ArcSDE® Configuration and Tuning Guide for DB2®

ArcGIS® 9.0

Contents

Appendixes

CHAPTER 1

Getting started

Creating and populating a geodatabase is arguably a simple process, especially if you use ESRI® ArcCatalog™ or ArcToolbox™ to load the data. So, why is there a configuration and tuning guide? Well, while database creation and data loading can be relatively simple, the resulting performance may not be acceptable. It requires some effort to build a database that performs optimally. ArcSDE™ for DB2® allows you to store geographic data in a DB2 database and requires, like any other application of DB2, consideration for configuring and tuning the data stored.

This document explains how to use ArcSDE and its applications to create, store, and index the spatial data in a DB2 database.

Tuning and configuring the DB2 instance

Building an efficient geodatabase involves properly tuning and configuring the DB2 instance and proper arrangement and management of the database's tables and indexes. Chapter 2, ‘Essential configuring and tuning’, provides a brief overview of proper container placement to minimize the impact of disk I/O contention. Also, the proper selection of the grid cell sizes for the construction of the spatial index is discussed.

Arranging your data

Every table and index created in a database has a storage configuration. How you store your tables and indexes affects your database's performance.

DBTUNE storage parameters

How is the storage configuration of the tables and indexes controlled? ArcSDE reads storage parameters from the DBTUNE table to define physical data storage parameters of ArcSDE tables and indexes. The storage parameters are grouped into configuration keywords. You assign configuration keywords to your data objects (tables and indexes) when you create them from an ArcSDE client program.

The initial source of storage parameters is dbtune.sde file found under the ArcSDE etc directory. When the ArcSDE sdesetupdb2* setup command executes, the configuration parameters are read from the file and written into the DBTUNE table.

Most ArcSDE storage parameters are configuration strings and represent the entire storage configuration for a table or index. Most SDE.DBTUNE storage parameters hold the parameters of a DB2 CREATE TABLE or CREATE INDEX statement.

The sdedbtune command provides the ArcSDE administrator with an easy way to maintain the SDE.DBTUNE table. The sdedbtune command exports and imports the records of the SDE.DBTUNE table to a file in the ArcSDE etc directory.

The ArcSDE installation creates the SDE.DBTUNE table. If the dbtune.sde file is absent or empty, sdesetupdb2* creates the SDE.DBTUNE table and populates it with default configuration keywords representing the minimum ArcSDE configuration.

In almost all cases, you will populate the table with specific storage parameters for your database. Chapter 3, 'Configuring DBTUNE storage parameters', describes in detail the SDE.DBTUNE table and all possible storage parameters and default configuration keywords.

Creating spatial data in a DB2 database

ArcCatalog and ArcToolbox are graphical user interfaces (GUIs) specifically designed to simplify the creation and management of a spatial database. These applications provided the easiest method for creating spatial data in a DB2 database. With these tools you can convert existing ESRI coverages and shapefiles into ArcSDE feature classes. You can also import an existing ArcSDE export file containing the data of a business table, feature class, or raster column.

Multiversioned ArcSDE data can be edited directly with either the ArcCatalog or ArcMap GUI.

An alternative approach to creating spatial data in a DB2 database is to use the administration tools provided with ArcSDE.

Chapter 4, ‘Managing tables, feature classes, and raster columns’, describes the methods used to create and maintain spatial data in a DB2 database.

ArcSDE geodatabase maintenance

Periodically the administrator must perform various maintenance tasks on the ArcSDE geodatabase to maintain performance. Tasks such as periodically updating table and index statistics and compressing the states table are discussed.

National Language support

If you intend to support a database that does not use the DB2 default character set, you will have to take a few extra steps in creating the DB2 database. You will also need to set the national language environment of the client applications.

Chapter 5, ‘National language support’, describes how to configure the DB2 database and setup the application environment.

