Versioning of Network Models in a Multiuser Environment

Petko Bakalov[1], Erik Hoel[1], Sudhakar Menon[1], and Vassilis J. Tsotras[2]

[1] Environmental Systems Research Institute, Redlands, CA 92373, USA
{pbakalov, ehoel, menon}@esri.com
[2] University of California, Riverside, CA 92507, USA
tsotras@cs.ucr.edu

Abstract. The standard database mechanisms for concurrency control, which include transactions and locking protocols, do not provide the support needed for updating complex geographic data in a multiuser environment. The preferred method to resolve conflicts in GIS systems is to encapsulate the modifications generated by the end users through the use of multiple versions. Multiuser (or versioned) geographic databases allow users to operate as though they have full access to the entire dataset. Instead of relying upon row locking, versioned databases allow multiple users to simultaneously edit the same row. They implement a model for conflict detection and resolution where the first to commit the change wins by default (though clients can manually intervene and select the latter change as the winner).

Network models are frequently used as a mechanism to describe the connectivity information between spatial features in many emerging GIS applications. Supporting networks within the context of a versioned database imposes additional requirements – the complex network model must retain integrity irrespective of the sequence of simultaneous edits by various clients. In this paper, we review our network model and discuss the enhancements necessary to maintaining topological network integrity in this complex environment. Our solution is based on the notion of dirty areas and dirty objects (i.e., regions or elements that contain edits that have not been reflected in the network connectivity index). The dirty areas and objects are identified and marked during editing of the network feature data. They are then subsequently cleaned as a byproduct of the incremental update of the connectivity network.

Keywords: Versioning, Network Models, Transportation Networks

1 Introduction

Network data models have a long history as an efficient way to describe the topological connectivity information among spatial features in geographic information systems [11], [14], [17], [18]. At an abstract level, the network model can be viewed as a graph whose elements explicitly represent the connectivity information about the features in the database. The presence of an edge in the graph depicts the information that the two features represented by the junctions are connected and vice versa. Different versions of the network model have been implemented in existing

operational systems such as ARC/INFO [20] and TransCAD [5]. Because of the large volume of data frequently found in these networks, the model is typically persisted inside a centralized database server. Using connectivity information, those systems can then be utilized to solve a wide range of problems, typical for the transportation or utility network domains (e.g., finding the shortest path between points of interest, finding optimal resource allocation, determining the maximal flow of a resource, and other graph theoretic operations).

A typical requirement for the network models (or to the GIS in general), is that they must provide support for many users simultaneously creating and updating large amounts of geographic information. In scenarios where those users are required to edit the same data at the same time, the system must provide an editing environment that supports multiuser concurrent modifications without creating multiple instances of the data. In contrast to traditional DBMSs, this editing environment must also support edit sessions that typically span a number of days or weeks, the facility to undo or redo changes made to the data, and the ability to develop models and alternative application designs without affecting the published database.

Concurrency control in traditional database management systems is addressed through the use of transactions and the two-phase locking protocol. This is efficient for short-lived edit operations that are typically completed in few seconds. It is not well suited however for the type of editing tasks required when updating geographic data. For a GIS multiuser environment, the row-locking mechanisms adopted by many DBMSs would be prohibitively restrictive for many common workflows.

To deal with long-lasting transactions, a solution based on the use of multiple versions has been proposed [16], [25]. A version can be logically viewed as an alternative, independent, persistent view of the database that does not involve creating a copy of the actual data. Since there is an independent view for every user, a versioned database can support multiple concurrent editors.

In addition, versioning is useful in many other GIS scenarios such as:

- **Modeling "what if" scenarios**. The versioning mechanism allows end users to exploit different alternatives (versions) during a design phase.
- **Workflow management**. Typically the design process goes through multiple steps organized in a workflow process where the output of one step is an input for another. The versioning scheme allows users to save intermediate results during the design process.
- **Historical queries**. The versioning scheme allows the preservation of different states of the data which later can be re-visited and re-examined if necessary.

Existing database versioning approaches cannot easily manage the specifics of the geographical data like topological network relations, the presence of connectivity among the stored elements, and traversability. Such information among spatial features is represented in a GIS by a network model.

Recently, we have proposed an incremental connectivity rebuilding algorithm for network models [1]. In this algorithm, the users are allowed to rebuild portions of the network model using the notions of dirty areas and dirty objects. Changes over portions of the network data are effectively captured and the incremental algorithm is

utilized to clean such dirty areas/objects and re-establish the associated portions of the network connectivity index. The connectivity rebuilding algorithm has been implemented in ArcGIS and provides an effective solution to maintain dynamic network models in an incremental manner.

In this paper, we propose a new versioning scheme for network models that utilizes the dirty areas/objects of the connectivity rebuild algorithm (a similar mechanism has also been applied to our topological data model [13]). Versioning of network models is different from version control over simple spatial data ("simple" meaning data that is geometrically unrelated to other data – i.e., no topological structuring). While the same basic principles are still in operation, resolving conflicts between features that are related to other features, as with network models, is different. This is because of the specific internal behavior of the network and the requirement that the connectivity information (or index) in the model should be kept consistent all the time.

The rest of the paper is organized as follows: Section 2 provides a brief description of the network model including logical structure and physical design and provides description of the algorithms used for connectivity establishment. Section 3 provides in depth discussion of versioning spatial databases. Section 4 addresses our proposed extensions of these techniques to the support of versioned network models. about the proposed versioning scheme we propose for network models. Section 5 discusses our implementation experiences, and Section 6 concludes the paper.

