserving the isolation property when concurrent transactions perform read and write operations on pages; (2) ensuring that search and other operations on the spatial access structure appear isolated; and (3) whatever method that is chosen they must provide feasible solutions with respect to performance. Existing approaches are discussed and presented.

1 Introduction

With distributed geographical databases comes the problem of concurrency control, i.e., ensuring that database operations from different users do not interfere with each other.

Queries issued from the GIS application domain often access a large portion of the database, perform lengthy operations, pauses for input from users or a combination of above; transactions may last for hours, days, or even months. Such properties lead to special requirement specifications that a transactional data server has to meet. Existing approaches adressing it, and future research need is identified and presented in section 3.

The background section is meant to clarify spatial concepts and common terms used within concurrency control.

2 Background

2.1 Queries

According to Tomlin, queries to geographical databases can be viewed as dealing with layers and zones in a map [27]. Each layer is partitioned into zones: a set of locations with a common attribute.

** This work is supported by GIN (a strategic institute programme at NORUT Information Technology AS) sponsored by The Research Council of Norway, Science and Technology (Project 127586/420).

For a given map, we can have a land-use layer and a pollution layer. The land-use layer is divided into land-use zones, e.g., wet-land, river, desert, city, park and agricultural zones. The pollution layer contains zones with different degrees of pollution.

The attributes in the two layers reflect properties of the map. For example, the land-use layer can have: soil type, land-usage, and a spatial attribute of some geometric data type—representing the shape or boundary of each land-use zone.

Queries can be classified in terms of this hierarchy:

Local queries. Involve locations that are coincident on various layers, e.g., what combination of features are found at location x?

Focal queries. Deal with neighborhoods of locations on the same layer, e.g., find all wheat growing regions within 10 kilometers of the boundaries of rice-growing regions.

Zonal queries. Involve groups of locations that have the same attribute value within itself, e.g., where does wheat grow?

From an algebraic point of view the above queries are denoted as *Map Algebra*[1] [27] (and later referred several places in literature, e.g., [1, 10, 24]).

An alternative view is to consider each layer as a set of *geographical objects*, each having a *descriptive* and a *spatial component*. According to Shekar et al., interesting operations on spatial objects can be classified according to [24]:

- *Set-Oriented*: Equals, is a member of, is empty, is a subset of, is disjoint from, intersection, union, difference, cardinality (no computational geometry).
- *Relationship between spatial objects (topological)*: Boundary, interior, closure, meets, overlaps, is inside, covers, connected, components, extremes, is within (operations that may return new geometry).
- *Metric*: Distance, bearing/angle, length, area and perimeter.
- *Direction*: East, north, left, above and between.
- *Network*: Successors, ancestors, connected and shortest-path.

A majority of the queries mentioned above involve the use of *spatial join*, i.e., combining two sets on the basis of a spatial predicate. Example: Given two collections (sets) R and S of spatial objects and a spatial predicate θ, find all pairs of objects $(o, \acute{o}) \in R$ x S where $\theta(o.G, \acute{o}.G)$ evaluates to true:

$$R \bowtie_\theta S = \{(o, \acute{o}) \mid o \in R \wedge \acute{o} \in S \wedge \theta(o.G, \acute{o}.G)\}.$$

As for the spatial predicate θ, it's a wide variety of possibilities, including:

$intersects()$
$contains()$

[1] Map Algebra enjoy much popularity in GIS use and GIS education. The basic Map Algebra concepts are easy to understand and provide a fairly powerful framework for GIS analysis.

$is_enclosed_by()$
$distance()\Theta q$, with $\Theta \in \{=, \leq, <, \geq, >\}$ and $q \in E$ (Euclidean space)
$northwest()$
$adjacent()$

2.2 Transactions

There is no notion of *consistent execution* or *reliable computing*[2] associated with a query—with transactions, this is what it's all about.

Transaction is a typical abstraction concept, it frees application developers from the burden of dealing with reliability and consistency, it can simply be left to a system providing that service. Transactions form the "interface contract" between an application program and a transactional data server:

1. The application program specifies, during its execution, the boundaries of a transaction, by issuing Begin transaction and Commit transaction calls.
2. The server automatically considers all request that it receives from the application program within this scope as it belongs to the same transaction; and, most important, guarantees certain properties for this set of requests and their effects on the underlying data.