CHAPTER 2

Essential configuring and tuning

The performance of an ArcSDE service depends on how well you configure and tune DB2. This chapter discusses the basic guidelines for configuring a DB2 database for use with ArcSDE. It assumes that you have a basic understanding of the DB2 data structures such as tablespaces, tables, and indexes, and that you are proficient with Structured Query Language (SQL). Refer to DB2's extensive documentation, in particular *Administration Guide: Performance* and *DB2 Spatial Extender User's Guide* and *Reference,* for your DB2 release.

How much time should you spend tuning?

The importance of having a well-tuned database depends on how it is used. A database created and used by a single user does not require as much tuning as a database that is in constant use by many users. The reason is quite simple—the more people using a database, the greater the contention for its resources.

By definition, tuning is the process of sharing available resources among users by configuring the components of a database to minimize contention and maximize efficiency. The more people you have accessing your databases, the more effort is required to provide access to a finite resource.

A well-tuned DB2 database makes optimum use of available central processing unit (CPU) time and memory while minimizing disk input/output contention. Database administrators approach this task knowing that each additional hour spent will often return a lesser gain in performance. Eventually, they reach a point of diminishing returns

when it becomes impractical to continue tuning; instead, they continue to monitor the database and address performance issues as they arise.

Reducing disk I/O contention

Although disk I/O contention has been alleviated through the advancement of hardware technology it remains an important consideration to the database administrator. Disk I/O contention within a DB2 database is minimized by properly arranging the components of the database throughout the file system. Ultimately, the database administrator must reduce the possibility of one process waiting for another to complete its I/O request. This is often referred to as "waiting on I/O".

Logfiles

DB2 V8 supports dual logging on all 8.1 platforms, which means that you can now specify the logpath through the MIRRORLOGPATH DB CFG parameter. This new parameter replaces the previous registry variable DB2NEWLOGPATH2.

The maximum amount of log space has been increased to 256 GB. When the MIRRORLOGPATH parameter is enabled, DB2 can write a copy of the log to a different path.

Arranging the database components

Minimizing disk I/O contention is achieved by balancing disk I/O across the file system—positioning frequently accessed "hot" files with infrequently accessed "cold" files. Estimate the size of all the database components and determine their relative rates of access. Position the components given the amount of disk space available and the size and number of disk drives. Diagramming the disk drives and labeling them with the components help keep track of the location of each component. Have the diagram handy when you create the DB2 database.

Storage Models

There are two types of tablespace storage models in DB2: System Managed Space and Database Managed Space. ESRI recommends that you use DMS tablespaces for top performance especially if your data is expected to grow on a regular basis. Separation of regular and long data is also recommended.

Note: At release 9.0 you will now require the use of an additional DB2 global temporary table (DECLARE GLOBAL TEMPORARY TABLE). Per DB2 documentation, in order to declare global temp tables requires 'either SYSADM or DBADM privileges' or 'USE privilege on a USER TEMPORARY "tablespace". A user temporary tablespace can be created via the DB2 Control Center or from the command

line using the 'CREATE USER TEMPORARY TABLESPACE' command. For example CREATE USER TEMPORARY TABLESPACE SDESPACE PAGESIZE 4 K MANAGED BY SYSTEM USING ('/peanuts2/db2_data/sdespace') EXTENTSIZE 16 OVERHEAD 10.5 PREFETCHSIZE 16 TRANSFERRATE 0.14 BUFFERPOOL "IBMDEFAULTBP";

Separate tables from their indexes

Each time DB2 accesses an index to locate a row, it must access the table to fetch the referenced row. The disk head travels between the index and the table if they are stored on the same disk.

Whenever possible, store indexes and tables in different tablespaces so you can store them on multiple physical disks, thus eliminating repetitive and costly disk head travel.

Buffer pools

Setting up the buffer pools is absolutely critical to performance. By default, DB2 provides a single small (250) pages) buffer pool named IBMDEFAULTBP. You should create a separate buffer pool for each tablespace. The database snapshot should be reviewed to check the “buffer pool physical read” values. The buffer pool should be large enough that a snapshot of a map redraw results in a small number of physical reads.