2 The Network Model

We now proceed with a brief description of the major aspects of the network model introduced in [14] and the algorithms for initial and incremental connectivity establishment presented in [1].

2.1 Basic Elements

A *network model* is described as a graph (named *connectivity graph*) that maintains the connectivity information about spatial features with line or point geometry. The basic elements of a network model are (*edges*, *junctions* and *turns*). Features with point geometry are represented with junction elements inside the graph, while lines are represented as one or more edge elements between pairs of junction elements. Figure 1 depicts the network model that is composed of spatial features and network elements. Similar designs have been used in many research or commercial implementations [7], [12], [15], [21]. In the network models we are considering, network elements are used only to describe the connectivity information for the spatial features they are representing; they do not carry any geometrical properties.

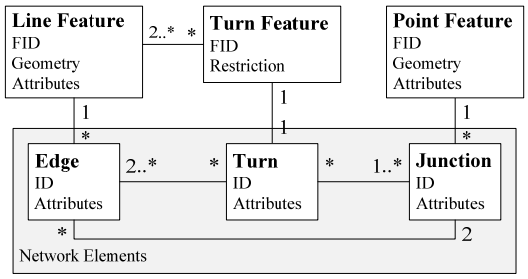

Fig. 1. Network model – features and elements

Most of the systems that utilize network models have client-server architectures. Because of their very large data size (e.g., many tens of millions of features for some nationwide or continent-wide transportation networks), the network models are usually located in a centralized server, persisted either in a RDBMS tables or in a file system. Typically the process of analysis is done within a GIS server (that acts as a client to the database) or within a thick client [4], [22], [26].

2.2 Traversability

While the connectivity elements (edges and junctions) allow the user to express connections, they are not sufficient for expressing specific restrictions from the real world (for example, no left turn, or, no u-turn allowed at an intersection) [3], [28]. Take for example a intersection formed by dual carriageway (a street with central divider) represented in the model with two line features traversable only in one direction and a two way street represented with a single line feature traversable both ways (see Figure 2). If we want to specify no u turn restriction in this intersection we have to restrict the traversability from edge e_1 to edge e_2 and then through edge e_3 in that sequence. This restriction cannot be expressed only in terms of junctions and edges. If in attempt to do so we disconnect e_1 and e_2 this will incorrectly imply no left turn specified by the sequence e_1, e_2 and e_4. To handle these scenarios, we introduce two new network elements, namely, turn (only between two edges) and maneuver (between two or more edges). These elements are used to implement restrictions and are universally enforced during analysis. As a result, the movement over a network is a subset of the network graph taking into account the movement restrictions.

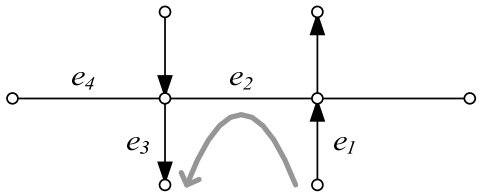

Fig. 2. Example of a three part maneuver e_1-e_2-e_3 at an intersection with a dual carriageway.

To introduce the turn restriction in addition to the edge and junction elements, a network model can also have a special network elements called *turns* (see Figure 1). Similar to the edges which are defined as a relation between junctions turns are defined as a relation between edges. A turn element is anchored to a specific junction (the junction where the turn starts) and controls the movement between sequence edges expressed as pairs (edge-In, edge-Out).

2.3 Physical Implementation

In our network implementation the connectivity information is maintained as a set of *adjacency pairs* of the form <*edgeId, junctionId*>, stored inside the "junction table" (see Figure 3). This approach is designed to answer the most common type of adjacency queries during the network analysis process. The junction table uses fixed-length records for direct access purposes; this implies a fixed number (four in our implementation) of adjacency pairs per record (see Figure 3). If the junction has more than four connected edges an overflow mechanism is applied.

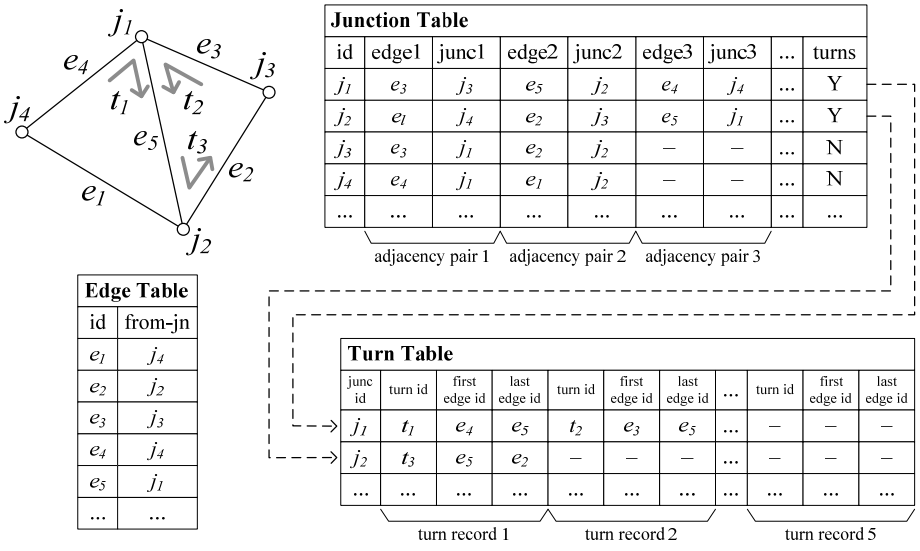

Fig. 3. Network tables example

In a similar way, in the traversal process, it is required that at each junction we know all the turns anchored at this junction. This has influenced the way we implement the turn storage scheme. Information about turns is stored in the "turn table", in the form of *turn triplets* <*turnId, edge-InId, edge-OutId*>. If there are any turns anchored at a junction j_i, the turn table will have a record with primary key j_i which also contains all the turns anchored on j_i. This storage scheme can be easily optimized for the most commonly used client access patterns [27].

