Modeling and Using History in ArcGIS™

An ESRI® Technical Paper • May 2002

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

Modeling and Using History in ArcGIS

An ESRI Technical Paper

Contents

Modeling and Using History in ArcGIS

Introduction

Many organizations wish to manage how their geographical database changes over time. This process is commonly referred to as history. A historical database records information about database entities (objects, features, and relationships) through time. The historical database can be queried at a point in time and get the correct set of entities that existed in the database at that time.

Modeling history in a database allows for analysis, such as visualizing the database's contents at a particular time in the past or visualizing how a particular object changed over time. This technical paper, which applies to ArcGIS™ 8.2 on all supported platforms, will review some historical concepts and outline how history is modeled and managed in a geodatabase. For developers who want to build applications for querying and analyzing a historical geodatabase, a sample application for navigating history is also presented.

What is database history?

History has been traditionally achieved through a special archiving strategy or custom applications that store deleted features in a special history layer or maintain fields on tables with active dates for each row. The geodatabase's versioning model provides an alternative to these methods. A version can be used to represent the state of a database at a specific point in time. The versioning model will maintain the old representation of objects, deleted objects, and the time that these events happened without having to manage special layers or date stamps on features.

Before detailing how the versioned geodatabase model can be used to manage history, it serves to discuss some general database history management concepts.

Change events

A change event in a historical database is the thing that moves a database from one point in time or state to another point in time or state. In the example of a database of county population from 1790 to 2000, the database might contain a record for each county every 10 years, based on the census. The change event in this case is the census. In the example of a parcel ownership database from 1990 to 2001 and beyond, the change events are the instruments (a deed, subdivision plat, etc.) that record the change in ownership.

Change events are important in interpreting and assigning meaning to records in a historical database in several ways. First, they associate meaning with the temporal granularity of the database. Second, they support queries on how the database changes over time.

Temporal granularity

The temporal granularity is the frequency between change events. Historical records can be recorded at the finest granularity, based on individual database transactions, or records can be aggregated or summarized to broader time periods or snapshots.

Transactions

The finest granularity in which history can be recorded is at the transaction level. A change event is created for each database transaction. The transaction could be at the completion of a project, a work order, or some type of document that effects a change in the database. In a parcel database, a historical version could be created every time a parcel or group of parcels are created, modified, or deleted. In this case there is typically some type of instrument, such as a deed or subdivision plat, that causes the change. This instrument is the change event.

Transaction level history in a geographic database is meaningful in terms of the contents of a long transaction. Typical geographic updates require many database transactions to be completed. For example, the splitting of a parcel is a single long transaction that may require the following collection of short transactions:

1. Split original parcel into two parcels.
2. Create a new parcel boundary along this split line.
3. Assign parcel numbers to the new parcels.
4. Create annotation features for each new parcel.
5. Create dimension features for each new parcel boundary.

While this operation was five (or more) short transactions in the database, it is logically a single long transaction. This can also be referred to as a unit of digital update. If a history is captured in the database at the transaction level, logically the long transaction or unit of digital update is the level to which this history is captured.

Snapshots

A database snapshot is created at a regular interval as a snapshot of the database state at that point in time. The snapshot is used as the public or official view of the database until the next snapshot is made. Over time these snapshots can be used to analyze changes in assessment patterns as well as to document the conditions or state of the data at that point in time. This type of historical information does not track intermediate changes that occur between the snapshots.

Local governments often do this once or several times a year to match their tax or assessment roll. Once all of the edits are made to the parcel data for the current snapshot period, the snapshot is created. This typically involves copying the parcel and other data to another table in the database or to a file format, such as a shapefile. There is generally a significant time lag between when a parcel is split or a subdivision is recorded and the recording of the event in the geographic information system (GIS). This typically results in the snapshot not being made until after the published date of the snapshot. The snapshot becomes the official assessed parcels view of the database that is used by all GIS users outside the assessment office. Once the snapshot is created, the assessment office editors will commence the editing for the next snapshot period. The intermediate changes made to parcels will be lost. For example, a property could change hands several times during a year; however, only the starting and ending owner would be captured in the archive.

Another example involves modeling population through time. A historical database may contain the population data for states from 1790 through 2000 and beyond. The temporal granularity of the version would be 10 years or each time the census is undertaken.

Real time versus transaction time

There are different types of time a historical database can maintain. Real time is the time an event actually occurred. Real times are usually user supplied. Transaction time is the time an event is entered into the database. Transaction times are usually system-generated times.

The real time and the transaction time for an event may be different. There is generally a time lag between when an event occurs and when it is entered into the database. Organizations have backlogs in entering data for any number of reasons: staff issues, workflow bottlenecks, etc. For example, in northern weather-affected areas, organizations will save data updates to be entered during the winter when field-workers cannot work outside.

The differences between real time and transaction time become apparent when historical queries are performed. Asking what the database looked like on a particular date is different than what the configuration of entities was on that same date. The effective start date of a hydrant may be April 23, but it wasn't entered into the GIS until June 2. Depending on what history is being used in the GIS, either type of date could be used.

When modeling history in a multiuser environment with many different people or departments contributing to the database, the differences between real and transaction time are also an issue. The order in which events are entered into the database (transaction time) may not be the same order in which they occurred in the real world (real time).

Historical queries

The typical requests that are made of historical databases are as follows:

- Show me the database at time "A".
  Depending on the temporal granularity of the database, the exact time may not be possible.

- Show me how feature "Y" has changed through time.
  This is generally referred to as a lineage search. The search is started by either selecting a feature or entering a known identifier. From there, a dialog box is used to walk forward or backward through the life cycle of the feature. The temporal granularity of the database will determine what information can be extracted.

- Show me what is in the space of feature "Z" at time "B".
  This is a similar query to the lineage query but is used to determine what the current configuration of features is where the queried feature previously existed. For example, a person at the front counter of the assessment office may want details about their parcel, but they have a PIN number for a deleted parcel.

