Incremental Posting with Versioned Data

An ESRI® Technical Paper • May 2005

ESRI 380 New York St., Redlands, CA 92373-8100, USA • TEL 909-793-2853 • FAX 909-793-5953 • E-MAIL info@esri.com • WEB www.esri.com

Copyright © 2005 ESRI
All rights reserved.
Printed in the United States of America.

The information contained in this document is the exclusive property of ESRI. This work is protected under United States copyright law and other international copyright treaties and conventions. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage or retrieval system, except as expressly permitted in writing by ESRI. All requests should be sent to Attention: Contracts and Legal Services Manager, ESRI, 380 New York Street, Redlands, CA 92373-8100, USA.

The information contained in this document is subject to change without notice.

U.S. GOVERNMENT RESTRICTED/LIMITED RIGHTS

Any software, documentation, and/or data delivered hereunder is subject to the terms of the License Agreement. In no event shall the U.S. Government acquire greater than RESTRICTED/LIMITED RIGHTS. At a minimum, use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in FAR §52.227-14 Alternates I, II, and III (JUN 1987); FAR §52.227-19 (JUN 1987) and/or FAR §12.211/12.212 (Commercial Technical Data/Computer Software); and DFARS §252.227-7015 (NOV 1995) (Technical Data) and/or DFARS §227.7202 (Computer Software), as applicable. Contractor/Manufacturer is ESRI, 380 New York Street, Redlands, CA 92373-8100, USA.

ESRI, the ESRI globe logo, ArcGIS, www.esri.com, and @esri.com are trademarks, registered trademarks, or service marks of ESRI in the United States, the European Community, or certain other jurisdictions. Other companies and products mentioned herein are trademarks or registered trademarks of their respective trademark owners.

Incremental Posting with Versioned Data

An ESRI Technical Paper

Contents **Page**

Design Parameters 2
Design Practices 2
Relationships 2

Solution Alternatives 3
Using Attribute Values 3
Editing Features in the Design Version 4
Posting the Design 5
Additional Considerations 5
Place Design Updates in Separate Object Classes 6
Editing Features in the Design Version 7
Posting the Design 7
Additional Considerations 7
Copy Selected Updates to a Parallel Version 8
Editing Features in the Design Version 10
Posting the Design 10
Additional Considerations 11
Use Child Version to Post Selected Updates 12
Editing Features in the Design Version 14
Posting the Design 14
Example Scenario 16
Additional Considerations 18
Using a Control Version 19
Modeling Deleted Features 23
Creating a Proposed View Version 24

Conclusion 25

Incremental Posting with Versioned Data

The incremental posting of design versions has long been a requirement for users who create potential designs for new facilities. These users consist of those in the utility industry who create designs for new electrical, gas, water, sewer, cable, and telecommunications facilities and those in the transportation industry who create new designs for roads and railroads. Regardless of the user, the requirements are the same. Designs often encompass many phases of a project, which are actually built on the ground at different time intervals. As a particular phase of a design is implemented, there is a need to post the work to date so the database is current. For instance, the design for a new subdivision may include new roads along with various utilities such as electric, gas, and water. As portions of the subdivision are opened to the public, the roads and utilities in place need to be entered into the system to keep the database current. The entire design cannot be posted because not all facilities have been built at the time of posting and, in some cases, never will be. Those facilities that have been built need to be posted so other applications (outage management, one call, billing systems, etc.) have the most up-to-date data to work with.

From an ArcGIS® standpoint, the requirement is to be able to select features to post from an open design version. The selected features will represent a subset of the features that have actually been edited in the design version. After posting of the selected features, the version is to remain open so design work can continue. At a later date, another subset of features or all remaining edits need to be available for posting.

The purpose of this case study is to look at four potential solutions for the incremental posting issue.

- Using attribute values
- Using separate feature classes
- Using a parallel version
- Using a child version

Although the requirements are relatively clear, the solutions are not. The solution to implement varies depending on the complexity of the database. If the database is composed of simple features with no relationships, the solution is rather simple. However, the inclusion of relationships in the data model exponentially increases the complexity of the solution. Another factor in the complexity of the solution is the need to deal with modified and deleted features in the design.

Design Parameters

As mentioned earlier, the general setup and complexity of the data model, along with the type and nature of designs that need to be created, go a long way in determining the method to use and the complexity of this method. There are two specific areas that warrant mentioning. The first is the design practices themselves and whether or not the modification and deletion of features are necessary as part of the design. The second area is relationships and how these relationships are modeled.

Design Practices

As mentioned previously, performing an incremental post from a design is relatively straightforward when the design consists only of new features. Suppose you decide to implement the attribute approach to incremental posting (this approach will be discussed in more detail later in the document). The process for incrementally posting the design would be to first post all features with an attribute value that indicates they are design features. When a portion or phase of the design is completed, all features in that section of the design would have their attribute value switched from "design" to "as-built," and the Editor Connect command could be run to ensure network features are connected to the network.

If designs can include the deletion of features, then you must determine how to post a design with a deleted feature without deleting that feature until that portion of the design is built (or possibly undelete the feature if the design is not approved for construction). You could have an attribute value that designates the feature as deleted in the design, but then you would have to add additional symbology (or possibly a definition query for the layer) to show the deleted design features in a different color (or possibly not at all). If you are not using the attribute approach, then you need to add the deleted feature back into the version if the delete is not to be posted as part of the current incremental post.

If the engineers want the design to include modified features, then you would design a solution that allows different sets of the same features to exist. Depending on which incremental posting solution you choose, you must either allow different sets of the same features to be in the default version (with attribute values to designate the as-built and design features) or determine a way to reset a feature back to the original value if it is not to be posted during the current incremental post.