The properties a server guarantees for a transaction include:

Atomicity. Ensures that either all operations of a transaction completes successfully or all of its effects are absent.
Consistency. Ensures that a transaction maps the database from one consistent state to another consistent one, i.e., database consistency.
Isolation. Ensures that no transactions ever view any partial effects of other transactions, even though they execute concurrently, i.e., transaction consistency.
Durability. Ensures that changes (committed transactions) to a database are persistent even when the system crashes.

"ACID," for short.

2.3 Concurrency control

Concurrency control is the problem of coordinating concurrent transactions in a transactional server. The naive approach is to have the transactions to execute serial. Unfortunately, that is not feasible with respect to performance. The main problem is to allow as much parallelism as possible, while preserving transactional consistency (maintaining the I property in ACID). Thus, it's a problem a

[2] Reliability refers to both the *resiliency* of a system to various types of failures and its capability to recover from them. A resilient system is tolerant of system failures and can continue to provide services even when failures occur.

transactional server has to cope with in order to fulfill his part of the "interface contract" between him and the application.

Serializability is the theoretical base associated with concurrency control. It's the "law" telling right from wrong—the *correctness criterion*. The idea is pretty simple: concurrently executed transactions are correct if they are equivalent to serially executed transactions; more precisely, a concurrent execution of transactions is X-serializable if it is X-equivalent to some serial execution of the same transactions.

2.4 Concurrency control methods

The goal concurrency control algorithms must accomplish, is to produce serializable schedules for multiple transactions. Testing for it, is infeasible (although algorithms for doing so exist), because it is "expensive" in computational terms (e.g.,testing for view equivalence is NP-complete[3] [22]). Instead, protocols are the prefered way of handling this.

Concurrency control methods can be classified according to:

1. Pessimistic (both locking[4] and non-locking techniques).
2. Optimistic (non-locking).

Pessimistic algorithms synchronize concurrent execution of transactions early in their execution life cycle;

$$validation(V) \rightarrow read(R) \rightarrow computation(C) \rightarrow write(W);$$

whereas, optimistic algorithms delay the synchronization of transactions *until* their termination:

$$read(R) \rightarrow computation(C) \rightarrow validation(V) \rightarrow write(W).$$

Locking. "Lock everything you access and hold all locks until commit." The informal *everything you access* can be refined to *every data item a transaction wants to perform read or write operations on*[5]. Most locking algorithms follow 2PL[6] (two-phase locking) or variants thereof. 2PL consists of a growing and a shrinking phase. In the growing phase locks are aquired, whereas in the shrinking

[3] Meaning that finding an efficient polynomial time algorithm for this problem is highly unlikely.

[4] The synchronization of transactions is achieved by employing physical or logical locks on some portion, unit or granule of the database. The size of these portions is usually called locking granularity, and usually; the smaller granule-size, the higher concurrency among transactions.

[5] The simple page model is motivated by the observation that all higher-level operations on data are eventually mapped into read and write operations on pages. This represent a strong form of abstraction, but it has prooved it's suitability in the sense that a comprehensive theory of concurrency control can be built on it.

[6] 2PL is the one used most in commercial database systems.

phase locks are released. Two types of locks—lock modes—are associated with each lock unit: read lock (shared lock), and write lock (exclusive lock). A transaction T_i that wants to read a data item contained in lock unit x, obtains a read lock on x; same procedure for write operations, but now with write locks. Two lock modes are compatible if two transactions, which access the same data item, can obtain locks on the data item at the same time. Read locks are compatible, whereas read-write and write-write locks are not.

Non-locking. A first approach to get rid of locks is to use timestamps. Timestamp ordering protocols select, a priori, a serialization order and execute transactions accordingly. To establish this ordering, the transaction manager assigns each transactions T_i a unique timestamp, $ts(T_i)$, at its initiation [29, 8]. The timestamp of a transaction is inherited by every operation of that transaction. The idea is, in case of conflict between operations, to give operations with lowest timestamp ordering priority.

2.5 R-Tree

An R-Tree (figure 1) is structured with index records in its leaf nodes containing pointers to data objects. Nodes correspond to disk pages if the index is disk resident. The design allows spatial search to visit only a small number of nodes. The index is completely dynamic; inserts and deletes can be intermixed with searches, and no periodic reorganization is required.