Transactional updates

Data providers are often required to provide updates of their database to their clients or other agencies on a periodic or on-demand basis. Each delivery consists of all of the changes from the previous delivery. If each data delivery is kept as a historical version, it is a straightforward exercise to perform queries to determine which records have changed between the current state of the database and the state of the database at the last data delivery.

This could potentially be delivered as a replication update; however, clients may also receive this as a transaction file where the adds, updates, and deletes are indicated. It then becomes the responsibility of the client to integrate these changes into their database.

The history data model and sample application

The versioned database data model in the ArcSDE™ geodatabase can be used to navigate history on both the database level and the individual feature level. The versioned database contains all the information necessary to visualize the state of the database at any given point in time if a reference (i.e., a version) exists at that particular point in time. One of the important points of how a versioned database maintains edits is that objects (features and rows) are never deleted. All changes to a versioned database (inserts, updates, and deletes) are inserts into delta tables. It is this aspect of the versioned database that allows maintenance and querying of the history of the database.

The remainder of this technical paper will focus on the versioning model in the geodatabase and how it is used to model history. For application developers, a sample application for navigating history in the geodatabase is reviewed and is available for download on ArcOnline (http://arconline.esri.com/arcobjectsonline). To begin the technical discussion of how history is modeled, it serves to define some terms and describe how versioning works.

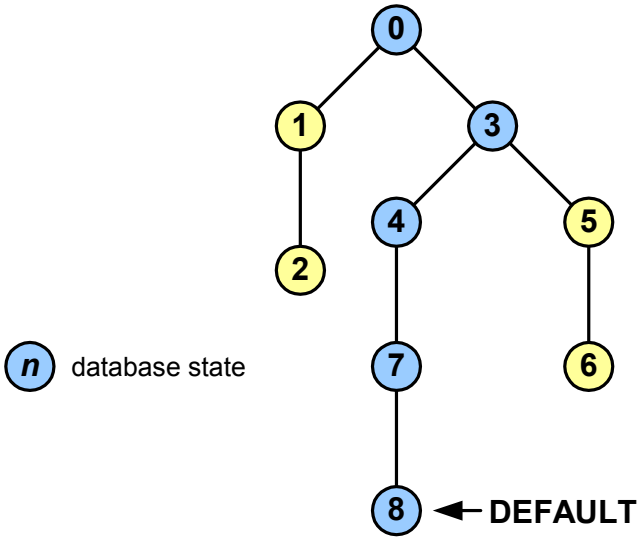
Versioning in the geodatabase

A versioned database is a database that can have multiple persistent representations of its contents without the need for data replication. Versioning allows an organization to manage alternative engineering designs and solve complex “what if” scenarios without impacting the corporate database and to create point-in-time representations of the database. The creation of point-in-time representations of the database is what is of most interest to modeling history.

A versioned database will contain a number of versions. A version is an alternative representation of the database that has an owner, description, level of access, and parent version. Versions are not affected by changes occurring in other versions of the database. Typically, a version will represent a work order, a design alternative, or a historical point in time of the database. All versioned geodatabases contain a DEFAULT version. The DEFAULT version is the as-built representation of the database and is the root of the version tree—that is, it does not have a parent, and it is the ultimate ancestor of all other versions in the database.

To manage versions, a versioned geodatabase contains a collection of states. States are a discrete snapshot of the database. They have the same schema and differ only in the set of rows for each table or feature class in the database and are organized in a tree structure. As you edit a particular version of the database, it moves through a succession of states. At any point in time, a version will reference a particular state of the database. To return

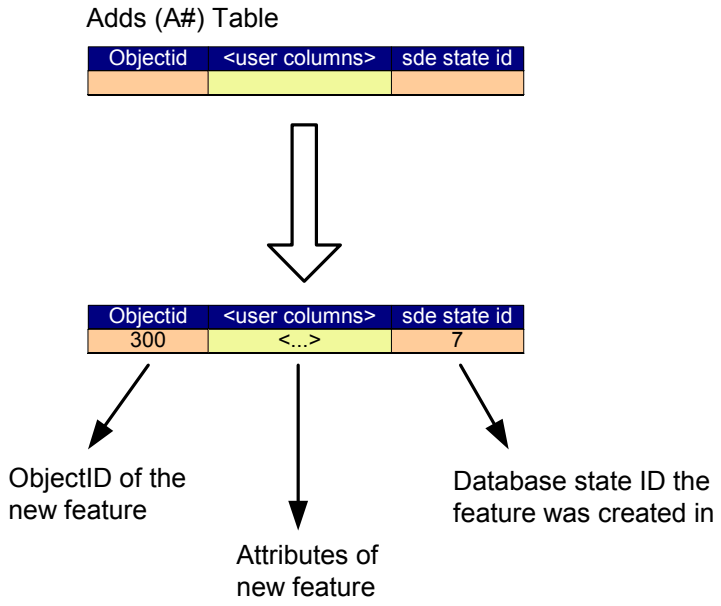
the contents of any particular version of the database, its states lineage is queried to determine the collection of rows to return for that version of the database.



The DEFAULT version's lineage is states 0, 3, 4, 7, and 8. These states are queried to determine the correct set of rows for the version.