2.4 Maintaining Network Connectivity

Maintaining network connectivity can be viewed as a two phase process [1]:

- Initial establishment of connectivity when the network model is first defined, with the connectivity index being derived from the features participating in the network.
- Incremental rebuilding the connectivity index on a periodic basis after edits occur on the spatial features in the network.

Having an incremental solution is of significant practical value - the amortized cost of maintaining an incrementally rebuildable network is far less than an ordinary network that must be periodically rebuilt in its entirety (e.g., editing a subdivision and only rebuilding that portion of the nationwide network versus rebuilding the whole nationwide network). In order to keep track of the modifications to the features that occur since the last full or partial rebuilding of the connectivity index, the network model employs the concept of *dirty areas*. Similarly, to track changes to elements without geometrical properties (e.g., turns), we use the concept of *dirty objects*.

Definition 1. *A **dirty area** corresponds to the regions within the feature space where features participating in the network have been modified (added, deleted, or updated) but whose connectivity has not been re-established.*

To simplify its computation and storage, a dirty area in our implementation is defined as a union of envelopes (e.g., bounding boxes) around the features that have been modified. It is possible however to use other shapes - the convex hull of the feature for example. In order to ensure that the network is correct, only the portion of the network encompassed in the dirty areas will need to be rebuilt.

Both the initial establishment of connectivity and the incremental rebuild algorithms follow the same four steps:

- **Geometrical extraction.** Extract the geometry information for all features in the area of interest (the whole area in the case of initial establishment or the dirty area in the case of subsequent rebuild) and analyze the vertices in those geometries. The extracted vertex coordinates and their corresponding feature identifiers are stored in a temporary table, called the "vertex table".
- **Connectivity analysis.** The content of the vertex table is sorted by coordinate values. As a result the coincident vertices from different features are grouped together. The algorithm scans the vertex table sequentially and picks groups of coincident vertices. Every single group is examined to determine if the vertices satisfy the connectivity model specified for the network.
- **Junction creation.** For each group which satisfies the connectivity model a new junction element is created in the network model. The junction id of this newly created junction element is added to all the vertices participating in this connectivity group.

- **Edge creation.** The content of the vertex table is then resorted using the feature identifier as the sorting key. As a result, the vertices for each line feature are again grouped together. The vertex table is scanned sequentially once more and for each pair of adjacent vertices which belong to the same line feature a new edge is created.

The difference between the incremental rebuild and the full (re)build algorithms, is that the incremental rebuild process adds to the vertex table those vertices that are outside of the rebuild region but belong to features which intersect the rebuild region. These vertices are saved and later reused as connection points through which the rebuild portion of the network is "stitched" together with the rest of the model.

Rebuilding turn features in the network requires additional processing. The complexity comes from the fact that the turn features are defined as a relation between two or more line features and typically do not have geometrical properties. As depicted in Figures 1 and 3, a record in the turn table consists of a turn identifier and a list of the line feature identifiers that participate in the turn. In order to cover network elements without geometrical properties, we extend our dirty area concept with the notion of *dirty objects*.

Definition 2. *A **dirty object** is an object without geometrical properties (like turn features) whose modifications have not yet resulted in the incremental rebuilding of the network connectivity index.*

During the rebuild process, we restore all dirty objects to their clean state. An object is kept as dirty until it is successfully cleaned. Turn features are marked as dirty objects when:

- The turn feature is directly modified (Insert, Update, Delete), or
- The associated line features are modified (Update, Delete), or
- The associated network turn element is deleted (this may happen during the rebuild process).

Using the dirty areas and dirty objects, we can capture the dynamic behavior of network maintenance. It is this dynamic behavior that complicates and thus requires extra attention during the versioning process.

3 Versioned Spatial Databases

Spatial databases have dramatically evolved in their capability to handle multiple simultaneous editors. Some solutions have required organizations to alter their workflow so as to ensure that no two editors are editing the same geographic region within the spatial dataset. Supporting such a constrained workflow can become problematic once the need for supporting long transactions (e.g., design alternatives) is considered. In order to address this problem where design alternatives on the same geographic area are necessary (as well as very long transactions spanning weeks or months are required), versioned geographic data management technologies were

developed [6], [8], [9], [19], [29]. Versioning does not prevent editing conflicts from occurring, rather, it provides an infrastructure for the detection and resolution of such conflicts.

Definition 3. *A **version** is a logical entity that represents a unique, seamless view of the database that is distinguished from other versions by the particular set of edits made to the version since it was created.*

Definition 4. *A **state** represents a discrete snapshot of the database whenever a change is made. Every edit operation creates a new database state.*

In versioned databases, there are two fundamental abstractions – *versions* and *states*. Versions are organized into a tree that is used to model the hierarchical relationships between versions (e.g., projects or design alternatives). A version is associated with a current state. A state is used to represent an instance of the database that is associated with a particular version. When a child state is created, it will initially have the same set of rows in each table as its parent state. However, as the state is edited, rows will either be added, deleted, or updated. Changes made in a child state are not visible in the parent state. Updated rows in the child will take precedence over the corresponding row in the parent when materializing the version associated with the child state.