A dataset in a spatial relational database consists of a collection of tuples, representing spatial objects; and each tuple has a unique identifier, used to retrieve it. Leaf nodes contain index record entries: <I,tuple identifier>, where *tuple identifier* refers to a tuple in the database, and I is an n-dimensional rectangle—the smallest possible bounding box of the spatial object indexed. Non-leaf nodes contain entries: <I,child pointer>, where *child pointer* is the address of a lower level node in the R-Tree; and, I is the smallest possible rectangle that cover all rectangles in the lower nodes' entries.

Let M be the maximum number of entries that will fit in one node; let $m \leq \frac{M}{2}$ be a parameter specifying the minimum number of entries in a node; then, Guttman's R-Tree satisfies [9]:

- Every leaf node contains between m and M index records unless it is the root.
- For each index record <I,tuple identifier> in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data object represented by the indicated tuple.
- Every non-leaf node has between m and M children unless it is the root.
- For each entry <I,child pointer> in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node.
- The root node has at least two children unless it is a leaf.
- All leaves appear on the same level.

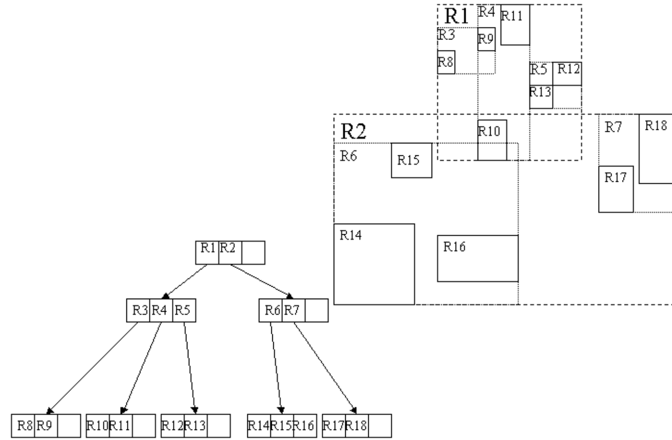

Fig. 1. An R-Tree along with the spatial distribution of the data items

The height of an R-Tree containing N index records is at most $\lceil \log_m N \rceil - 1$.

A closer look at figure 1 illustrates the containment and overlapping relationships that can exist between its rectangles.

Variants of the original R-Tree: the R+-Tree [26], and the R*-Tree [3] minimize the amount of overlapping rectangles in order to reduce I/O complexity that the R-Tree introduces. The R*-Tree uses *forced reinsert*, i.e., if a node overflows, it is not split right away, but entries are removed from the node and reinserted. The R+-Tree avoids overlapping rectangles in intermediate nodes of the tree. Thus, the I/O complexity is bounded by the depth of the tree. Analytical results indicates that R+-Trees can achieve up to 50% savings in disk accesses compared to an R-Tree when searching files of thousands of rectangles [26]. However, the downside is: (1) More expensive insert operation; and (2) more waste of storage space.

3 Concurrency control in distributed geographical databases

Concurrency control in distributed geographical databases is: (1) Preserving the isolation property when concurrent transactions perform read/write operations on pages; (2) ensuring that search and other operations on the spatial access structure appear isolated; and (3) whatever method that is chosen—it must perform well.

(1) Concurrently executed transactions need synchronization because they may interfere in: Read/write, write/read and write/write ways. This can cause undesirable phenomena:

Dirty reads. A transaction reads data written by concurrent uncommitted transactions. This may be a value that never really existed, for example, when another transaction had to abort.

Lost update. If a second transaction read a item for update after the first transaction has read it, but before the first transaction has committed. Whichever of the transaction commit first, *that* update will be lost.

Non-repeatable reads. A transaction re-reads data it has previously read and finds it modified.

Phantom read. A transaction re-executes a query, finding a set of data not equal to a previous one—although the search condition is unchanged.

(2) Maintain the isolation property when inserts and deletes are intermixed with searches.

(3) Whatever method that is chosen: pessimistic, optimistic, locking, non-locking, it must perform well. This follows from Gray and Reuters' second law of concurrency: concurrent execution should not have lower throughput or much higher response times than serial execution [12].