Establish the threshold table size

As a rule, store small tables together in the same tablespace and large tables by themselves in their own tablespaces. Decide how large a table must be before it requires its own tablespace. Generally, the threshold corresponds in part to the maximum container size. Tables capable of filling the maximum size container should be stored in their own tablespace. Tables approaching this limit should also be considered. Follow the same policy for indexes.

Separate the tables and indexes into those that require their own tablespaces and those that will be grouped together. Never store tables and their indexes together in the same tablespace.

Store small tables and indexes by access

Base the decision of which small tables to store together in the same tablespace on expected access. Store tables of high access in one tablespace and tables of low access in another. Doing so allows you to position the containers of the high access tablespaces with low access containers. This same rule applies to indexes. They, too, should be divided by access.

Positioning the files

Once you have estimated the size of the containers, determine where to position them on the file system. This section provides a list of guidelines that you may not be able to follow in its entirety, given the number and size of your disk drives.

Package Cache Size (pckcachesize)

Package cache size specifies the amount of memory allocated for caching dynamic and static SQL requests. This is allocated at database startup and freed at shutdown. If DB2 does not find an SQL statement in the package cache, the statement will need to be recompiled (dynamic SQL) or loaded from the package catalog (static SQL), which can consume considerable time. In order to have a well performing database it is important to make maximum use of the package cache size. ESRI recommends that you start with an initial size of 6000 and then monitor the package cache lookups and package cache insert values in a database snapshot.

AIX tuning

ESRI recommends turning OFF the following variables on AIX® in order to improve performance.

These variables are used in conjunction with each other to allow DB2 to use a multifunction, multiservice access platform (MMAP) as an alternative method of I/O. This is used to avoid operating system locks when multiple processes are writing to different sections of the same file. Default is ON.

db2set DB2_MMAP_READ=OFF

db2set DB2_MMAP_WRITE=OFF

Make sure that the DB2 logs have their own disks apart from index and data. ESRI recommends that you set the maxuproc value to a higher value than the default so that all processes owned by the SDE and DB2 instance users can run when invoked.

Network tuning can consist of the following

Place the client and server on the same switch or connect them via a minimum number of routers Set the following ifconfig parameters: tcp_nodelay 1, rfc1323 0. Check to see if the following parameters exist and change them as follows –

a. Set RX checksum offload yes.

b. Set TCP large send offload yes.

ArcSDE system tablespaces

The ArcSDE system tablespaces store the ArcSDE and geodatabase system tables and indexes created by the ArcSDE sdesetupdb2 command. The number and placement of the tablespaces depend on what you intend to use the ArcSDE database for.

The placement of these tables and their indexes is controlled by the storage parameters of the dbtune DATA_DICTIONARY configuration keyword. The DATA_DICTIONARY keyword is used exclusively for the creation of the ArcSDE and geodatabase system tables.

Multiversioned databases that support ArcGIS OLTP applications have a highly active state tree. The state tree maintains the states of all editing operations that have occurred on tables registered as multiversioned. Four ArcSDE system tables—STATES, STATE_LINEAGES, MVTABLES_MODIFIED, and VERSIONS—maintain the transaction information of the versioned database's state tree. In this type of environment these four tables and their indexes have their own DATA_DICTIONARY configuration keyword storage parameters.

In an active multiversioned database, the STATES_LINEAGE table can easily grow beyond one million records, occupying more than 26 MB of tablespace. The STATES table is much smaller, storing approximately 5,000 records, occupying about 2 MB of tablespace. The MVTABLES_MODIFIED table typically has approximately 50,000 records occupying about 1 MB of tablespace. The VERSIONS table is usually quite small with less than 100 rows occupying about 64 KB.

For most applications you can probably create a tablespace for the ArcSDE system tables and one for their indexes on different disk drives and set the DATA_DICTIONARY parameters accordingly. For highly active editing ArcGIS applications, the STATES, STATES_LINEAGE, and MVTABLES_MODIFIED tables and their indexes need to be created in separate tablespaces and positioned across the file system to minimize disk I/O contention.