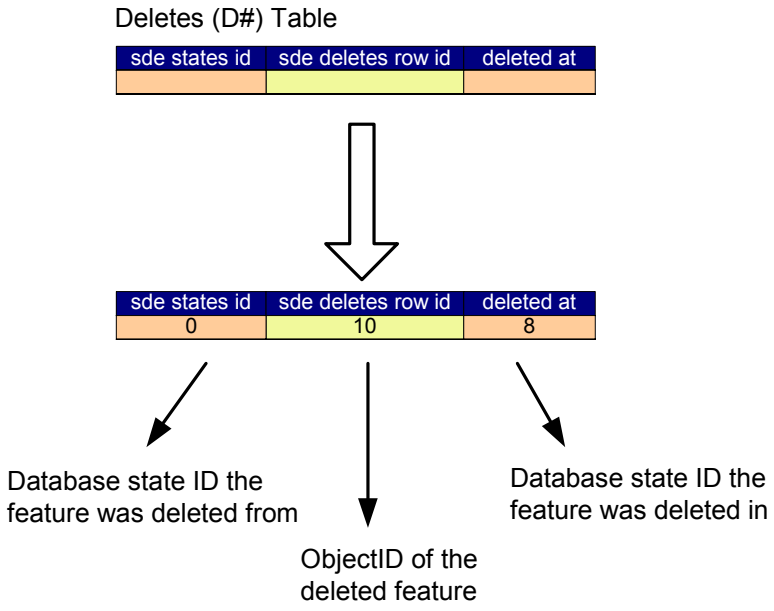
Each feature class or table that is registered as versioned in a geodatabase has a set of delta tables. There are two delta tables for each feature class or table: the Adds table and the Deletes table. These tables record what objects are created, updated, and deleted in the various states of the database.

For example, when you create a new feature in a feature class, a new row is added to the Adds table recording in what state the feature was added to the database:



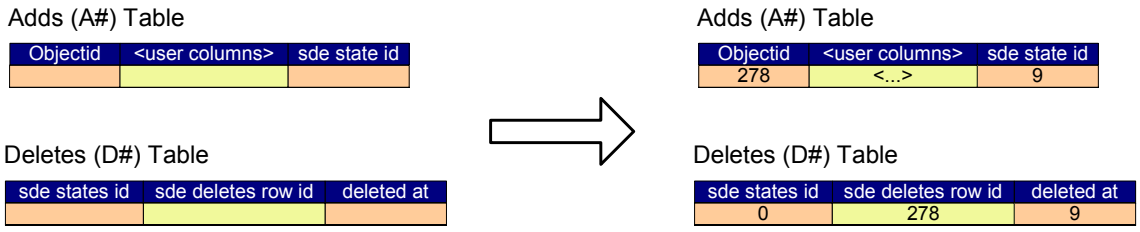
When a new feature or object is created, a row is added to the class's Adds (A#) table.

When you delete a feature, a row is added to the Deletes table to record what state the feature was deleted in and what state it was deleted from:



When a feature or object is deleted, a row is added to the class's Deletes (D#) table.

When you update a feature, a row is added to both the Adds and Deletes tables:



When a feature or object is updated, a row is added both to the class's Adds (A#) table and deletes (D#) table.

When you edit a geodatabase, a new state is created for each edit operation. At any point in time, the state tree will contain a number of states that are not referenced by versions. Periodically, a versioned database needs to be compressed. Compressing a versioned database simplifies the state tree and moves rows from the delta tables to the base tables of versioned feature classes and tables. Each state that is referenced by a version can still be queried for its set of rows after a compress.

Like other versions, the DEFAULT version also references a series of database states through time. The DEFAULT version will change states when it is edited directly or when other versions are posted to the DEFAULT version. To maintain the history of a database, those states that represent significant points in the life of the database need to be referenced by versions such that they are not removed during a compress and their contents can be retrieved at some later point in time.

By creating versions from DEFAULT at certain time intervals (daily, weekly, monthly, etc.) or after certain significant events (such as a significant work order or document posted to DEFAULT), these versions will continue to reference the state of the database that DEFAULT referenced when they were created. Later, after DEFAULT has continued to change, these snapshot or historical versions can be queried to determine the objects that existed in DEFAULT at the time that version was created.

As discussed above, all edits to a versioned database are inserts into delta tables. When an update is made to a particular object, the representation of the object before it was edited remains in the database. The state tree can be analyzed to determine when a particular object was created, when and how it was updated during its lifetime, and when it was deleted. The granularity of this information is determined by which states of the database are referenced by versions.

Comparing other history models to versioning

A common strategy for maintaining history in a database is to track the effective date of an object by making all updates inserts of new rows for that object in the table. Each row has included a start and end date to indicate at what point in time a particular representation of the object applies. Structured Query Language (SQL) queries can be used to return the set of rows that apply to any point in time by using ranges against the start and end date fields in the table. This is illustrated below:

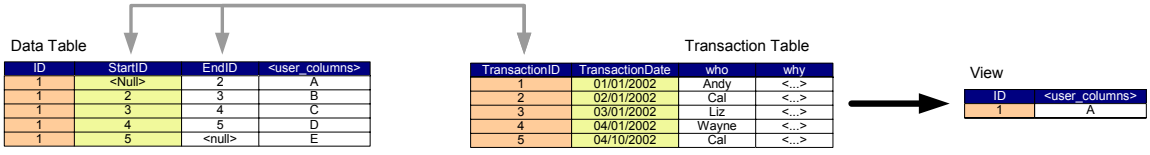
Data Table

ID	StartDate	EndDate	<user_columns>
1	<Null>	01/01/2002	A
1	01/01/2002	02/01/2002	B
1	02/01/2002	03/01/2002	C
1	03/01/2002	04/01/2002	D
1	04/01/2002	<Null>	E