3.1 Properties

Transactions can be characterized according to their duration, structure and frequency of read and write operations:

Short-duration transactions. Typical when few data items with a simple structure are involved, e.g., point queries.

Long-duration transactions. A spatial object, e.g., a polygon can consist of thousands of edges; retrieval opertions (I/O) and processing of spatial joins is expensive [11]; together with human interaction and intellectual decision making on intermediate results—this implicates the presence of long-duration transactions.

Low degree of conflicts. Certain types of geographic data are infrequently updated, observe: (1) they seldom change, e.g., contour lines, boulders, marshes, water and administrative borders; and (2) databases are often updated in so-called production databases and at some point made available to *the rest of the world*.

3.2 Requirement specification

A geographic data server providing transactional services must fulfill three points:

1. The isolation property must be guaranteed in the presence of long- and short-duration transactions; while doing so, concurrent execution should not have lower throughput or much higher response times than serial execution.
2. Flexibility with respect to isolation level. e.g., "map browsers" are more eager to see an inconsistent map, than no map at all.
3. Concurrent operations on the spatial index structure must appear isolated.

3.3 Existing approaches

Fortunately, the *the long-duration problem* is not a problem related *only* to the GIS-application domain. Thus, it can adopt general solutions to the problem. Unfortunately, less is accomplished when it comes to improve concurrency on spatial index structures.

With this on mind, existing approaches can be characterized according to:

1. Extend traditional locking techniques.
2. Versioning and checkout.
3. Hybrid.
4. Relaxed isolation.
5. Dynamic granular locking in the R-Tree.

Extend traditional locking techniques. Extracting semantic information about operations in a transaction, and use it to increase concurrency when resources are no longer needed; other transactions can benefit those resources, if they satisfy certain requirements.

A formal mechanism that follows this approach is altruistic[7] locking [25]—an extension to 2PL. It uses information about access patterns of a transaction to decide what resources it can release. In particular, the technique uses two types of information: (1) negative access pattern information, which describes objects that will not be accessed by the transactions; and (2) positive access pattern information, describing what objects and in what order they will be accessed. Taken together, this information allows long-duration transactions to release theirs locks after they are done with them.

The set of all data items that have been locked and then released by a long-duration transaction is called the *wake* of the transaction. Releasing a resource is a conditional unlock operation because it allows other transactions to access the released resource as long as they follow two restrictions: (1) no two transactions can hold locks on the same data item simultaneously, unless one of them has locked and released the object before the other locks it; and (2), if a transaction is in the wake of another transaction, it must be completely in the wake of that transaction.

Versioning and checkout. The main idea behind versioning is to allow more than one copy of each data item. This scheme makes it possible to keep the "old" version of a data item that is subject to overwriting, at least until the transaction that writes a "new" version commits. More generally, distinct transactions could be given distinct versions of the same data item to read, or to overwrite.

A common approach along this direction is checkout [7, 13, 14, 16, 19]. Some authors call this non-transparent versioning (e.g., Weikum and Vossen [29]), in the sense that users or applications are aware of the fact that data items can

[7] The unselfish transaction informs other that resources are free to use, without benefiting from itself.

appear in different versions. The idea of checkout comes from CAD (computer-aided design) and CASE (computer-aided software engineering) where database users need to play around with versions of data in order to develop, change, alternate or experiment with their design.

In this approach, the user specifies an area in which he wishes to work, and the data selected is copied to a separate working area which is used by that user only. The working area may or may not use the same DBMS and data structures as the master database. Updates are made to this working data set, and when the work has been completed the changes are applied to the master database. The concurrency control mechanism used may either be: (1) pessimistic, in which case all data retrieved by a user is locked and can only be viewed (read) by other users; or (2) it can be optimistic, in which case multiple users can retrieve overlapping areas and conflicts are identified when changes are passed back to the master database.

Advantages of checkout may be:

- Total control of your data.
- The data are near you, and they are not lost in case of conflicts (not rolled back).
- In the optimistic approach other transactions are not blocked.
- Provides a viable solution to the long-duration problem when *the policy* is that only one user is allowed to update data.