These problems can all be overcome with a variety of procedures, but hopefully it is apparent how much more complex the solution becomes when deleted and modified features are included in the design. One solution to simplifying the process is to use annotation or some other form of notes in the design to indicate the deletion or modification of a feature without actually performing the edit.

Relationships

Relationships are a standard and very useful component of the geodatabase model. They allow the user to link features and table rows to other features and table rows. For instance, a relationship must exist between a feature and a piece of annotation for that annotation to be "feature-linked." This relationship allows the annotation string to be automatically updated when the related features are changed, moved when the features are moved, and deleted when the features are deleted. How relationships are defined and, in particular, the fields that are used as keys have a large impact on the incremental posting solution.

J-9443

Building on the Design Practices section above, consider a transformer bank feature (origin class) with a one-to-many simple relationship to a transformer units table (destination class). As part of a road widening design, a set of transformers and their related records in the units table are deleted and replaced with a new set of transformers. When the first phase of this project is approved and built, not all of the deleted transformers are part of the first incremental post. As a result, the solution must undelete the transformers. Difference cursors can determine the deleted features, and you can add the deleted transformers back into the design. However, by adding the features, new object IDs are assigned. If your relationships are built on object IDs, then your solution must be robust enough to reestablish the relationship to the units in the related table. If your relationship is not built on object IDs, then the solution is not as complex.

As with the design practices, relationship issues can be solved, but a more complex solution is required. All these potential issues will be discussed in more detail in the four sections on the proposed solution options.

Solution Alternatives

Using Attribute Values

The first approach to be discussed is the use of attribute values to designate what has and has not been built. For example, suppose you are working with electrical data. Portions of the design you have been working on are "energized" (carrying electricity), while other parts have not yet been built. The goal, of course, is to make the energized facilities available to the system as soon as possible so, for example, you do not have people working in the field around energized lines they do not know are there.

The attribute approach to this problem is to create an attribute field for each electrical feature class, which holds a value designating whether the feature is active (energized) or not. For example, the object model for a Fuses feature class would look something like figure 1. The BuildStatus field contains the values representing the status of the feature. A coded value domain could be created and used with the field to ensure only appropriate values are entered. This field needs to be added to the object class of every feature and related table that can hold design objects.

Figure 1

Simple Junction feature class
Fuses

Geometry *Point*
Contains M values *No*
Contains Z values *No*

Field name	Data type	AllowNulls	Precision	Scale	Length
OBJECTID	Object ID				
DSCKTLABEL	Double	Yes	0	0	
TAG	String	Yes			15
TYPE	String	Yes			3
SUBTYPE	String	Yes			6
PHASES	String	Yes			3
FUS_TYPE_CD	String	Yes			3
FUS_HSNG_AMPS	Short integer	Yes	0		
FUS_LINK_CD	String	Yes			1
FUS_LINK_AMPS	Short integer	Yes	0		
NSTATUS	String	Yes			1
ANGLE	Double	Yes	0	0	
SYMBOL	Long integer	Yes	0		
Enabled	Short integer	Yes	0		
BUILDSTATUS	String	Yes			10
SHAPE	Geometry	Yes			

Coded value domain
BuildStatus
Description
Field type *String*
Split policy *Default Value*
Merge policy *Default Value*

Code	Description
ASBUILT	Facilities in the ground
DESIGN	Facilities under design
MODIFIED	Modified in design
DELETED	Deleted in design

The BuildStatus field was added to the standard set of fields to hold the current status of the feature. The BuildStatus-coded value domain holds the valid values for this field.

A network weight would be established on the BuildStatus field so tracing could occur as required on both the as-built and design features. The weight would be necessary for tracing regardless of whether you are attempting to trace on the design features, the as-built features, or both. Using a weight also requires that the user know the correct filter values to use or that there be custom code written for making these settings.

As with all potential solutions, one of the big considerations is whether or not designs will include modified and deleted features. With the attribute values approach, including deleted features in a design is not complicated. It simply requires changing an attribute and most likely establishing a network weight so tracing can be accurately performed on the design features (assuming tracing is a requirement). Modified features are more complicated as moving a feature would require the maintenance of both the as-built and design renditions of the features in the default version. An alternative to both modifying and deleting features in the design is to add annotation or other graphics to indicate which features to update when the design is built. Utilities that have implemented this solution solve the problem by creating new or modified facilities at an offset from the original. When the update is posted, the features are moved to their correct location and the original is deleted or marked as being out of service.

Editing Features in the Design Version

The approach for editing features in the design version would be as follows:

- **Adding Features**—All features and related objects added to the design version would have their BuildStatus value set to DESIGN to indicate the feature does not exist on the ground yet.

- **Deleting Features**—Objects that are deleted as part of the design would actually remain in the design but would have their BuildStatus set to DELETED. The feature would need to remain connected to the network, though, so tracing on the as-built features could be accomplished with the design features present. A network weight

would be used to ensure tracing did not occur on deleted design features when checking connectivity for the design.

- **Modifying Features—**Objects that are modified also remain in the design and have their BuildStatus set to MODIFIED. In addition, a new feature is added to represent the updated state of the feature. The BuildStatus of this feature is set to DESIGN to indicate it is only part of the design version. An alternative to actually modifying features is to add annotation or other graphics to denote the update (particularly a move of network features) in the design.

Posting the Design

Once a design has been completed, the version can be posted to default using the standard out-of-the-box Reconcile and Post commands. Some organizations require that all versions be posted to the database every night, so all work to date on a design would be posted at the end of the day. The next day the engineer would create a new version and continue working on the design. One benefit of posting at the end of each day is the minimization of conflicts as there are fewer changes in each area of a design during any given day. At the beginning of each day when the design engineer creates a new version, the version contains all the updates posted to the system from the previous day. In this manner, the designer is always able to see the most recent updates to the system.