If you are not using a multiversioned database, the aforementioned tables are dormant, in which case the tables can be stored with the other ArcSDE system tables and indexes.

The remainder of the ArcSDE and geodatabase system tables store information relating to schema changes. They are relatively small and have a low frequency of I/O. They should be grouped together in two separate tablespaces—one for tables and one for indexes—and positioned with other tablespaces of high activity.

To summarize, if you are creating an active multiversioned database, create a 70 MB tablespace to store ArcSDE tables. On a separate disk drive create a 30 MB tablespace for the indexes.

If you are not going to use a multiversioned database, reduce the extent sizes of the STATE_LINEAGES, STATES, and MVTABLES_MODIFIED tables and their indexes to 40 KB. Create two 5 MB tablespaces on separate disk drives—one for the tables and one for the indexes.

For more information about the DATA_DICTIONARY configuration keyword, see Chapter 3, 'Configuring DBTUNE storage parameters'.

A note about RAID storage devices

RAID, short for Redundant Array of Inexpensive Disks, is a method whereby information is spread across several disk drives to reduce latency, increase data availability, improve data safety or a combination of these. In RAID storage, two or more physical disk drives are combined into one physical configurable device. RAID storage operates in various modes identified as RAID 0, RAID 1, RAID 4, RAID 5 and RAID 10.

With RAID 0, also called "stripe mode" data may be spread over multiple disks. The first 4 KB might be written to disk 0, the second 4 KB block to disk 1, the third to disk 2 and so on. Data is distributed back to disk 0 after disk N is written. RAID 0 may be faster during sequential reads and writes since successive blocks can be read or written nearly in parallel over multiple disks. However, RAID 0 provides no extra data safety since there is no duplication of any data.

RAID 1 provides an exact duplicate of the file information maintained on one disk in the array. RAID 1 can be used on two disks with zero or more spare disks. RAID 1 protects data through redundancy; however, it is always slower to write since the data must be written to at least two locations.

RAID 4 is similar to RAID 1 except that one disk stores parity information and not duplicate data. There need be only one parity disk to protect any number of data disks. However, the disk holding parity data will become a bottleneck on writes since it will always be written to. Like RAID 1, RAID 4 is also slower on writes because two physical blocks—the data block and the parity block—must be written for each logical block sent. The disk holding the parity data becomes a bottleneck since all data writes will affect this disk.

RAID 5 may be the most useful mode, since it eliminates the bottleneck of RAID 4, but still provides data protection. In RAID 5, parity information is distributed evenly among the participating disk drives.

RAID 1+0 is also useful. A group of drives are duplicated, akin to RAID 1, and the resulting group is striped, akin to RAID 0. The real benefit of this arrangement lies in the duplication, which provides data protection.

While RAID 1, RAID 1+0 and RAID 5 provide an additional level of protection, ESRI does not recommend relying solely on a disk configuration for safeguarding data.

While offering high availability and a good price to performance ratio, RAID-5 is by far the most popular RAID level in use today. It does have a "write penalty" associated with it but, *fast write cache* (FWC), which is now fairly common with RAID adaptors, can reduce the effects of the write penalty.

Since a RAID-5 array enables parallel I/O to be used as the data is spread over multiple physical disks in the array, you need to set the DB2_PARALLEL_IO registry variable to *. At DB2 V8, the DB2_STRIPED_CONTAINERS = YES is the default behavior used in creating tablespace containers.

Dealing with deadlocks

The following situation may not be an uncommon occurrence, where the new_edit_state stored procedure call has deadlocked the calling application, as well as blocked all other use of the SDE database.

Imagine a scenario where the stored procedure acquires a large number of rowlocks on the state_lineages table, exceeding a threshold for maximum number of locks, and attempting to escalate to an exclusive table lock. Unfortunately, the calling application's query already holds a shared lock on the state_lineages table, thus leading to a deadlock. The large number of rowlocks will arise from having a very deep state lineage. This, along with having a very low setting for lock list size, guarantees that there would be problems. Given how DB2 handles lock escalation, there are other deadlock scenarios imaginable.