However, checkout has its disadvantages: First, the initial retrieval of data can take a long time—checkout times tend to be measured in minutes rather than seconds; second, the user has a restricted subset of the database to work in; if he discovers that he needs to work outside the area that he originally requested, then, it is necessary to do another retrieval, which may again take a significant time. If relationships (including topological relationships) are allowed between objects in the database, this introduces further complications in deciding exactly how much data to check out—some mechanism is required for controlling relationship among objects, which have and have *not* been extracted.

Transparent versioning. This approach allows us to create different versions of the database—alternatives. Only one user can update an alternative at one time, but any number of users can read an alternative. Changes made by a user within an alternative are only seen by himself. The whole database can be seen within an alternative, but data is not replicated; because, only changes relative to the parent version are stored. When changes to an alternative have been completed, it can be posted to the parent alternative.

Version management is implemented in the Smallworld Version Managed Data Store (VMDS) [21]. This implementation is based on an optimistic approach: conflicts are detected and corrected *until* commit. Newell and Batty argues that version management overcomes all the problems with checkout mentioned above: No initial retrieval time, no copying of data is required, and the user has access to the whole database at all times [21].

However, the downside with versioning is: first, complicated garbage collection—must limit the number of versions due to system efficiency and memory space; second, a transaction need to know which version of data to read—complicated, although a version function is proposed to solve *that* by Papadimitriou [22]. This implicates that much overhead is associated with version management.

Hybrid. Bayer et al., 2V2PL (two-version-two-phase locking) merge conventional concurrency control with versioning [2]. Within this scheme at most two versions of any data item is kept at each point in time. Read operations benefit from this, because they can read any version of the item. If the most current version is locked the reader can access the *stable* version.

The ROMV protocol (a multi-version protocol for read-only transactions) proposed by DuBourdieu [6] and Chan et al. [5] with implementation issues in Mohan, Pirahesh and Lorie [20]; provides some interesting concepts along this direction. Within ROMV, each read-only transaction is assigned a timestamp corresponding to the start of the transaction, rather than to its commit; thereby, each read operation is assigned to the most recent version that has been committed at the time of the readers begin; so, read-only transactions always access the version with the largest timestamp, which is smaller than the transactions timestamp (assuming that timestamps of different transactions are never equal).

Update transactions, on the other hand, is subject to the conventional S2PL (strict-two-phase locking) protocol[8]. They aquire conventional locks for both read and write operations, which are released according to the two-phase rule—with read and write locks held *until* commit. In contrast to the conventional monoversion setting; each write operations creates a new version rather than overwriting the data item; each version is timestamped with the timestamp of its transaction that corresponds to the commit time of the transaction. So, update transactions do not benefit from versioning at all, but benefit the read-only transactions while keeping the overhead of the protocol as low as possible.

Relaxed isolation. Observing that a consistent view of data is not always considered to be critical: (1) GIS-applications "browsing maps"; or (2) performing statistical analysis, e.g., computing the average size of real estate land parcels within a region—allows us to *relax* the isolation property.

The ANSI/ISO SQL standard defines four levels of transaction isolation in terms of three phenomena that must be prevented among concurrent transactions. The phenomena are:

- Dirty reads.
- Non-repeatable reads.
- Phantom read.

The four isolation levels and the corresponding behaviors are described in table 1.

[8] All write locks are released after the transaction commits.

Table 1. SQL Transaction Isolation Levels

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Remark 1. The isolation levels supported by the SQL standard are defined in terms of S2PL. However, they should be enforced by whatever appropriate concurrency control algorithm a server chooses to employ.

A specific isolation level is chosen by issuing a corresponding SQL command "set isolation level. . . ." Read uncommitted does not require any read locks, whereas write locks are required and released according to S2PL. The Read committed level (cursor stability level) prevent applications from reading *dirty data*. It is characterized by short read locks, and long write locks; thus, it reduces the possibly data contention among long readers and short update transactions. The Repeatable read option allows phantom anomalies, whereas, Serializable prevents against all undesirable phenomena.

PostgreSQL[9] offers the read committed (snapshot[10] isolation level) and serializable isolation levels in its implementation. Read committed is the default level. These isolation levels are defined in terms of a multiversion concurrency control protocol (not according to the SQL standard), because PostgreSQL has adopted a multiversion model for concurrency control. This means that while querying a database each transaction sees a snapshot of data (a version, an alternative) as it was some time ago, regardless of the current state of the underlying data.