Timing of the posting of the design does not necessarily have to correspond to the completion, approval, or construction of the design. All design features can be held in the default version with attribute values designating which features are as-built or part of a design. When a phase or portion of a design has been approved and built on the ground, the design engineer creates a new version and sets the attribute values to reflect the current condition of these features and table rows. (For example, using our sample data model, the BuildStatus value of each feature and table row would be changed to ASBUILT.) It may also be necessary to run the Editor Connect command to ensure connectivity between the newly energized features and the rest of the network. Objects whose BuildStatus is set to DELETED or MODIFIED would be removed from the system. These updates would then be posted to default.

As additional phases of the design are completed on the ground, the process of changing attribute values to ASBUILT, removing DELETED and MODIFIED features, and so forth, continues until all the design has been posted (or removed from the system if it does not end up being built).

Additional Considerations

- Symbology, definition queries, and stored displays must be defined to incorporate different representations of the features in the default version with attribute values as discussed. The BuildStatus field will indicate whether a feature is part of the as-built feature set or a design and should be used accordingly to obtain the desired display. Some organizations like to see deleted features displayed in gray or some other nonstandard color, while others do not want to see them at all, so approaches to symbolization will vary.

- Visualization of the design features can be problematic even with extra symbology as outlined above. If you have more than one alternative for the same design, the designs are essentially layered on top of one another, making it difficult to distinguish between them.

- Tracing is difficult and, at a minimum, requires network weights. To trace, you must also ensure all updates are connected to the network, and so forth. For most scenarios, this alternative should not be considered if extensive tracing of the design is required.

- On the plus side, this option does not require designs to live in versions for extended periods. Partial posting is simply an attribute change.

Place Design Updates in Separate Object Classes

Another approach currently being implemented by some users is the use of separate object classes to hold design features. The design object classes are essentially copies of the standard object classes, with the addition of an object ID field, which links updated or deleted design features back to the originals, and a field designating whether the object is new, modified, or deleted (basically the same as the BuildStatus field discussed in the attribute approach above). New objects are simply added to the design classes. Modified and deleted objects are copied from the standard object classes to the design object classes, with the object ID link set to that of the original object.

Figure 2

Simple Junction feature class **Fuses**

Geometry Point
Contains M values No
Contains Z values No

Field name	Data type	AllowNulls	Precision	Scale	Length
OBJECTID	Object ID				
DSCKTLABEL	Double	Yes	0	0	
TAG	String	Yes			15
TYPE	String	Yes			3
SUBTYPE	String	Yes			6
PHASES	String	Yes			3
FUS_TYPE_CD	String	Yes			3
FUS_HSNG_AMPS	Short integer	Yes	0		
FUS_LINK_CD	String	Yes			1
FUS_LINK_AMPS	Short integer	Yes	0		
NSTATUS	String	Yes			1
ANGLE	Double	Yes	0	0	
SYMBOL	Long integer	Yes	0		
Enabled	Short integer	Yes	0		
SHAPE	Geometry	Yes			

Simple Junction feature class **Fuses_Design**

Geometry Point
Contains M values No
Contains Z values No

Field name	Data type	AllowNulls	Precision	Scale	Length
OBJECTID	Object ID				
DSCKTLABEL	Double	Yes	0	0	
TAG	String	Yes			15
TYPE	String	Yes			3
SUBTYPE	String	Yes			6
PHASES	String	Yes			3
FUS_TYPE_CD	String	Yes			3
FUS_HSNG_AMPS	Short integer	Yes	0		
FUS_LINK_CD	String	Yes			1
FUS_LINK_AMPS	Short integer	Yes	0		
NSTATUS	String	Yes			1
ANGLE	Double	Yes	0	0	
SYMBOL	Long integer	Yes	0		
Enabled	Short integer	Yes	0		
BUILDSTATUS	String	Yes			10
OBJECTID_LINK	Long	Yes	0		
SHAPE	Geometry	Yes			

The design object class is the duplication of the original feature class plus the addition of two new fields (highlighted in yellow). The BuildStatus field holds the status of the design feature (new, modified, or deleted), while the OBJECTID_LINK field contains a reference to the object ID of the original entity for modified and deleted objects.

Assuming that tracing on the design features is a requirement, the design feature classes need to be registered as network feature classes. A network weight is necessary if there is a requirement to trace from the design features back into the as-built features. Along with the weight, it would also be necessary to have the design feature classes be part of the same network as the as-built feature classes, which can have some impact on the performance of the database as a whole.

Similar to the attribute approach outlined above, the separate object class approach can be accomplished without code. The copying of objects from the standard object classes to the design classes and back again for posting could be accomplished manually if desired. However, most implementations of this approach would include at least some relatively

simple application code to facilitate the process. The application code may also be necessary to ensure relationships remain intact when added and modified features are moved from the design feature classes to the standard feature classes.

Editing Features in the Design Version

The approach for editing features in the design version would be as follows:

- **Adding Features—**All features and related objects added to the design version would be added directly to the design feature classes. The BuildStatus value of these objects would be set to DESIGN to indicate the feature does not exist on the ground yet.

- **Deleting Features—**Objects that are to be deleted as part of the design are copied from the standard object class to the design feature class. The OBJECTID_LINK field in the design feature class is set to the object ID of the standard feature for reference purposes. The BuildStatus value is set to DELETED to indicate the object is being deleted in the design.

- **Modifying Features—**Objects that are modified are also copied from the standard object classes to the design object classes. The OBJECTID_LINK value is set to the object ID of the original object so it can be found during the posting process. The BuildStatus value is set to MODIFIED to indicate the object is being changed as part of the design.