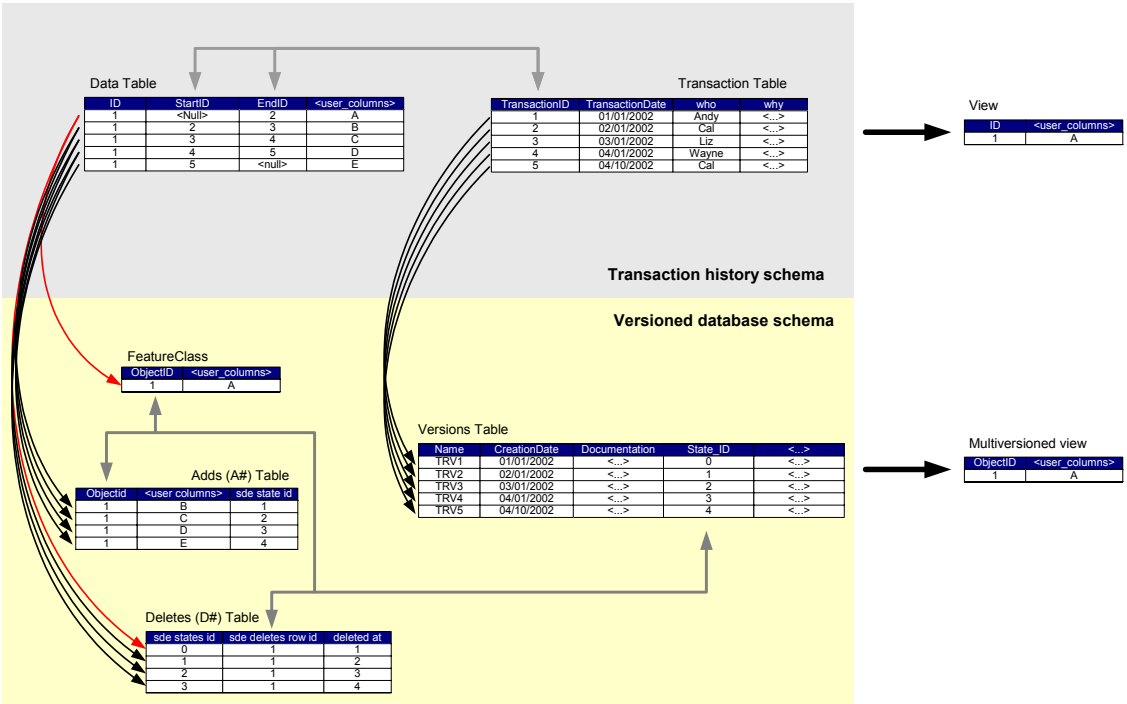
Query

ID	<user_columns>
1	A

Due to the long transactional nature of GIS databases, a more applicable strategy is to maintain a table of long transactions. Each representation of an object is still a new row in the table, but that table has a start and end long transaction ID associated with it. The transactions themselves are date stamped, and a view can be created between the table and the transaction table to produce a set of rows that are relevant to any point in time. This is illustrated below:



To contrast the versioning model with the transaction table strategy, each row in the transaction table is a version in the geodatabase (and therefore a row in the versions table). The original representation of an object is in the base table for the object class, and each additional representation of the object is organized into rows in the Adds and Deletes tables. These Adds and Deletes rows correspond to database states to which the versions are references. To view the contents of any version at any point in time, a multiversioned view can be used to access the records through SQL, or ArcMap™ or other applications written with ArcObjects™ can be used to work with the data in any particular version of the database. The comparison of these models is illustrated below:



Historical queries

There are two key parts to the sample applications included with this technical paper: navigating the history of the database as a whole and navigating the history of (or lineage of) a particular object. How those pieces of the application work is detailed here.

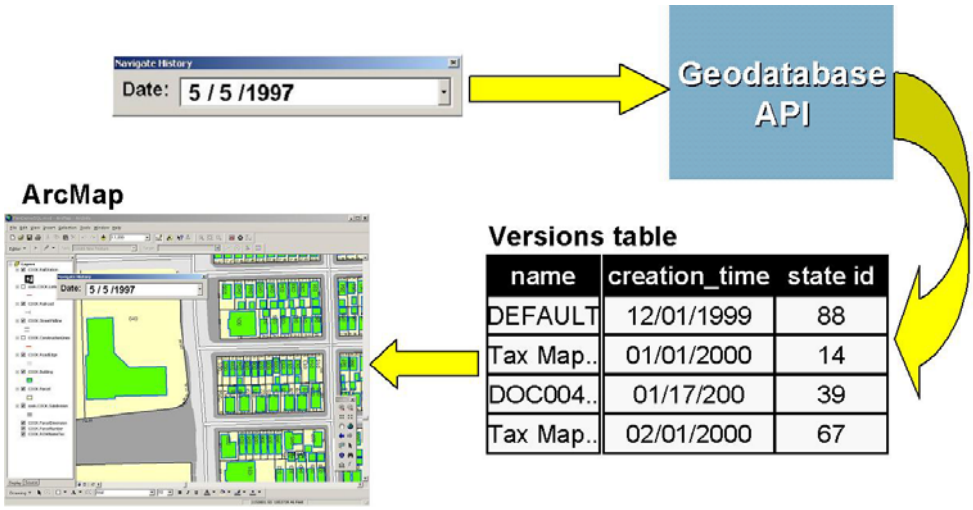
In the discussion above, three key queries were identified for navigating the history of a database. Each query will be discussed on the level of how the information is retrieved from the database, and some implementation details for application developers will also be presented. Knowledge of programming or ArcObjects is not required to understand how these queries work. To review, the three key queries are:

- Show me the database at time "A".
- Show me how feature "Y" has changed through time.
- Show me what is in the space of feature "Z" at time "B".

Show me the database at time "A"

This type of query or capability allows us to work with the representation of the database at some point in time (in the past). The versioned geodatabase can be queried for its contents at any point in time for which a historical version exists. Each version has a name, a parent version, and a creation date. Using the creation date of a historical version, the geodatabase can be queried to display the contents of a historical version whose creation date is the closest to the requested date—without going over. The results of any query against that version of the database will reflect the contents of the geodatabase at that point in time. Once you have a reference to that version, you can work with it to perform any kind of analysis you require.