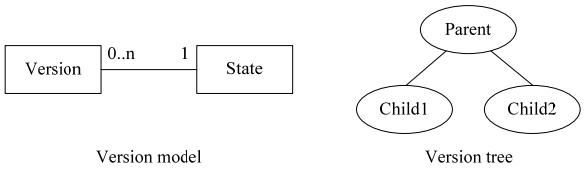

Fig. 4. Model depicting the relationship between versions and states is on the left, while a simple example version tree is shown on the right

Similar to versions, states are also organized into trees. A version will commonly be associated with numerous states over its lifetime; however, it will only be associated with a single state at any given moment in time. A given state may or may not be associated with one or more versions (as shown on the left side of Figure 4).

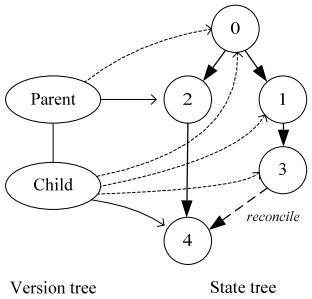

Fig. 5. Example version tree and state tree

In Figure 5, we highlight a simple example where there are two versions, labeled parent and child, and an associated state tree. In the example, the parent version initially is associated with state 0. When a child version is created (as a child of the parent), it will also point to state 0. Following an edit to the child version, the child will then point to state 1. Assuming that the next edit is to the parent version, the parent will then point to state 2. The child is then edited one more time (causing the child version to point to state 3) prior to reconciling (making the changes made in the parent visible to the child – see Section X.X for additional details) with the parent version. The reconcile will cause the changes that have been made in the parent version (i.e., the differences between states 0 and 2) to be visible in the new state that the child will point to following the reconcile (i.e., state 4). This sequence of edits and a reconcile leaves the parent version pointing to state 2, while the child version points to state 4.

Versioned databases are useful in supporting a number of database usage patterns and workflows [16]; this includes:

- Direct multiuser editing of the main database,
- Two-level project organizations – work-order processing systems,
- Multi-level project organizations – hierarchical design parts and alternatives,
- Cyclical workflows (multiple stages of approval), and
- Historical states (temporal snapshots).

Some organizations will require the versioned database to support several of these workflows simultaneously; for example, a utility company may organize itself into a two-level project organization for maintaining its 'as built' status, while additionally requiring the maintenance of historical states (temporal snapshots). The key point is that a versioned database must be able to support each of these usage patterns (oftentimes simultaneously).

3.1 Operations on Versioned Databases

There are two fundamental operations that can be performed on versioned databases that are required in order to support versioning. There two operations are termed *reconciling* and *posting* (note – in the following discussion, we will employ the general terms 'child' version and 'parent' version; child version will refer to a version of interest, while parent version will generically refer to any ancestor version of the child within the version tree). Reconciling is logically the process of taking a child version and merging all the changes that have been made in its parent version (effectively making changes made to the parent version visible in the child). These changes may be either inserted, updated, or deleted features. This results in the creation of a new state that is then associated with the child version (e.g., state 4 in Figure 5). Note that it is possible that conflicts may be detected during reconciliation if a given feature has been modified in both the child version as well as the parent version. Additionally, if a feature is updated in one version and deleted in another, this is also a conflict (an update-delete conflict). When conflicts occur, the changes that are made in the parent version will take precedence by default (note that it is

equally reasonable to implement a reconcile process where the child version takes precedence by default). Thus, human intervention is oftentimes necessary in order to resolve the difference if any of the changes made in the child version (that are in conflict with the parent) are to take precedence. In sum, reconciling is the process of making all the changes that were made to a parent version visible in a child version.

Posting is conceptually the converse operation to a reconcile. Posting involves taking a child version that has been reconciled with its parent version, and making all the changes made in the child visible to the parent version. Conceptually, changes in the child are pushed up into the parent. Once two versions have been reconciled and posted (with one version assuming the role of descendent, and the other as the ancestor in both operations), the parent and child versions will represent the same instance of data within the versioned database (at least until another edit is made to either version).

Version reconciliation (and conflict detection) may be implemented using queries against the underlying relational database that allow all inserts, updates, and deletes that occur between two states in the state tree to be detected. We term these queries 'difference queries' (detect the differences between two states). Note that for a conflict to occur between a feature in a child and parent version, the difference queries between the two states associated with the child and parent version relative to their common ancestor state (e.g., state 0 in Figure 5) must show that either the feature was either updated in both, or updated in one and deleted in the other state.