Posting the Design

When a portion of the design is ready for posting, the updates need to be moved from the design feature classes to the standard feature classes. All added objects (BuildStatus = DESIGN) are simply copied from the design object classes to the standard object classes. The OBJECTID_LINK value of the features in the design object classes with the BuildStatus = DELETED or MODIFIED are used to select the original features in the standard object classes. The original features are then deleted from their respective object classes. Modified features are copied from the design object classes to the standard object classes with some application code most likely keeping track of relationships. Once all the updates being posted have been moved from the design classes to the standard classes, they can be deleted from the design classes. At this point, the version can be posted.

Assuming that some application code will be used to move updates between feature classes, it would follow that the updates to be posted would be identified by a selection set. The application code would loop through the selection set copying and deleting features as required in preparation for reconciling and posting.

Additional Considerations

- As with all the alternatives being looked at, some form of symbology needs to be set up to identify the features under design and whether they are being added, modified, or deleted. This alternative may have some additional symbology requirements since there are actually two feature classes involved for each feature type (two fuse classes, etc.).

- Some application code is most likely required to move updates back and forth between the object classes and to make sure relationships stay intact.

- This approach has the additional overhead of a complete duplication of object classes for those entities that may be part of a design.
- On the plus side, this option minimizes the need to maintain outstanding versions for designs. All design features are maintained in separate feature classes so it is easy to distinguish these features from the as-built features.

Copy Selected Updates to a Parallel Version

Another suggested approach to incremental posting is the copying of updated features to post from the design version to a parallel version. (A parallel version would be a new version created from the same parent as the design version.) Features that are copied would then be deleted from the design version. After the post of the parallel version, the design version is reconciled with the parent version to bring the posted updates back into the design.

Figure 3

The copy approach requires the updates to be copied to a new parallel version and posted to the default version. A reconcile then brings these updates back into the design version.

The basic steps for this solution are as follows:

1. Complete design work in the design version.
2. Create a new version (for this discussion, the parallel version) from the same parent version as the design version.
3. Add layers from both the design and parallel versions to your map and begin editing the layers from the parallel version.
4. Select the features to post from the design version and copy/paste them in the parallel version. You will need to select and copy/paste one layer at a time as the paste is based on the target layer.

5. Make sure network features are connected as desired and reconcile and post the version.

6. Stop editing on the parallel version and begin editing on the design version. (At this point you could remove the layers from the parallel version from the map and delete the version.)

7. Reselect the same features you did the copy/paste on and delete them.

8. Reconcile against the parent version to bring the features that were posted back into the design version. Save your edits.

Unlike the attribute values, the copy select option would most likely require custom code. Again, the complexity of the solution determines how you handle the modification and deletion of features in the design. If you only add new features in a design, then you only copy and paste features from the design to the parallel version when you want to incrementally post. The simple copy and paste gets more complicated when you are dealing with related features, though, as the copy/paste results in new object IDs being generated for each pasted feature. The instances of relationships based on object IDs would need to be reset. Custom code would be most efficient for resetting these relationships.

If you modify and/or delete features as part of your design, then you must decide how to move these updates over to the parallel version for posting. Deleted features can be determined by performing difference queries against the parent version, and the user could then be presented with a dialog box for specifying which features to delete from the parallel version for posting. Another method would be to use attribute values to designate deleted features (without actually deleting them), and the user could then select the features to delete in the design version. Custom code would ensure these selected features are removed from the parallel version before posting. For the implementation discussion below, assume the features are actually being deleted in the design version. For some, this assumption would not be appropriate because there is a requirement to show deleted features on the design so field crews know what to remove. This requirement will be discussed in more detail later.

For modified objects, you can delete the original version of the object in the parallel version, then copy over the modified object from the design version, or you could use code to update the feature in the parallel version based on the current settings of the feature in the design version. The advantage of the update approach is that the object ID does not change and, therefore, you do not need to worry about affecting relationships. For the implementation discussion, assume that modified objects will be moved from the design version to the parallel version by application code that updates the object.

The recommended approach for this option is writing application code to assist in the implementation. Copying, modifying, and deleting features while at the same time ensuring related objects stay in sync could possibly be accomplished without application code, but the logistics make it difficult. Correctly written application code makes the task more manageable.

Editing Features in the Design Version

As discussed previously, the options chosen for the implementation of the copy features approach will dictate how editing needs to be done. Because you have chosen to delete features in the design version, there are no special steps that need to be taken during the creation of the design. There are no extra fields that need to be added or domains that need to be assigned. Editing can proceed as usual within the design version.

Posting the Design

The real effort for this solution is in the application code that would need to be created to move selected updates from the design version to the parallel version for posting. When users are ready to post a portion of their design work, they begin by selecting the features to be posted. After selecting the features, the application code runs through the following procedures:

1. Store the object IDs of the selected features and stop editing if necessary.

2. Create a new parallel version from the parent of the design version and begin editing on that version.

3. Use difference queries to determine what has changed within each object class.

   a. **Added Features—**Perform an insert into the same feature class in the workspace of the parallel version. Keep track of the original object ID and new object ID for resetting the link to annotation and other objects in object ID-based relationships. Loop through the feature-linked annotation classes in the database and turn off the automatic generation of annotation so you can copy the original annotation from the design. If object IDs are used for relationships, then code would need to be written to walk through the relationships and reset the link value. The recommendation here is to not use object IDs as the key values in relationships. It may also be necessary to do multiple passes through relationships to ensure links are reset in nested relationships (for instance, transformers to bank attributes and bank attributes to unit attributes). Annotation should be copied last using the cross-reference to the stored old and new object IDs for resetting the link to the correct features. Annotation is copied instead of automatically regenerated to capture any custom placement. After copying all features, reset the automatic add property of the feature-linked annotation classes as necessary.

   b. **Modified Features—**Loop through the fields in the modified features and apply the updates (including shape changes) to the same features in the parallel version.

   c. **Deleted Features—**Write the class name and object ID to a collection so they can be presented to the user for selection after the difference queries have been run against all updated object classes.