This functionality can be realized through an application created with ArcObjects that can search for the correct version based on date and connect the application to that version of the database. The sample history application shows an example of a user interface in ArcMap to do these kinds of queries through the History navigator tool. This tool is an ArcMap command and consists of a date control in which the user can type a desired date. The application uses this date to examine the versions of the database to which ArcMap is connected and determines which version has its creation date closest to the requested date without going over. Once that version is found, the equivalent of the ArcMap change version command is executed such that ArcMap points at that version of the database:

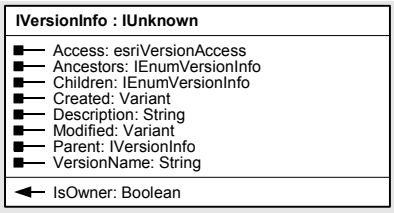


Once ArcMap is connected to this historical version, any queries or analysis done with ArcMap will be done against the contents of the database as they existed at that point in time.

For developers:

There are a few key ArcObjects classes and interfaces used by the History navigator to find the appropriate version and change ArcMap to point at that version of the database. These are `VersionedWorkspace`, `IVersionedWorkspace`, `VersionInfo`, and `IVersionInfo`.

IVersionedWorkspace : IUnknown
DefaultVersion: IVersion Versions: IEnumVersionInfo
Compress FindVersion (in Name: String) : IVersion



When the command is clicked, the History navigator dialog box is loaded. When the dialog box loads, it queries the workspace for its versions using `IVersionedWorkspace::Versions`, which returns an enumeration of `VersionInfo` objects. Each `VersionInfo` object is cached in the dialog box and sorted based on `IVersionInfo::Created`.

Versions do not have metadata associated with them, so in order to determine which versions are historical versions and which are not, a special version naming convention is used. In this example, all historical versions start with the name “Tax Map”. Versions that do not start with that name are considered to be nonhistorical and are not searched during historical navigation.

In addition to this, since the time that any version represents in the database is based on the creation date of the version, there is special logic to handle the DEFAULT version. The DEFAULT version always has the earliest creation date but represents the most current point in time of the database. So, when caching the DEFAULT version, its date is cached as the current date.

An excerpt of the code from a helper function called by the Form_Load sub is included below:

```
‘ Get a list of all versions.
Dim pSdeWks As IVersionedWorkspace
Dim pDefaultVersion As iVersion
Dim pCurrentVersion As iVersion
Dim pVersionInfo As IVersionInfo
Dim pVersion_list As IEnumVersionInfo
Dim sName as String

Set pVersion_list = pSdeWks.Versions
Set pVersionInfo = pVersion_list.Next
Do While Not pVersionInfo Is Nothing
If InStr(1, pVersionInfo.VersionName, "Tax Map", vbTextCompare) >
0 or (Name, Len(Name) - InStr(Name, ".")) = "DEFAULT" Then
 If Right(Name, Len(Name) - InStr(Name, ".")) = "DEFAULT" Then
  aVDate(j) = Time
 Else
  aVDate(j) = CLng(Format(pVersionInfo.Created, "yyyymmdd")
 endif
 sName = pVersionInfo.VersionName
 ‘ Add the versions to an array sorted by date.
 Set pVersionInfo = pVersion_list.Next
Loop
```

When a new date is entered into the date control, the cached version information is searched for the version with the closest creation date, and once found, the version's name is used to call `IVersionedWorkspace::FindVersion` and change ArcMap to point at the new version.

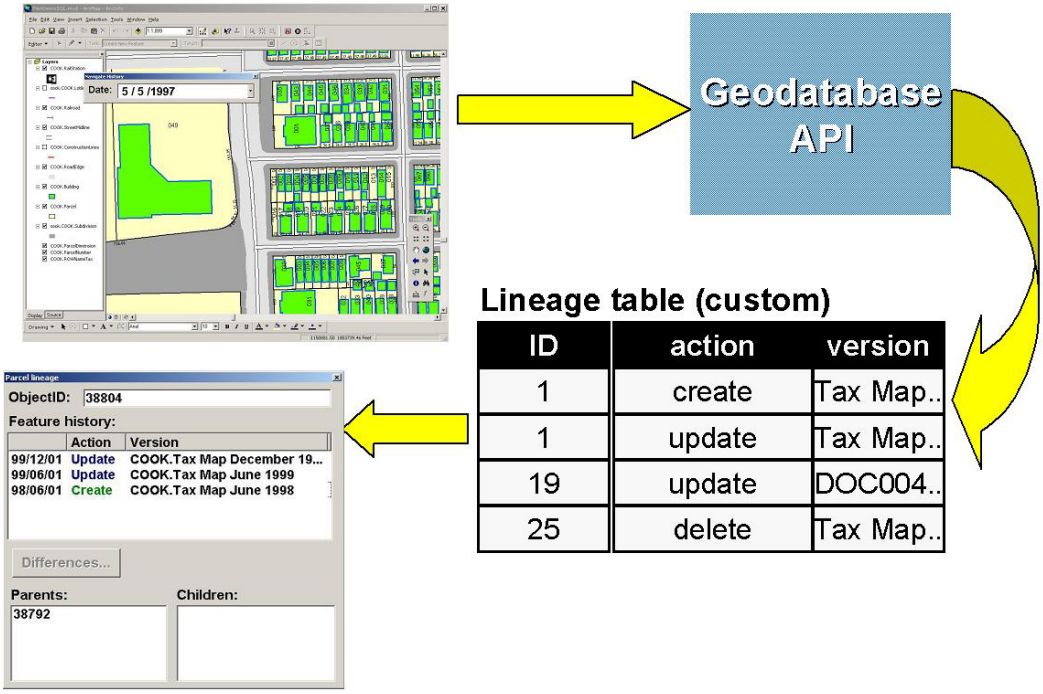
Show me how feature "Y" has changed through time

The previous example deals with querying the geodatabase as a whole at some point in time. The second key historical query is to look at how an individual feature or object in the database has changed over time. How an object changed over time is often referred to as that feature's lineage; the second part of a feature's lineage deals with parent and child features, and this will be discussed later.