Once again, the implication here is that deadlocks may not be an uncommon occurrence at user sites, depending on the user application and on the database configuration. One again it is to be noted that the problem may be aggravated with very deep states lineages. Fortunately, DB2 does provide tuning parameters to control the size of locklist (LOCKLIST), max percentage of locks an application can hold (MAXLOCKS), deadlock detection (DLCHKTIME, LOCKTIMEOUT), as well as deadlock rollback behavior (DB2LOCK_TO_RB).

Briefly, to increase locklist capacity and lock escalation threshold, modify LOCKLIST and MAXLOCKS parameters, respectively.

To set or prevent infinite wait deadlocks, or tune lock request wait time, modify LOCKTIMEOUT.

To tune period between deadlock detection checks, adjust DLCHKTIME.

By default, a lock timeout will rollback the request transaction. To change this behavior to only rollback the statement making the lock request, modify DB2LOCK_TO_RB with 'db2set DB2LOCK_TO_RB=STATEMENT'. The default behavior should be fine for SDE, though.

See the DB2 documentation or performance-tuning guides for detailed info on properly setting these parameters.

To view locklist settings issue the following command -

db2 get db cfg

```
  Max storage for lock list (4KB)            (LOCKLIST) = 50
  Interval for checking deadlock (ms)        (DLCHKTIME) = 10000
  Percent. of lock lists per application     (MAXLOCKS) = 22
  Lock timeout (sec)                    (LOCKTIMEOUT) = -1
  Max number of active applications          (MAXAPPLS) = AUTOMATIC
```

(For DB2LOCK_TO_RB registry value, use 'db2set' and look for 'DB2LOCK_TO_RB=')

Quick check of Lock list capacity - total number of locks:
  Upper max # of locks == (LOCKLIST * 4096) / 36
  Lower max # of locks == (LOCKLIST * 4096) / 72

Quick check for maximum locks allowed before escalation:
  Upper Threshold = MAXLOCKS * (LOCKLIST * 4096 bytes) / (100 * 36)
  Lower Threshold = MAXLOCKS * (LOCKLIST * 4096 bytes) / (100 * 72)

To set LOCKLIST:

1. Estimate maximum number of active applications (MAXAPPLS, if set).
   Estimate average # of locks per application.

2. Estimate lower & upper lock list size:

   (Avg # locks per application * 36 * MAXAPPLS) / 4096
   (Avg # locks per application * 72 * MAXAPPLS) / 4096

   where:
   72 == # bytes of first lock on object
   36 == # bytes of additional locks on object

3. Set initial LOCKLIST somewhere between upper & lower bounds.

   For example:
   db2 update db cfg using LOCKLIST 200

To set MAXLOCKS:
1. Determine percentage of locklist any single application can consume before lock escalation occurs. This could be a flat percentage, or based on common transaction volumes.

For Example, if applications are to be allowed 2X average # of locks:

```
100 * (Avg  # locks per application * 2 * 72 bytes per lock)
 / (LOCKLIST * 4096 bytes)
```

For example:
db2 update db cfg using MAXLOCKS 22

Additional tuning of locklist parameters involves use of the Snapshot and Event Monitors. Look for the following info:

Use Snapshot Monitor at database level for
- Total lock list memory in use
- Number of lock escallations that have occurred

Use Event Monitor for
- Max # of locks held by transaction

A few useful tools to diagnose lock problems:

1. Find db2 application ids for sde processes:

SELECT appl_id FROM TABLE(SNAPSHOT_APPL_INFO('SDE',-1)) as SNAPSHOT_APPL_INFO where appl_name like 'gsrvr%'

SELECT appl_id,appl_name FROM TABLE(SNAPSHOT_APPL_INFO('SDE',-1))

2. Use snapshots for lock and application info.
For example:

db2 get snapshot for locks on sde > all_locks.txt

db2 get snapshot for locks for application applid '*LOCAL.DB2.00AB42215335' > app_locks.txt

db2 get snapshot for application applid '*LOCAL.DB2.00AB42215335' > app_info.txt

A quick search on the snapshot output for items of interest:

Application status = Lock-wait
Locks held by application = 1254
Number of SQL requests since last commit = 12
Open local cursors = 1
Most recent operation = Execute

Object Type = Table
Tablespace Name = USERSPACE1

Table Schema	= SDE
Table Name	= STATE_LINEAGES
Mode	= X
Status	= Converting
Current Mode	= IX
Lock Escalation	= YES

As noted above, very deep lineages may be an issue for acquiring large number of row locks. The following SQL statements can provide a quick check of lineage depths and of max lineage depth:

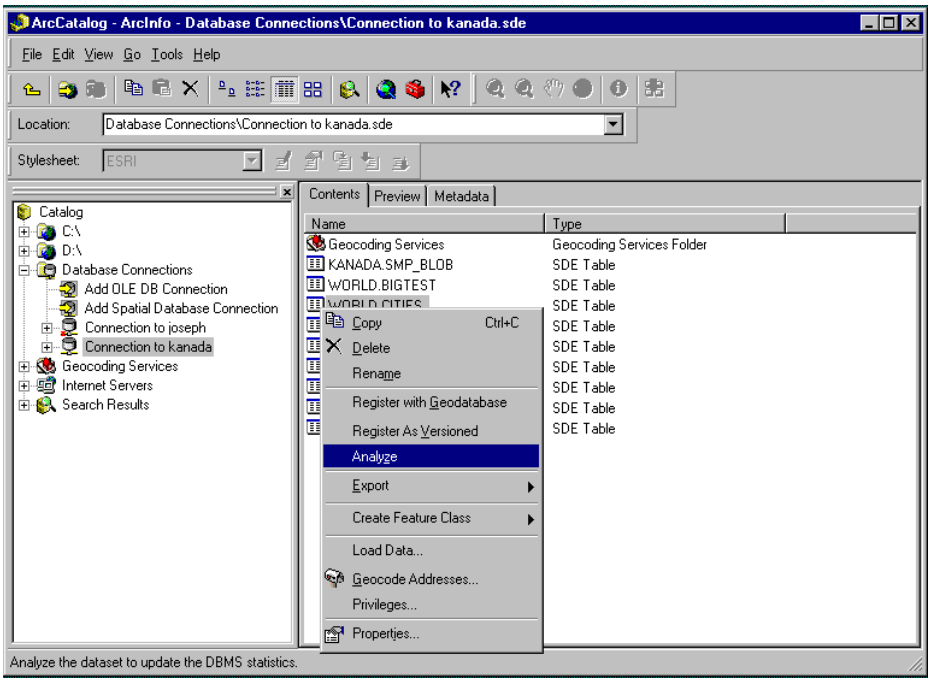
select count(*) from state_lineages group by lineage_name

select max(a.depth) from (select count(*) from state_lineages group by lineage_name) a(depth)

Updating DB2 statistics

For the best performance, the statistics of the ArcSDE tables and indexes that you have stored in DB2 must be kept up-to-date.

In ArcCatalog, to update the statistics of all of the tables and indexes within a feature dataset, right-click the feature dataset and click Analyze. To update the tables and indexes within a feature class, right-click the feature class and click Analyze.



From the command line, use the UPDATE_DBMS_STATS operation of the sdetable administration command to update the statistics for all the tables and indexes of a feature class. It is better to use the sdetable UPDATE_DBMS_STATS operation rather than individually analyzing the tables with the DB2 RUNSTATS statement because it updates the statistics for all tables of a feature class. In addition to the business table, an ArcSDE for DB2 feature class may include an adds and deletes table as well.

To have the UPDATE_DBMS_STATS operation update the statistics for all the required tables, do not specify the -K (schema object) option.

```
sdetable -o update_dbms_stats -t roads -m compute -u av -p mo
```

When the feature class is registered as multiversioned, the adds and deletes tables are created to hold the business table's added and deleted records. The version registration process automatically updates the statistics for all the required tables at the time it is registered.