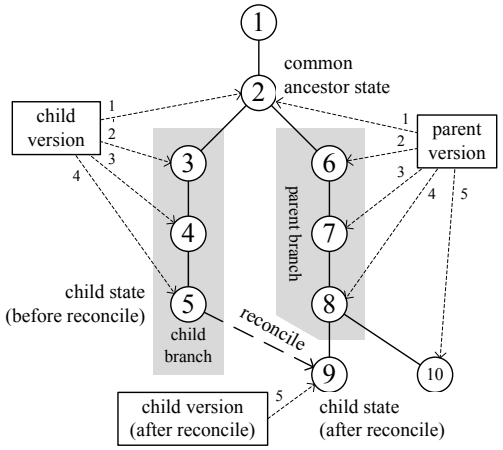

Fig. 6. Example state tree showing the interaction between child and parent versions

Figure 6 depicts a simple example highlighting the interaction between states, versions, and a reconcile. In the example assume that the parent version corresponds to state 2 (as indicated by the dashed arrow labeled "1" between the parent version and the circle labeled "2" (not that states correspond to labeled circles in the diagram). If a child version is now created, it will also reference state 2 (also depicted by a dashed arrow labeled "1" between the child version and state 2). State 2 also becomes what is termed the common ancestor state between the parent and child version. Assume that the child version is then edited three times. Each edit operation (an

atomic set of edits) results in a new state; in this instance, states 3 through 5. At the end of the three edit operations, the child version will be referencing state 5. Following the edits to the child, assume that the parent version also has three edits made to it. This results in the creation of states 6, 7, and 8, with the parent version referencing state 8 following the edits. Now assume that the child version is reconciled with the parent version. The reconcile will require that the edits made in the parent version (essentially, the edits represented by states 6 – 8 in what is termed the parent branch) are made visible to the child version. This is accomplished by creating a new state (state 9) off of state 8, and pushing all the changes that have occurred in the child branch (states 3 – 5) into state 9, and making the child version reference state 9. Finally, the example concludes with another edit being made to the parent version, resulting in a new state (state 10) being created off of the last state that the parent version referenced (state 8 in the figure).

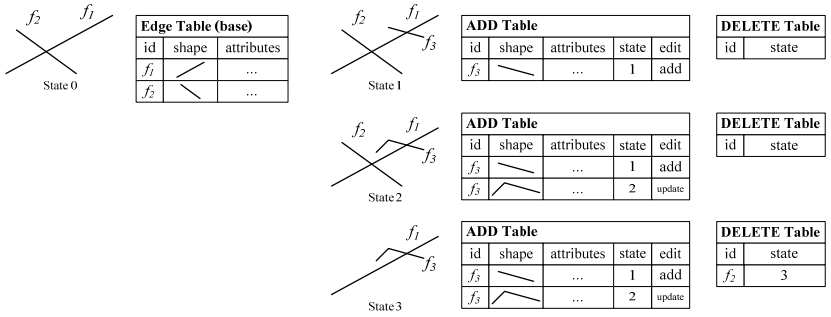

Fig. 7. Simple edit scenario highlighting the ADDs, DELETEs, and the base table

3.2 Implementation Details

Versions are associated with a state identifier that corresponds to each update that occurs in the view. The state identifiers are unique and map to a set of updates corresponding to a single logical edit. For each state, the database keeps information about the modification type (either an insert, update, or delete). The ADDs table contains information related to inserts and updates, while the DELETEs table maintains the deletes (Figure 7). These two tables are collectively referred to as *delta tables*. One set of delta tables is associated with each base table in the versioned database. Thus, if a data model contained two tables, one representing parcels, and the second representing owners, there would be four additional tables necessary to represent the two sets of delta tables. A versioned dataset, therefore, consists of the original table (referred to as the *base table,* which corresponds to State 0), plus the two delta tables. The versioned database keeps track of which version the user is connected to. In addition, when modifications are made to the data, the versioning system populates the delta tables as appropriate. When a user queries a dataset in a versioned environment, the system assembles the relevant rows from the base table and the delta tables to present the correct view of the data for that particular version.

4 Versioned Network Models

Network models, with their associated network connectivity indexes, dirty areas, and dirty objects, introduce complexities into the standard reconcile and post processes within a versioned database (as described in Section 3). The primary cause of this complexity is the fact that inconsistent network indexes may occur when an edited child and parent version are reconciled. This is irrespective of whether or not each version has its full extent rebuilt (i.e., no dirty areas or objects).

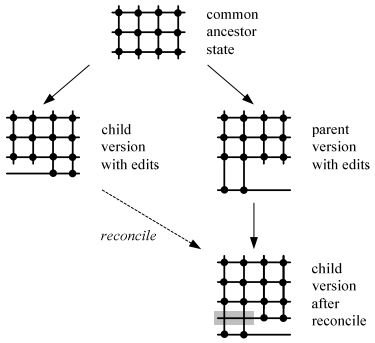

Fig. 8. Example highlighting a reconcile that results in an inconsistent network index (the inconsistent index is depicted by the shaded region at the bottom of the figure)

Consider the situation shown in Figure 8 (an annotated state tree is depicted – the common ancestor state refers to the state that the parent version was pointing to when the child version was originally created). In this example, assume that the network is clean; no dirty areas or objects exist with the features and the network index being in a consistent state. Edits are then made to both the parent and child versions. In the child version, the network is augmented in the southeast direction, while in the parent version the network is augmented toward the southwest. Assume that the network has been incrementally rebuilt following all edits in each version (i.e., no dirty areas or objects exist). In the figure, connectivity between line features is represented by the small black circles. As can be observed, both the parent and child versions have planar connectivity.

If the child version is then reconciled against the parent version, new edits made in the parent version are made visible in the child version. This is depicted in the southeast corner of Figure 8. Making these new features visible in the child version results in an inconsistency between the features and the network connectivity index as depicted in the area enclosed by the gray area. Thus, we observe a simple situation where two versions that are completely rebuilt can have a network connectivity index inconsistency following reconciliation. For this reason, the version reconcile process must be augmented to handle networks correctly.