      For developers: The most efficient method for determining the edits in the design version is to create a new VersionDataChanges object, then initialize that object (IVersionDataChangesInit::Init) with the source (design version) and target (the version you wish to post to) versions. You can then use the

IDataChanges::GetModifiedClassesInfo method to get an enumeration of just the object classes with edits. These objects are new at ArcGIS 9. The ArcGIS 8.3 solution is to run the IVersionedTable::Differences method for each class and difference type (insert, update/update, etc.) combination. When using ArcGIS 8.3, the differences approach can be simplified by using a nonedited control version to compare against so you do not have to check for conflicts.

4. Reconcile and post the edits to the target version (presumably the default version).

5. Stop editing on the parallel version and begin editing on the design version (at this point the parallel version may be removed).

6. Use the stored set of object IDs to reselect the features that were posted and delete all added objects (features and rows). Do not delete the modified objects.

7. Reconcile the design version against the target version to bring the posted objects back into the design. Ignore the conflicts from the modified features (accepting the versions being brought down from the target) or use a filter to remove those conflicts.

8. Allow the user to resolve any other conflicts as necessary and save edits.

9. Repeat the process for posting additional incremental updates from the design version.

Additional Considerations

- Relationships (particularly those based on object IDs) are the wild card for this solution. Without relationships on the object ID, the solution is relatively straightforward as it is not necessary to synchronize relationships after the copy creates new object IDs. When you have relationships based on the object ID, you are required to write quite a bit of application code to keep everything in sync. Nested relationships are problematic as several passes through the updated objects have to be made to ensure each new object ID is found and the relationship is reestablished. Regardless of whether the object ID is used as a link, rows added to related tables need to be copied over to the temporary design version.

- As mentioned, feature-linked annotation is copied separately to maintain any custom placement of the text. You could allow the feature-linked annotation to automatically be created during the copy and paste, but then you would get the default position of the annotation string in the post and not the custom placement.

- When changing the shape of features in the parallel version based on updates in the design version, you may need to do some subsequent checks to ensure network connectivity. The Editor Connect command may be run on each modified feature after a shape change, but this should be done after all features have been modified.

- On the plus side, only as-built (energized) features are in default, so there is no confusion with design features. Tracing is fully supported within the design and from the design to the as-built features.

Use Child Version to Post Selected Updates

The child version approach to incremental posting is similar to the copy selected updates option in that a selection set is used to separate those features to be posted from those that are not to be posted. The difference is that instead of using a parallel version, a child version of the design version is used.

The basic premise of this approach is that a child version is created from the design version; therefore, it contains all the edits in the design version, so it is only necessary to remove what you do not want to post. As with the other solutions, the approach is quite simple if you are dealing only with new features. In that case, all you would need to do is select the features you do not want to post from the child version, remove them, and perform the reconcile and post. You can then remove the child version and continue working on the design. Because you are not copying and pasting features, there are no new object IDs to worry about and relationships are not affected. Of course, things are never that simple.

Figure 4

The child version approach creates a new version from the design version and removes everything that is not to be posted (based on a selected set of features). Reconciliation with the target version (parent) is necessary for additional incremental posts.

With the copy approach, the handling of deleted features is relatively straightforward in that you just did not delete the feature from the parallel version if you did not want to reconcile and post it. With the child version approach, you must determine how to undelete a feature if you are not ready to post that change during the current phase. The same is true for modifications. In the copy approach, you do not bring over the modifications if they are not to be posted, while in the child approach you must undo the modifications if they are not to be posted.

In the absence of a retrieve or restore function for a deleted object (this function does not currently exist because of the impact undeleting a feature would have on the logical network, relationships, etc.), there are two basic approaches you can use to model deleted features in the child version approach. The first approach would be to use an attribute value (in essentially the same way as deleted features are handled in the attribute values approach) to designate features that are deleted in the design. When the child version is created for posting, features to be deleted would then actually be deleted if you wish to post that change. The other approach would be to delete the feature as part of the design. The downside to this approach is that you would then have to go to a parent version or a control version to replace the feature if the delete was not being posted. As you probably guessed, the object would come back in with a new object ID when you added it back in, and related objects would have to be synchronized.

Using attribute values requires schema changes to the database for the addition of a field on each object class to maintain the status (as-built, design, proposed deleted, etc.). It also requires additional steps to ensure deleted network features are not used when performing tracking functions on the proposed features. These additional steps could be manually disconnecting the features from the network, running the Disconnect command on the selected deleted features, or setting the Enabled property on the features to False. Even with these additional steps, the approach of modeling deletes as an attribute update (i.e., not actually deleting the feature) is used in the sample code. The majority of people want to be able to display deleted features on the design plot, so the technicians doing the work in the field know what needs to be removed. This requirement will be discussed in more detail later.

As with the other solutions, modifications are the hardest to handle. Resetting attribute values to their original settings is easy to do, but repositioning features can be problematic when networks are involved. You basically have to make the assumption that the user is selecting the correct set of features to post and does not select services to post without the connected transformer, for example. The recommended approach is to use annotation or other graphics to denote the movement of features within a design (and not actually move them until posting). However, because many users will not be happy with that solution, another approach will be discussed. The second approach (the one implemented in the sample code) would be to use code to reset the values of the modified features that you do not want posted back with original values (this would include the SHAPE field). The sample code does not necessarily return the modified features to their original values but instead retrieves the values from the version being posted to. If the object being reset is modified in the version being posted to, then the modified values are used. The benefit of this approach is that no conflicts are generated when the version is posted.