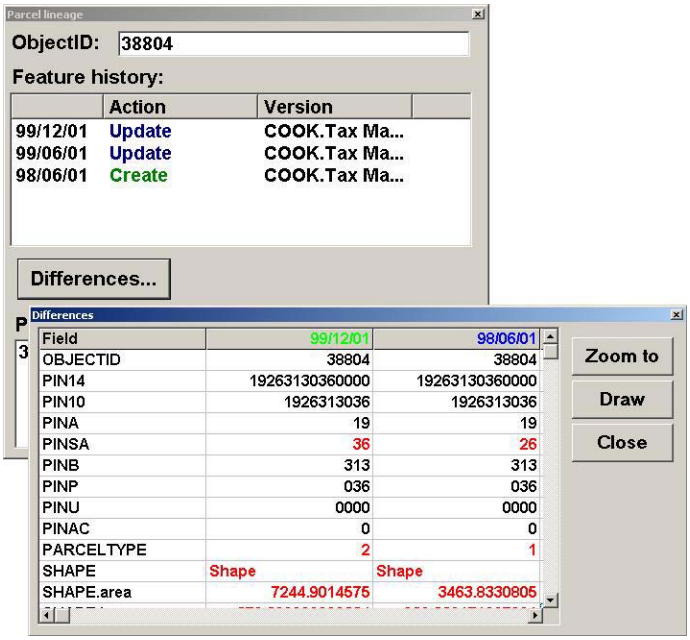
As discussed previously, updates and deletes to objects (rows) in a versioned database are inserts into delta tables (the Adds and Deletes tables). So the representation of an object is never deleted from a versioned database while that representation is referenced by a version. This means that an object's representation at any point in time and information as to what historical versions of any particular object in the database are available are through the versioned geodatabase.

Once you determine which historical versions of a particular object were created, updated, or deleted, you can look at the representation or state of an object at a particular point in time. The sample ArcObjects application includes a special identify command that displays lineage information for a feature in the map:

The combo box contains a list of all the layers in the map for which there exists a lineage index (see below). When the tool is activated and the map clicked, a spatial search is executed to find any features from the feature class selected in the combo box that were clicked. Once found, the class's lineage table is queried to populate the lineage dialog box with the list of versions and what occurred in that version (i.e., was the feature created/updated/deleted).



The lineage dialog box contains a list of all the historical versions in which this object was affected and the creation dates of those versions. It displays whether the object was updated, inserted, or deleted. Since the information about when a feature was updated is at hand (i.e., the version names), the differences can be displayed:



In this case, each version is opened, and a reference to the object in each version is obtained and compared to determine which attributes have changed between the two representations and is used to fill in the dialog box above.

In order to determine what versions of the database a particular object was created, updated, or deleted in, special queries called difference queries are used to discover the set of rows that changed for a particular table (object class or feature class) between two versions.

To determine all of the versions that a particular object was affected in would require that difference queries be run against each version pair in the set of historical versions. This number of queries in the database would be expensive. The sample ArcObjects application simplifies this by using a lineage index. The lineage index is a special table for each feature class or table that you want to examine lineage for its objects that contains references to those versions that the object was affected in. Each class for which you track lineage has its own lineage table. The naming convention for this table is <class_name>LINEAGE. For example, the PARCELS feature class's lineage table is PARCELSLINEAGE. The lineage table is a nonversioned table.

The schema of this table is as follows:

OBJECTID—an ObjectID field that represents the object ID of the lineage table
FEATUREID—a long integer field that represents the object ID of the feature whose lineage is recorded
UPDATED—a date field that represents the creation date of the version referenced by the record
ACTION_—a long integer field that represents what happened to that feature in that version (0 = insert, 1 = update, 2 = delete)
VERSION—a string field that represents the name of the version

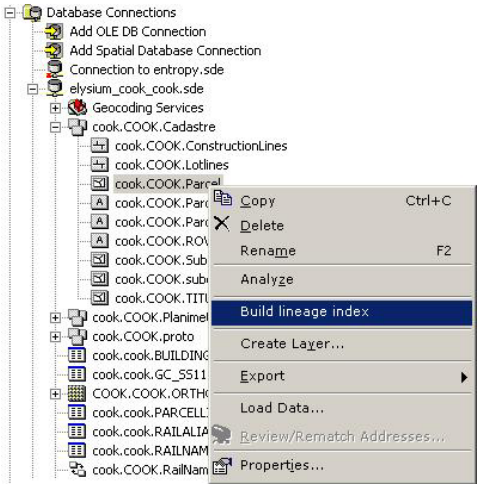
For example, a feature class that has a single feature that was created in Version1, updated in Version2, and deleted in Version3 would have a lineage table that looks like this:

OBJECTID	FEATUREID	UPDATED	ACTION_	VERSION
1	1	01/01/2001	0	Version1
2	1	02/01/2001	1	Version2
3	1	03/01/2001	2	Version3

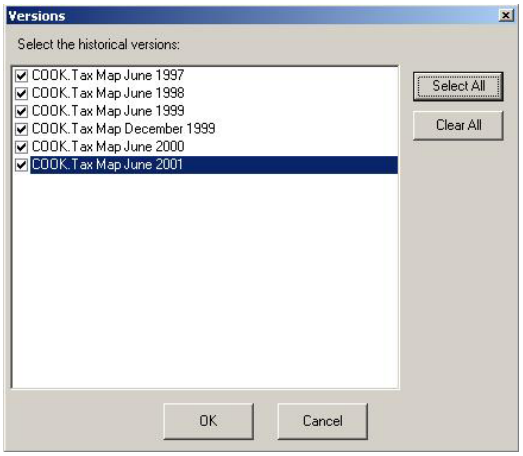
Since the version is referenced in the lineage table by name, and no references are made to database states, the compress command will not affect the correctness of the lineage index if the state tree is reshuffled.