4.1 Dirty Area and Object Management During Reconciliation

As has been discussed, versioning of network models requires additional functionality on top of the versioning scheme for simple feature classes. This is due to the fact that

the model includes both: (i) a feature space with features modeling real world objects, and (ii) a logical network where connectivity information about these features is stored. The connectivity information has to be kept consistent with the state of the feature space during the process of reconciliation when new features have been introduced or existing ones have been updated or deleted in the child version as a result of the reconciliation. All these modifications introduce changes in the connectivity inside the feature space of the network model, which have to be reflected in the logical network.

There are two general approaches to solve this problem. The first one employs the concept of reactive behavior which is applied to the network and has been used in the ArcGIS geometric network model [2]. The reactive behavior refers to the logical connectivity network reacting automatically to the changes in the feature space. Thus, the process of reconciliation will require the maintenance of the connectivity information. This entails both logical networks (in the child and parent versions) being analyzed concurrently during the reconciliation process and merged together in the resultant child version. The main disadvantage to this approach is the complexity of the problem (analyzing and merging graphs) which itself can deteriorate the performance of reconcile.

To avoid this disadvantage when reconciling a Network model, we choose to employ another strategy which we call the active approach. Instead of ananlyzing and restitching the connectivity information during reconcile, we instead utilize the incremental network rebuild algorithm discussed in [1]. We relax the requirement that the connectivity network must always reflect the state of the feature space. From a connectivity perspective, the logical network is allowed to be in an incorrect state; however, the regions of inconsistency are tracked by marked with dirty areas (or dirty objects in the case of turn features).

Dirty area (and object) management becomes a key concept in the versioned network model. In order to ensure that the incremental rebuilding of the network index is properly handled, we rely upon a strategy where dirty areas or objects are generated for the areas where spatial features or turn features are modified (created, updated, or deleted). The user may then choose to rebuild the network over the portions of the network where these dirty areas are introduced as a byproduct of the reconcile at a time of their choosing. More specifically, we may summarize the rules related to the handling of dirty areas and objects during a reconcile as follows:

1. All dirty areas and objects that are present in the child or parent that do not exist in the common ancestor state (i.e., before the child and parent were edited) remain in the child version after reconcile (depicted in the left side of Figure 7).

2. All dirty areas and objects that exist in the common ancestor state but do not in the child (i.e., an incremental network rebuild in the child) will exist in the child following the reconcile (depicted in the right side of Figure 7).

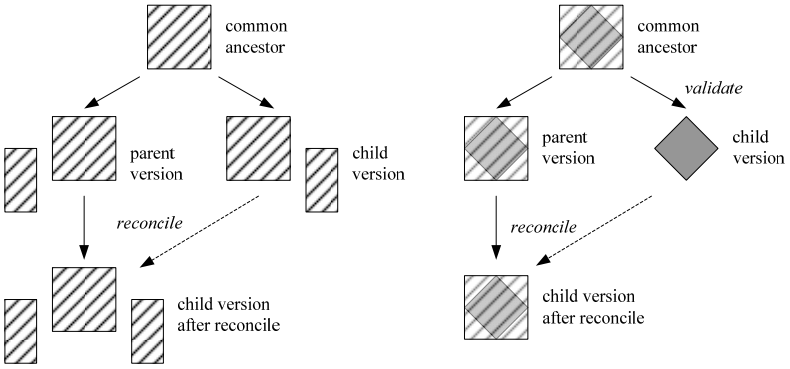

Fig. 7. Example of dirty area management policies 1 and 2. In this figure (and all others), dirty areas are represented by the crosshatched rectangles.

3. All dirty areas and objects that exist in the common ancestor state but do not in the parent version (they were validated) will not exist in the child version following the reconcile.

4. All dirty areas and objects created in the child version, irrespective of whether or not they exist at the time of reconciliation, will exist following the reconcile.

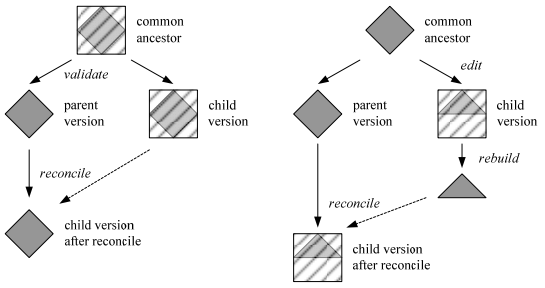

Fig. 8. Example of dirty area management policies 3 and 4.

5. All dirty areas and objects created in the parent version will only exist in the child version following the reconciliation if they exist at the time of reconcile. This situation is shown in Figure 9.

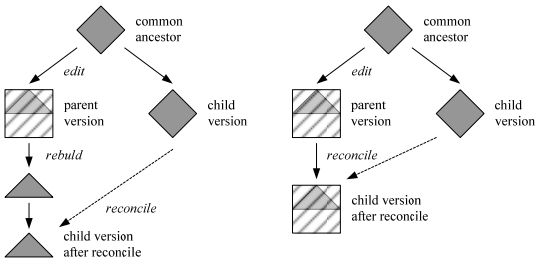

Fig. 9. Example of dirty area management policy 5. The left side depicts how a dirty area that no longer exists in the parent at the time of reconcile will not exist in the child following the reconcile. The right side depicts the opposite situation where the dirty area exists at the time of reconcile.