Figure 4 shows an optional control version generated from the parent version (the version you plan to post to) at the same time as the design version. The purpose of the control version would be to keep track of the original values of features for use when resetting modified features that are not to be posted. By using a control version, you can maintain the original values. But if the same object is updated in the parent version, a conflict will arise. This is because even though you did not intend to update the feature in your design (or at least not to post that update), the act of setting then resetting the values makes the

feature appear as an update. When the object is also changed in the parent version, you get an UpdateUpdate conflict. This can be avoided by not using a control version and retrieving values from the version being posted to. This topic (control versions) will be discussed in more detail later.

As with the copy solution, it is necessary to have application code to make this solution functional. Sample code is provided for this solution to get you started.

Editing Features in the Design Version

The recommended solution involves the use of the BuildStatus field (or a similar field) in the sample code. A schema change is necessary to add the BuildStatus field to all object classes that might participate in the design. A network weight is also required on the BuildStatus field for toggling tracing on design and as-built features. With the BuildStatus field, the editing process must be modified to include the updating of this field for features that are added, modified, and deleted (objects are not actually deleted, but instead the BuildStatus value changes to DELETED). The updating of this field could be done manually, but it would probably be beneficial to have an extension that listened to the editor events and updated the field appropriately during adds and modifications. For deletions, the BuildStatus field can be simply updated manually.

If the BuildStatus field is not being used to monitor updates, then no changes to the editing process are necessary. Adds, deletes, and updates can proceed as they would normally.

Posting the Design

Assuming that application code will be used to implement this solution, the following steps apply to the procedures to be programmed (these are the steps followed by the sample code). When users are ready to perform an incremental post from the design version, they begin by selecting the features to post. The selection set needs to include the new, modified, and deleted (deletion marked by an attribute change) features to be posted. The application code would then follow these procedures.

1. Store the feature class and object ID of the selected features and stop editing if necessary.

2. Create a child version from the current design version and begin editing on that version.

3. Use difference queries to determine the edits in the design version. Loop through these edits and compare them to the stored selected set of features to determine what to post. Two loops are made through the features with the first loop excluding network junctions. The second loop then picks up the network junctions. This is done to ensure no network junctions are left behind in locations where there used to be other simple junctions (this can occur when the new features you are not going to post are removed). Features are addressed based on their modification type.

    a. Added features: Check the list of added features returned by the difference queries against the set of features selected by the user. Delete the features that are not found in the selected set along with the new objects related to these features. Change the value of the BuildStatus field for the new features being posted to ASBUILT.

b. Modified features: Check the list of modified features against the set of features selected by the user. Features that are not found in the selected set are not to be posted, so they need to be returned to their original state. Get the original value of the feature from the version you are posting to and reset the attribute and shape values of the updated feature. If the modified feature is being posted, then change the BuildStatus to ASBUILT.

c. Deleted features: Perform the delete on features that are selected and tagged for deletion. Because you are using attribute values to designate deleted features, no features should actually be deleted in the design before executing the application code. The one caveat to this might be features that are deleted as part of a Split operation. For instance, if the Split command is used to break a polyline feature, the database executes a delete and two adds. When the incremental posting application code is run against the version, a split then appears to be a delete. As a result, the application code displays a popup dialog when deleted features are found, and the user must select the deleted features to be posted. Another approach would be to use commands and tools other than Split to divide features. Some third-party applications already include procedures designed to resolve this issue.

For developers: The most efficient method for determining the edits in the design version is to create a new VersionDataChanges object, then initialize that object (IVersionDataChangesInit::Init) with the source (design version) and target (the version you wish to post to) versions. You can then use the IDataChanges::GetModifiedClassesInfo method to get an enumeration of just the object classes with edits. These objects are new at ArcGIS 9. The ArcGIS 8.3 solution would be to run the IVersionedTable::Differences method for each class and difference type (insert, update/update, etc.) combination. If using ArcGIS 8.3, the differences approach can be simplified by using a nonedited control version to compare against so you do not have to check for conflicts.

4. Reconcile and post the child version to the target version (presumably the default version). At this point you are done with the child version, so it can be removed.

5. Resume editing on the design version. Reconcile the design version with the target version to get it in sync with the features that have been posted. Allow the user to reconcile conflicts as necessary and save the edits.

6. *Optional:* If a control version is used for retrieving the original values of modified and deleted features not being posted, then it can be deleted after the design version has been reconciled. A new control version would then be created from the parent version to be used with the next incremental post.

7. Continue working on the design version, making additional incremental posts as necessary. The user can make additional edits as necessary on the design features that have already been posted. No conflicts should arise when posting these changes unless the features were also updated in the target version.

Example Scenario

Using the process outlined above and the sample code, the following example was examined.

Figure 5

The design is shown with symbology based on the setting of the BuildStatus field.

Figure 5 shows the new design. At the top of the display you can see features tagged as deleted, which are highlighted in grey. All new features are symbolized with line features as dashes, polygons as crosshatch, and point features in yellow.

The first phase of the new design is ready to be posted, so you select the features (including the deleted features) to be posted.

J-9443

Figure 6

The features to be posted are selected (highlighted in cyan).

To post features, you select them as shown in figure 6. Features to be deleted are also selected, since they have not been removed yet. Once you have the correct set of features selected, the application code is run. The application code will do the following:

- Create the child version.
- Remove updates not being posted.
- Perform the reconcile against the version to which you are posting.
- If there are no conflicts, the child version is posted then removed.
- Resume editing on the design version, reconciling it with the version to which you are posting.

J-9443

Figure 7

After the application has run, the symbology shows the features that have been posted and, therefore, are no longer pending.

With the final reconcile of the design version against the parent version, the newly posted updates are brought down. Figure 7 shows the change in symbology after the selected features have been posted, then brought back down into the design version with the reconcile.

Additional Considerations

- Assuming the attribute method is used for deleting features, it would be necessary to use additional symbology to show deleted and modified features appropriately. Alternative symbology would probably also be used for features added in the design.