To create this index, Build lineage index, an ArcCatalog™ command, is included as part of the sample application. This command is run for each feature class for which you want to generate this lineage table. The command examines the historical versions in a database and populates the lineage table all at once. In the example below, the command

has been placed on the feature class context menu. To execute it, right-click the feature class for which you want to create a lineage index and click the command:



This command will attempt to get an exclusive lock on the feature class. If it succeeds, then it displays the following dialog box so the user can specify which versions in the database represent historical versions.



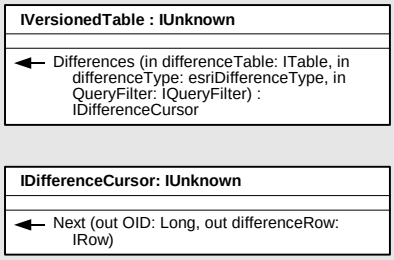
Once the user clicks OK, if the lineage table does not yet exist, then it is created. Within a database transaction the following is performed:

- The lineage table is truncated.
- For each historical version pair:
    - Perform an Insert, Update no delete, and Delete no update difference cursor, which gives us the objects that were inserted, updated, and deleted between the versions, respectively.

- Using the results of these difference cursors, populate the lineage table with records (one record will be created for each row returned from each difference query).

For developers:

The key object used to determine the information to insert into the lineage index table is the `DifferenceCursor`. The `DifferenceCursor` object returns the set of rows that is different between two versions in the database for a particular table or feature class. Examples of the types of differences are inserted in one version, updated in one version, deleted in one version, etc. For the purposes of building the lineage index, focus on the following: `esriDifferenceTypeInsert`, `esriDifferenceTypeUpdateNoChange`, `esriDifferenceTypeDeleteNoChange`, which translate into the set of rows that were updated, inserted, and deleted between two historical versions.



Once the set of rows has been returned from the difference cursor between any two historical versions, that information can be inserted into the lineage index table.

The following function executes the difference cursor on a table for the specified difference type:

```
Private Function getDiff(sTabName As String, sV1 As String, sV2 As
String, enumDiffType As esriDifferenceType) As IDifferenceCursor
Dim pWS1 As IWorkspace
Dim pWS2 As IWorkspace
Dim pProps1 As IPropertySet
Set pProps1 = m_pWS.ConnectionProperties
Dim pProps2 As IPropertySet
Set pProps2 = m_pWS.ConnectionProperties

Dim pWSF As IWorkspaceFactory
Set pWSF = New SdeWorkspaceFactory

‘ where sV1 and sV2 are the names of the two historical versions
‘ being compared
```

```
pProps1.SetProperty "version", sV1
pProps2.SetProperty "version", sV2

Set pWS1 = pWSF.Open(pProps1, 0)
Set pWS2 = pWSF.Open(pProps2, 0)

Dim pTable1 As ITable
Dim pTable2 As ITable
Dim pVTable As IVersionedTable

Dim pFWS1 As IFeatureWorkspace
Dim pFWS2 As IFeatureWorkspace
Set pFWS1 = pWS1
Set pFWS2 = pWS2

‘ where sTabName is the name of the table or feature class for
‘ which a lineage index table is being created or updated
Set pTable1 = pFWS1.OpenTable(sTabName)
Set pTable2 = pFWS2.OpenTable(sTabName)

‘ where enumDiffType is the particular difference type being
‘ queried: esriDifferenceTypeInsert,
‘ esriDifferenceTypeUpdateNoChange,
‘ esriDifferenceTypeDeleteNoChange
Set pVTable = pTable2
Set getDiff = pVTable.Differences(pTable1, enumDiffType, Nothing)

End Function
```

The results of this function are then passed into the following routine to populate the table based on the results of the difference cursor:

```
Private Sub popTable(pLinTab As ITable, pDiffCursor As
IDifferenceCursor, lAction As Long, sVersion As String)
Dim pVInfo As IVersionInfo
Dim pVWS As IVersionedWorkspace
Set pVWS = m_pWS
Dim pVersion As IVersion
Set pVersion = pVWS.FindVersion(sVersion)
Set pVInfo = pVersion.VersionInfo

‘ Open an insert cursor on the lineage index table.
Dim pInsCursor As ICursor
Set pInsCursor = pLinTab.Insert(True)
Dim pRowBuff As IRowBuffer
Set pRowBuff = pLinTab.CreateRowBuffer
Dim pFlds As IFields
Set pFlds = pRowBuff.Fields

‘ Get the first row and OID from the difference cursor.
Dim pRow As IRow
Dim lOid As Long
pDiffCursor.Next lOid, pRow

‘ For each row in the difference cursor, insert a row into the
‘ lineage index table.
```