The approach is attractive. However, it's not without problems: manually controlling the isolation level on a per-application or even on a per-transaction level represents in some sense a step backward; to the time before the transaction concept was adopted in distributed computer engineering. Because: (1) The basic idea behind transactions is that they provide services to the application side, and guarantees certain properties (ACID)—it makes life much easier for the application programmer, he is not exposed to obscure fault scenarios and may concentrate on getting the job done; and (2) according to Murphy's law, everything that can go wrong, will do so—especially in computing—thus, it's likely that experimenting with reduced isolation will lead to unacceptable behaviour.

Dynamic granular locking in the R-Tree. Recall from the background section that an R-Tree may be highly dynamic: insertions may cause propagation effects and reorganization of the structure, thus, concurrent operations on the

[9] `www.postgresql.org/docs/view.php?version=7.3idoc=1file=mvcc.html`

[10] This term is used when the transactional server employ a multiversion concurrency control protocol. The snapshot is then the version.

index structure may cause inconsistency on the structure itself. An easy approach would be to lock the whole tree. Unfortunately, this will not provide high concurrency—but, possibly low performance.

Defining lockable granules at a finer level than the whole tree; let them grow and shrink according to the dynamic behaviour of the tree—that's the essence within Dynamic granular locking.

Deployed to an R-Tree, the approach must pay attention to: spatial overlap that can exist among its nodes—due to the lack of linear ordering in multidimensional space—so, overlap can also exist between granules; a lock on a granule may not provide exclusive coverage on the space it covers—the granules are not *mutually disjoint*.

Chakrabarti propose a solution addressing the *phantom* problem based on the above idea [4]. Lockable granules are defined as tree granules, according to figure 1 they correspond to: R3, R4, R5, R6 and R7. Non-covered space is partitioned into external granules, associated with each non-leaf of the R-Tree, whose shape and size is determined by the R-Tree partitionings. Inserts, which may occur in the overlapping regions when they should *not*, are handled in two ways: First, let O be the object inserted and g the target granule, then, inserts that cause granules to grow aquires locks on the minimal set of granules covering O and *all* granules into which g grows; second, inserts that *not* cause granules to grow aquires locks on *only* the minimal set of granules covering O.

The consistency problem (not covered by Chakrabarti) is addressed in [15, 18, 23]. The approach taken by Kornacker, Banks and Stonebraker is to add sibling pointers to nodes, a technique first deployed to B-link trees; compensating for concurrent structure modifications. The protocol operates on a variant of the R-Tree—the R-link tree.

4 Summary

Preserving the isolation property when concurrent, possibly long-duration transactions perform read and write operations on pages; search intermixes with updates on a spatial index structure; acceptable behaviour with respect to performance; taken together, this forms the main goal concurrency control in distributed geographical database systems must accomplish.

Existing approaches focuse: (1) extending traditional locking techniques, (2) versioning, and (3) a hybrid variant. Among these, keeping multiple versions of data items is an attractive idea. A transaction is not locked out by another, thus, more concurrency is allowed.

A more pragmatic approach is to give the user some influence on what isolation levels the system should support—the relaxed isolation approach—it is attractive when a consistent view of the data is not considered to be critical; performance is more important than consistency, e.g., GIS-applications whose purpose is to "browse maps" or perform statistical analysis.

A brief state of the art for concurrency control and spatial index structures reveals that little is done in the research community on these topics. For the R-Tree, Dynamic granular locking and the R-link Tree represents initial approaches.

5 Concluding remarks

Concurrency control literature is vast. This is reasonable, because concurrency control is an essential issue whenever it comes to many users and shared resources. Large, distributed, heterogenous computer systems cannot be made reliable without an understanding of the transaction concept. This will also be the case for distributed geographical database systems.

Surprisingly little work is done in the area of concurrency control and spatial index structures. In the future, I'm planning a trip into the forest of R-Trees and concurrent operations on them.