- How to handle related objects may differ depending on how other application code deals with the deletion or modification of objects. For instance, application code may trigger the deletion of transformer attributes in a related table when the transformer is deleted even though the relationship is simple.

J-9443

- As mentioned, handling modified features that are not posted can be problematic. When resetting the shape value of a feature, it can potentially cause issues with the logical network, so it is important to check these features before the final post.

- It is desirable to automate the reconciling of the design version after an incremental post so this step is not inadvertently skipped. The sample code will automatically perform the reconcile after a successful post of the child version to the parent. Also, the removal of the previous control version and the creation of the new one (after reconciling the design version with the parent) should be automated if you plan to use control versions.

- The positive aspects of this alternative are the same as the parallel approach. Only as-built features are in default, so it is easier to distinguish design from as-built features. Tracing is fully supported between all features.

Using a Control Version

As mentioned previously, the child version approach could make use of a control version. For this discussion, a control version is created from the parent version at the same time as the design version (or right after the reconciling of the design version with the parent version when doing incremental posts). The control version is never edited but instead used as a comparison version for difference queries and as a place to retrieve the original values of modified and deleted features that are being posted. If you are using ArcGIS 8.3 Desktop, then a control version can be used to limit the number of queries that must be performed with the IVersionedTable::Differences method during the posting procedure. Using ArcGIS 8.3, the differences method would need to be run only with the Insert, DeleteNoChange, and UpdateNoChange arguments if there is a control version. Without a control version, the DeleteUpdate, UpdateDelete, and UpdateUpdate difference types must also be checked. A new object (VersionDataChanges) was added at ArcGIS 9 to speed up the process of finding edits in a version with or without the control version.

Some have argued that to avoid conflict, you do not want to use a control version but instead want to check for differences against the version you are posting to and retrieve features from that version when undeleting and unmodifying. This is a valid argument and something to be considered when designing a solution. For example, look at the behavior you might expect when the same feature is edited in the design and in the version being posted to (for this example, that would be the default version).

Figure 8

In this new gas pipeline design, the features selected are those that are to be posted during the first pass.

The features in figure 8 represent the gas pipeline design. The pipe at the top of the display (annotated with "785") is an existing feature that was moved slightly to connect the rest of the pipes. The remaining pipes, services, junctions, and annotation are all part of the design. For the first phase of the design you only want to post the selected features (highlight in cyan). While you were working on your design, the pipe across the top was edited by someone else (a simple attribute change) and the update posted. Since you do not have the updated pipe selected (the pipe at the top of the display), it will not be posted as part of the incremental post, and it must be returned to its original state by retrieving a previous version of the feature. For this first run, you are checking for updates against the version you are posting to (the default version) and will pull the updated version of that feature down into your design. The results of the reconcile after running the incremental post application are shown in figure 9. Notice how the annotation has changed for the pipe at the top of the display from "785" to "783.5." This shows that the feature was updated by bringing the feature down from the default version.

J-9443

Figure 9

Using the default version as the comparison version, no conflicts are found during the reconcile. This is because the updated version was brought down from the default version.

In this case, using the default version worked well because you did not get any conflicts during the reconciliation process. Figure 10 shows the results of using the same feature updates but using a control version for comparison and for retrieving the values of the pipe you modified and left out of the post process.

J-9443

Figure 10

Using a control version allowed you to retrieve the original state of the pipe at the top of the display before editing. However, because this feature had been updated in the default version, you get conflicts during the reconcile process.

As figure 10 demonstrates, by using a control version to return the pipe back to the original state, you get conflicts during the reconcile process because that pipe was updated in the default version. This is important to be aware of. The incremental post application code (if using a control version) puts the pipe back to its original state, but the system sees this as an update when you attempt to do the post. This is seen as an update because you do actually update attribute values and the shape of the feature, even though you are setting the values back to their original settings. If this same feature is updated in

the version you are posting to (default in this case), then conflicts will be returned. In this example, the conflicts are escalated to the connected pipe because of network connectivity issues.

Based on the example of a pipe being modified in a design as well as in the default version (the version the design would be posted to), it appears that using a control version complicates the process because it creates conflicts that appear during the reconcile. By pulling features from the default version, you do not get any conflicts, but you are not aware that the feature was updated. That is the main trade-off between the control version and noncontrol version approaches.

One serious consideration to using the default version for comparison is determining how to deal with features that are deleted. In the scenario above, you would not be able to retrieve the original state of the pipe at the top of the display if it had been deleted in the default version. Without a control version, the only way to retrieve the original state of the feature would be to run your application code as part of the reconcile process and retrieve the feature from the common ancestor version. Future versions of ArcGIS may expose the common ancestor version outside the reconcile process, but for now, using the default version for comparison and feature retrieval is not feasible unless some additional method for retrieving features that have been deleted is discovered. The next section (Modeling Deleted Features) discusses a method to help restore features deleted in the design that are not to be posted. This would be no help in the scenario in which a feature is modified in the design and deleted in the default version. This scenario requires attribute values to be retrieved from the feature in the default version, which as mentioned, would not be possible if the feature no longer existed. The sample application code leaves the object in its current state if it cannot retrieve the original version from the version being posted to.

Modeling Deleted Features

How to handle delete features in your design is another topic that warrants some additional discussion. The sample code assumes deleted features are to be flagged with an attribute value and are not actually deleted. This approach is taken because crews in the field building the design would obviously find it beneficial for features that are being removed to be shown on the map so they know what to remove. These features would be shown with different symbology from standard features so they are easily distinguishable.