4.2 Examples

We illustrate the work of the versioning scheme using the example shown in Figures 10 and 11. In the common ancestor version (see Figure 10a) a new feature f_7 has been added to the feature space. Because the network model is versioned, all new features inserted in the system after the registration of the model as versioned are stored in the Add table. The common ancestor version the reactive behavior of the network creates dirty area d_1 around feature f_7 in order to keep track of the modifications in the feature space. Since the area has not been rebuild with the incremental rebuild algorithm the new line feature f_7 has not been reflected in the model.

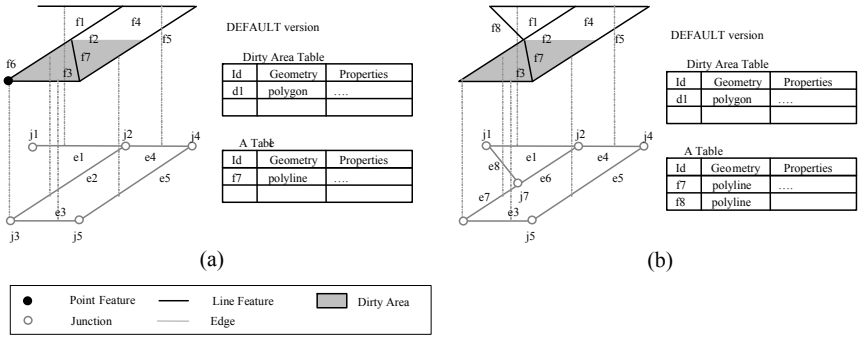

Fig. 10. Version example – common ancestor and parent versions

In the next step a child version is created. The dirty area in the parent version (see Figure 10b) associated with the feature f_7 remains unclean while in the child version (see Figure 11a) it is cleaned and the connectivity information for feature f_7 is reflected in the connectivity graph. In the parent version a new feature f_8 has added. The dirty area around it was cleaned and the feature is correctly described in the network model with edge e_8. The new feature however is not visible to the child version since it is stored locally in the parent add table. The modifications in the parent logical network are also not visible to the child version.

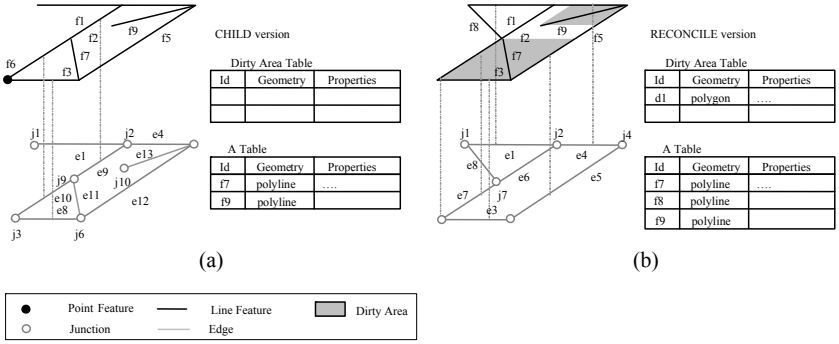

Fig. 11. Version example – child and reconcile versions

In the child version a new feature f_9 has been added as well (the generation of the feature primary keys - f_7, f_8, f_9 is done by a centralized sequence in the database so there are no features with the same id in the different versions). The dirty area for this feature has been cleaned and it is properly reflected in the child logical network.

Finally we perform the process of version reconciliation over the parent and child versions (Figures 10b and 11a). The content of both Add tables is merged so the reconciled version has feature f_7, f_8 and f_9 in its Add table. The logical network from the parent is copied in the child version. The connectivity information for feature f_8 has already been reflected so no dirty area is created for this feature (version rule 3). The connectivity information for feature f_7 which was in the child version of the logical network is lost so it has to be reestablished. This is indicated by recreating the dirty area around f_7 (version rule 2). In a similar way the connectivity information for feature f_9 is lost so it has to be cleaned as well (version rule 4).

5 Implementation Experiences

The proposed versioning scheme for network models has been implemented and will be shipped with the next upcoming ESRI ArcGIS product. It has been used to provide multiuser environment for large continental wide network models, including a model derived from the set of features representing the full street network within the entire continental United States (35.9 million line features). Similarly sized networks were constructed for all of Europe.

6 Conclusion

In this paper, we explored the difficulties of managing large network models in a multiuser environment and presented solutions to address these problems using a flexible versioning scheme. Many current applications (i.e. mapping services) involve

network based capabilities (e.g., underground pipeline management). It is thus indispensable for users to be able to edit the same geographic data simultaneously with long transactions and to resolve conflicts between these edits. Traditional DBMS have not been tailored to meet the special needs that are required in handling large network models.

Taking into account the dynamic nature of network models, we presented innovative versioning schemes that facilitate the notions of dirty areas and dirty objects (used already for maintaining the dynamic network model). The following summarizes key features of the versioning scheme:

- The flexible reconciling rules allow the definition of a resolving mechanism between conflicting edits according to user needs.
- In addition, the utilization of dirty areas/objects minimizes the overhead of tracking editing history.

We have implemented the versioning scheme presented in this paper within the well-established ArcGIS development framework. The proposed ideas have proven to be efficient methods in handling concurrency control of large network datasets. As a future research direction, we plan to extend our versioning scheme in a distributed database environment.