References

1. Bruns, T. H. and Egenhofer, M. J. *User Interfaces for Map Algebra*, Journal of the Urban and Regional Information Systems Association, vol.9, no.1, pages 44-54, 1997.
2. Bayer, R. Heller, H and Reiser, A. *Parallelism and Recovery in Database Systems*, ACM Transactions on Database Systems, vol.5, pages 139-156, 1980.
3. Beckmann, N. Kriegel, H. P.Schneider, R.Seeger, B. *The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles* In Proceedings of ACM SIGMOD Conference, pages 322-331, 1990.
4. Chakrabarti, K. *Supporting Spatial Index Structures as Access Methods in a Database System* Thesis, B.Tech, Indian Institute of Technology, Kharagpur, 1999.
5. Chan, A. and Gray, R. *Implementing Distributed Read-Only Transactions*, IEEE Transactions on Software Engineering 11, pages 205-212, 1985.
6. DuBordieu, D. *Implementation of Distributed Transactions*, In Proceedings of 6th Berkely Workshop on Distributed Data Management and Computer Networks, pages 81-93, 1982.
7. Du, H. C. and Ghanta, S. *A framework for efficient IC/VLSI CAD databases*, In Proceedings of the 13th International Conference on Very Large Databases, VLDB Endowment, pages 619-625, 1987.
8. Elmasri, R. and Navathe, S. B. *Fundamentals of Database Systems*, Addison-Wesley, Third Edition, 2000.
9. Guttman, A. *R-Trees: A dynamic index structure for spatial searching*, In Proceedings of ACM SIGMOD Conf. pages 47-57, 1984.
10. Güting, R. H. *An Introduction to Spatial Database Systems*, VLDB Journal, vol.3, pages 357-399, 1994.
11. Gaede, V. and Günther, O. *Multidimensional Access Methods*, ACM Computing Surveys, vol.30, no.2, pages 170-231, 1998.
12. Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, CA, 1993.
13. Haskin, R. and Lorie, R. *On extending the functions of a relational database system*, In SIGMOD International Conference on Management of Data, ACM, New York, pages 207-212, 1982.

14. Katz, R. *A database approach for managing VLSI design data*, In 19th Design Automation Conference, IEEE, New York, pages 274-282, 1982.
15. Kornacker, M. and Banks, D. *High-concurrency locking in R-Trees*, In Proceedings of Very Large Databases (VLDB), pages 134-145, September 1995.
16. Kim, W. Lorie, R. McNabb, D. and Plouffe, W. *A transaction mechanism for engineering design databases*, In Proceedings of the 10th International conferance on Very Large Databases, VLDB Endowment, pages 355-362, 1984.
17. Kornacker, M. Mohan, C and Hellerstein, M. J. *Concurrency and Recovery in Generalized Search Trees*, In Proceedings of ACM SIGMOD Conf. May 1997.
18. Kanth, K.V. Serena, D. and Singh, A. K. *Improved Concurrency Control Techniques for Multi-dimensional index Structures*, Department of Computer Science, University of California, Santa Barbara, CA 93117.
19. Lorie, R. and Plouffe, W. *Relational databases for engineering data*, IBM, San Jose, Calif. 1983.
20. Mohan, C. Pirahesh, H. and Lorie, R. *Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions*, In Proceedings of ACM SIGMOD International Conference on Management of Data, pages 124-133, 1992.
21. Newell, R. C. and Batty, P. M. *GIS Database are Different*, Smallworld Systems Ltd, 1994.
22. Papadimitriou, C. H. *The Theory of Database Concurrency Control*, Rockville, MD, Computer Science Press, 1986.
23. Salzberg, B. *Grid file concurrency*, Informations and Systems, vol. 11 (3), pages 235-244, 1986.
24. Shekar, S. Chawla, S. Ravada, S. Fetterer, A. Liu, X. and Lu, C. *Spatial Databases Accomplishments and Research Needs*, IEEE, Transactions on Knowledge and Data Egineering, vol.11, no.1, January/February 1999.
25. Salem, K. Garcia-Molina, H. Shands, J. *Altruistic Locking*, ACM Transactions on Database Systems, vol.19, no.1, pages 117-165, March 1994.
26. Sellis, T. K. Roussopoulos, N and Faloutsos, C. *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*, In Proceedings of Very Large Databases (VLDB), pages 507-518, 1987.
27. Tomlin, C. D. *Geographic Information Systems and Cartographic Modeling*, Prentice Hall, Englewood Cliffs, N.J. 1990.
28. Worbys, M. F. *GIS: A Computing Perspective*, Taylor & Francis, 1995.
29. Weikum, G. and Vossen, G. *Transactional Information Systems*, Academic Press, 2002.