To use this approach it is necessary to have a field that designates the status of the feature (DELETED in this case). The BuildStatus field and domain discussed in the attribute values alternative could be used for this purpose. During the editing process, instead of deleting features you would set their attribute value to DELETED. It would also be necessary to make sure network features were no longer traceable. This could be accomplished by physically disconnecting the feature from the network, running the editor Disconnect command, or setting the Enabled property on the feature to False. Network weights could also be used, but you would not want to add a network weight for this purpose alone. Symbology for your features would need to use the BuildStatus attribute field so features with values of DELETED would be shown differently from other features.

During the posting process, features that are flagged to be deleted would need to actually be removed. This is a rather straightforward process using application code. Features

marked as DELETED that are not being posted need to have their attribute value set back to the original state. This process is easier than adding a feature back in (as you would have to do if you delete the feature in the design), which creates new object IDs and requires additional application code to ensure relationships are reset, and so forth. This approach also has the distinct advantage of allowing users to select the features they want to post (delete in this case). Deleting features in the design requires users to pick the deleted features to post using another method (such as a popup dialog box implemented in the sample tools for features that are split).

For incremental posting purposes, flagging features for deleting may be an easier process to implement than the removal process. The process assumes that plots will be made of the design before posting for use in the field by the crews doing the work.

Creating a Proposed View Version

Some organizations have the additional requirement of needing to see all design work in a single version. This would allow all proposed work to be viewable in a single display without having to switch back and forth between all the outstanding designs. Early proposals for incremental posting included the use of a proposed view version with posting being done directly from this version. If the child version approach outlined above were to be used with a proposed view version, then incremental posts would require the undoing of not only the updates in the current design that are not being posted but also all the other edits in each design. If you are an organization with thousands of outstanding designs, then it is not really feasible to undo all these edits while attempting to incrementally post one phase of one design.

Another approach considered was to have a proposed view version that sat between the default version and the designs. All designs would be created from the proposed view. When a design was completed it would be reconciled and posted with the proposed view so all designs could be found in one central place. After posting to the proposed view, the individual design versions would not be posted but instead would stay around until the design was built. At this point the individual design version would be reconciled and posted directly to default. The problem with this approach is that each time you create a design from the proposed view, you bring down all the other pending designs and end up with the same issues discussed in the paragraph above.

The only way to properly maintain a proposed view is to somehow separate it from the branch that the design versions reside on. Figure 11 illustrates this structure.

J-9443

Figure 11

The proposed view version resides as a child of the parent (default) version but through reconciliation is kept at the same state. With the two versions at the same state, a copy of the design to be posted can be made and its parent reassigned to the proposed view. This would then allow the design to be reconciled and posted with the proposed view version.

The flow described in figure 11 will be possible with ArcGIS 9.2. To accomplish this process with the current version of ArcGIS you need to either create a database trigger to update the state that versions are pointing to (not recommended) or use application code along the lines of the parallel version approach to copy the design features to another version underneath the proposed view.

Conclusion

If you do not make modifications and deletions in your design and do not have related objects based on the object ID, then it should not be too difficult to perform incremental posts from your design version. However, just about everybody needs to delete and modify and has relationships based on the object ID. Even with these requirements, though, there are ways to make incremental posts work.

If your database schema is not already defined, avoid using object IDs as the origin or destination key in a relationship. Reestablishing relationships after retrieving a deleted or modified feature can be done, but it only has to be done when object IDs are involved, as they are changed when the object is added back into the design. If user-controlled fields are used, then they are not recalculated when the object is added, and therefore, relationships do not need to be reestablished.

Everyone would like a partial posting solution to be part of the core software, but work on the alternatives for this paper has shown this is not practical. Based on the differences in database schema, design practices, and so forth, among the different users, a clear

picture of what to actually add to the core is not apparent. If you consider the uncertainty of unmodifying and undeleting objects as required on network connectivity and relationships, application code emerges as the best solution to the problem.

There are four potential changes to the core software that might make possible solutions to incremental posting more palatable. The first is the requirement to add a network weight if an attribute value is used to designate the status of a feature. It would be advantageous to be able to add a network weight without having to re-create the network so users can maintain their currently open versions, and so forth.

The second change would be the ability to access the common ancestor version outside the reconcile process. The common ancestor version is the state of the database that both the design and the version being posted to (most likely the default version) are derived from. As a result, this state of the database contains the original versions of all the features. This common ancestor state could be used to retrieve features when you need to reset values in the design version during incremental posting and the feature has been deleted in the version you are posting to.

The third change would be an adjustment to how splits are seen by the database. As mentioned previously, a split with the out-of-the-box tools is seen as a delete and two adds by the database. As it currently stands, it is not possible to distinguish a split from a standard delete when looking at the difference queries at reconcile time. Changing splits to a modification and an add would make it easier and less problematic when writing application code for incremental posting.

The fourth change would be the ability to undelete or unmodify a feature in a more graceful manner. As mentioned in the paper, the ability to retrieve a deleted row is currently not available because there is no way to predict the impact of undeleting a feature on the logical network, among other things. This predicament is understandable, but in the future other methods may be discovered for making this process less error prone.

All these modifications are under consideration by the development team. The first two are already part of the specification for an upcoming release.

The first two alternatives presented (the attribute approach and the separate object class approach) are the easiest to implement and do not require designs to remain in open versions until they are built (which would then allow the constructed features to be posted to the default version). However, they both have limited tracing capabilities. The requirement to trace from the design features into the as-built features would make both of these alternatives difficult to implement. Both also have issues when it comes to identifying the features in a particular design (though this could be accomplished by attribute values) and getting an accurate view of a design when there are multiple alternatives to it (the features all end up stacked on top of each other).

The parallel and child version approaches offer better solutions for visualizing the individual designs and for tracing. However, they do require the designs to remain in outstanding versions until they are built, and they do not currently offer a method for

J-9443

viewing all proposed design features in one version. In addition, both of these alternatives require advanced application code.

Be sure to fully understand your requirements before choosing an alternative to implement.