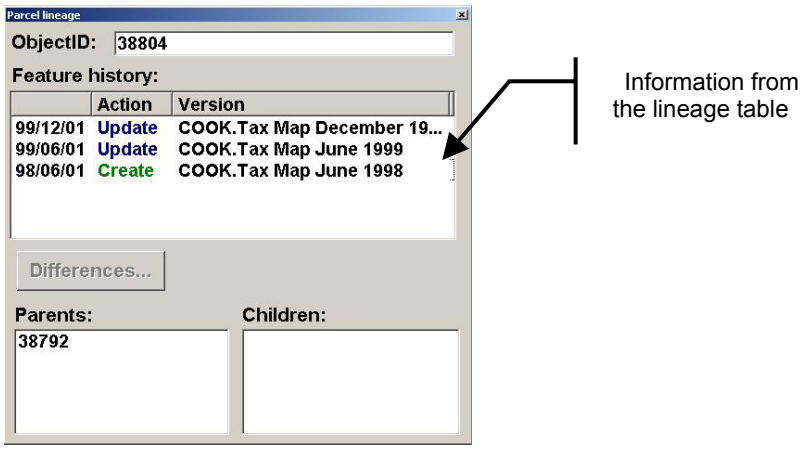
```
Do Until lOid = -1
  pRowBuff.Value(pFlds.FindField("FEATUREID")) = lOid
  pRowBuff.Value(pFlds.FindField("UPDATED")) = pVInfo.Created
  pRowBuff.Value(pFlds.FindField("ACTION_")) = lAction
  pRowBuff.Value(pFlds.FindField("VERSION")) = pVInfo.VersionName
  pInsCursor.InsertRow pRowBuff
  pDiffCursor.Next lOid, pRow
Loop

pInsCursor.Flush
End Sub
```

Once created, the lineage index table is static. It is not incrementally updated as new historical versions are created in the database. The Build lineage index command would need to be run periodically to re-create the lineage index to incorporate new historical versions.

When the object ID of the feature being examined is determined, the lineage table is used to directly populate the lineage dialog box. In this application, the special identify history command discussed above is used to find the ID of the feature in question.

Typically, a database will contain a number of features that were not created in any particular historical version. The creation date for these features is displayed as the creation date of the earliest historical version.



Show me what is in the space of feature "Z" at time "B"

This dialog box also contains information about the parents and children of the specified feature. The parents are the features that occupied the same space in the database before the feature was created. The children are the features that occupy the same space after the feature is deleted from the database. Those features that have not yet been deleted do not have children, as is the case in the example above.

To determine the parents and children, spatial searches across versions are used. In this example, the feature was created in the Tax Map June 1998 version. The geometry of the feature in that version is used to do a spatial query against the parcels feature class in the next oldest historical version, and the object IDs of the parcel features returned from that

spatial query (38792 in this case) are displayed in the dialog box. Similarly, if the feature had been deleted, then its last geometry would be used to query against the version from which it was deleted to determine its children.

For developers:

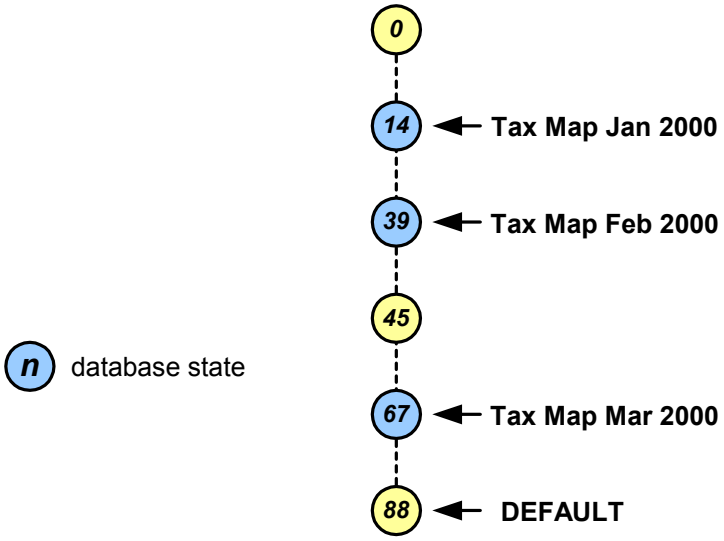
The spatial filter operator used in this case is the interior relationship:

```
pSpatialFilter.SpatialRel = esriSpatialRelRelation
pSpatialFilter.SpatialRelDescription = "T********"
```

Different spatial filters can be employed depending on the different types of spatial relationships across historical versions you want to examine or display.

Strategies for capturing history in a geodatabase

Historical versions are always part of the DEFAULT version's lineage. They are created as children of the DEFAULT version based on some kind of event (regular snapshot, significant work order, document, or transaction). In that sense, historical versions track the history of the DEFAULT version of the database. Once a historical version is created, it will represent the state of the DEFAULT version of the database at that point in time if it is never edited. Historical versions should never be modified in any way (being edited, being reconciled against other versions, or having other versions posted to them).



Historical versions are always part of the DEFAULT version's lineage.

Some important decisions need to be made when determining how and when historical versions are created in the database. These decisions mainly surround the granularity for which history is captured and the lifetime of historical versions.

When considering the granularity of your historical versions, an important point is that less is better. This means the less historical versions there are in a database, the less storage required for the database and, ultimately, the simpler, more efficient, and faster the queries to retrieve the information are. It is important to choose a granularity that you really need, meaning there is no point to capturing a historical version on a daily basis if you only query history on an end-of-the-month basis (in that case you should capture a historical version on a monthly basis).

For those organizations that do require very fine-grained history, an important factor is for what period of time is that fine-grained history required. For example, a tax assessor's office must be able to view the history of the database for any particular day of the current tax year. However, for previous tax years, the assessor needs only to know the tax map for the end of the tax year. In this scenario, daily historical versions can be captured throughout the tax year, and then at the end of the tax year, all the historical versions for that year (except the historical version created on the last day of the year) can be removed because they will never be queried again.

An organization that requires history on the transaction level should be sure they understand the implications of capturing that number of versions and consider capturing historical versions for logical transaction blocks. For example, when adding a subdivision, a number of transactions may take place in the database, but the only significant one as far as capturing history for is the one that represents the completion of the new subdivision.

Once you have determined the strategy for how often you will capture historical versions, how that process is done and how the lineage index is updated are a simple user- or application-driven process:

- To create a historical version, use the version manager to create a new version whose parent is the DEFAULT version. That version will then represent the state of the database at that point in time.

- To update the lineage index tables, rerun the Build lineage index command when new historical versions are created.