References

1. Bakalov, P., Hoel, E., Heng, W.L., and Tsotras, V.: Maintaining Connectivity in Dynamic Multimodal Network Models. In Proceedings of the International Conference on Data Engineering (ICDE 2008), Cancun, Mexico, April 2008
2. Borchert, R.: Geometric Network: What Is It and How to Make It? In Proceedings of the 23rd Annual ESRI User Conference, San Diego, July 2003
3. Caldwell, T.: On Finding Minimum Routes in a Network with Turn Penalties. Communications of the ACM, 4 (2), 1961
4. Cho, H.-J., and Chung, C.-W.: An Efficient and Scalable Approach to CNN Queries in a Road Network. In Proc. of the 31st Intl. Conf. on Very Large Data Bases (VLDB 2005), Trondheim, Norway, August 2005
5. Caliper Corporation: TransCAD Transportation GIS Software Reference Manual. Caliper Corporation, 1996
6. Dittrich, K., and Lorie, R.: Version Support for Engineering Database Systems. IEEE Transactions on Software Engineering, 14 (4), April 1988
7. Dueker, K., and Butler, A.: GIS-T Enterprise Data Model with Suggested Implementation Choices. Journal of the Urban and Regional Information Systems, 10 (1), 1998
8. Easterfield, M., Newell, R., and Theriault, G.: Version management in GIS - applications and techniques. In Proc. of the European Conference on Geographical Information Systems (EGIS 1990), Amsterdam, April 1990
9. ESRI: *Building a Geodatabase*. Prepared by Environmental Systems Research Institute, ESRI Press, Redlands, CA, 2002
10. Evans, J., and Minieka, E.: Optimization Algorithms for Networks and Graphs. Dekker, Marcel Incorporated, 1992
11. Goodchild, M.: Geographic Information Systems and Disaggregate Transportation Modeling. Geographical Systems, 5 (1–2), 1998

12. Hage, C., Jensen, C., Pedersen, T., Speicys, L., and Timko, I.: Integrated Data Management for Mobile Services in the Real World. In Proceedings of the 29th Intl. Conf. on Very Large Data Bases (VLDB 2003), Berlin, September 2003
13. Hoel, E., Menon, S., and Morehouse, S.: Building a Robust Relational Implementation of Topology. In Proceedings of the 8th International Symposium on Spatial and Temporal Databases (SSTD 2003). Santorini Island, Greece, July 2003
14. Hoel, E., Heng, W.L., and Honeycutt, D.: High Performance Multimodal Networks. In Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD 2005), Angra dos Reis, Brazil, August 2005
15. Jensen, C., Pedersen, T., Speicys, L., and Timko, I.: Data Modeling for Mobile Services in the Real World. In Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD 2003), Santorini Island, Greece, July 2003
16. Katz, R.: Toward a Unified Framework for Version Modeling in Engineering Databases. ACM Computing Surveys, 22 (4), 1990
17. Longley, P., Goodchild, M., Maguire, D., and Rhind, D.: Geographical Information Systems, Principles, Techniques, Applications and Management. Wiley, 1999
18. Mainguenaud, M.: Modeling of the Geographical Information System Network Component. International Journal of Geographical Information Systems, 9 (6), 1995
19. Menon, S., Aronson, P., Brown, T., Muller, M., Ryden, K., and Morehouse, S.: Requirements and Design Considerations for Versioned Geographic Data Management. Unpublished manuscript. ESRI, Redlands, CA, July 2000
20. Morehouse, S.: ARC/INFO: A Geo-relational Model for Spatial Information. In Proceedings of AUTOCARTO 7, Washington, DC, March 1985
21. Oracle Corp: Oracle Database 10g: Oracle Spatial Network Data Model: technical white paper, May 2005
22. Papadias, D., Zhang, J., Mamoulis, N., and Tao, Y.: Query Processing in Spatial Network Databases. In Proc. of the 29th International Conference on Very Large Data Bases (VLDB 2003), Berlin, September 2003
23. Peuquet, D., and Duan, N.: An Event-based Spatiotemporal Data Model (ESTDM) for Temporal Analysis of Geographic Data. International Journal of Geographical Information Science, 9 (1), 1995
24. Ralston, B.: GIS and its Traffic Assignment: Issues in Dynamic User-optimal Assignments. Geoinformatica, 4 (2), 2000
25. Sciore, E.: Versioning and Configuration Management in an Object-oriented Data Model. International Journal on Very Large Data Bases, 3 (1), 1994
26. Shahabi, C., Kolahdouzan, M., and Sharifzadeh, M.: A Road Network Embedding Technique for k-nearest Neighbor Search in Moving Object Databases. In Proceedings of the 10th ACM International Symposium on Advances in Geographic Information Systems (ACMGIS 2002), McLean Virginia, November 2002
27. Shekhar S., and Liu, D.-R.: Ccam: A Connectivity-clustered Access Method for Networks and Network Computations. IEEE Transactions on Knowledge and Data Engineering, 9 (1), 1997
28. Speicys, L., Jensen, C., and Kligys, A.: Computational Data Modeling for Network-constrained Moving Objects. In Proceedings of the 11th ACM Intl. Symp. on Advances in Geographic Information Systems (ACMGIS 2003), New Orleans, November 2003
29. Stokes, A., Balasubramanian, S., and Harrison, S.: Building Versioning Applications with the Oracle Internet File System. Oracle Technical Brief, Oracle Corporation, December 2000
30. Worboys, M., Hearnshaw, H., and Maguire, D.: Object-oriented Data Modeling for Spatial Databases. International Journal of Geographical Information Systems, 4 (4), 1990