Periodically update the statistics of dynamic tables and indexes to ensure that the DB2 optimizer continues to choose an optimum execution plan. To save time, you can update the statistics of all the data objects within a feature dataset in ArcCatalog.

If you decide to update the statistics of all or some of the feature class tables with the DB2 RUNSTATS statement, use the following syntax:

```
RUNSTATS ON TABLE <table_name> WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

For more information on the DB2 SQL RUNSTATS statement, refer to the *IBM DB2 Universal Database Command Reference*.

The statistics of a table's indexes are automatically computed when the table is analyzed, so there is no need to analyze the indexes separately. However, if you need to do so you can use the UPDATE_DBMS_STATS -n option with the index name.

The example below illustrates how the statistics for the roads_ix index of the roads table can be updated.

```
sdetable -o update_dbms_stats -t roads -n roads_ix -u av -p mo
```

For more information on analyzing geodatabase objects from ArcCatalog, refer to *Building a Geodatabase*.

For more information on the sdetable administration command and the UPDATE_DBMS_STATS operation, refer to ArcSDE Developer Help.

Tuning the spatial index

Applications querying the two-dimensional geographic data contained in a spatial column require an index strategy that will quickly identify all geometries lying within a given

extent. For this reason the DB2 Spatial Extender provides the three-tiered grid spatial index.

The two-dimensional spatial index differs from the traditional hierarchical Btree index provided by DB2. To better understand the difference, first review how a Btree index is structured and used.

The top level of a Btree index, the root node contains one key for each node at the next level. The value of each of these keys is the largest existing key value for the corresponding node at the next level. Depending on the number of values in the base table, several intermediate nodes may be needed to bridge the root node with the leaf nodes, which hold the actual base table row IDs.

The DB2 database manager searches a Btree index starting at the root node, working its way through the intermediate nodes until it reaches the leaf node with the row ID of the base table.

The Btree index may not be applied to a spatial column because the two-dimensional characteristic of the spatial column requires the structure of a spatial index. For the same reason, you may not apply a spatial index to a nonspatial column, and a spatial index may not be applied to a composite column of any kind.

The spatial index's `CREATE INDEX` syntax includes the additional *USING* clause, which directs DB2 to use the Spatial Extender's spatial index rather than a Btree index. The full syntax is as follows:

```
create index <index_name> on <table> (<spatial column>)
using db2gse.spatial_index (<grid level 1>, [grid level 2] , [grid level
3])
```

The addition of the *USING* clause distinguishes the spatial index from the Btree index. The *db2gse* schema name must qualify the *spatial_index* index extension name as this statement does not follow the current function path.

Because of the simple nature of the data that a Btree was designed to index, the database designer merely directs DB2 to create the index on one or more table columns. However since spatial data is complex it requires the designer to understand its relative size distribution. The designer must determine the optimum size and number of the spatial index's grid levels.

The grid levels [grid level 1], [grid level 2], [grid level 3]) are entered by increasing cell size. Thus the second level must have a larger cell size than the first and the third a larger cell size than the second. The first grid level is mandatory, but you may disable the second and third with a zero value (0).

How the Spatial Extender generates a spatial index

The DB2 Spatial Extender constructs a spatial index as follows:

1. The Spatial Extender intersects each geometry's envelope with the grid beginning with the first level.

2. If less than four intersections occur with the first grid level, the Spatial Extender enters the geometry ID and the intersecting grid cell IDs in the spatial index and continues with the next geometry.

3. If the Spatial Extender detects more than four intersections, it intersects the geometry with the second grid level. If you have not enabled the second grid level, the Spatial Extender enters the geometry ID and grid cell IDs in the spatial index and continues with the next geometry.

4. If less than four intersections occur with the second grid level, the Spatial Extender enters the geometry ID and the intersecting grid cell IDs in the spatial index and continues with the next geometry.

5. If the Spatial Extender detects more than four intersections, it intersects the geometry with the third grid level. If you have not enabled the third grid level, the Spatial Extender enters the geometry ID and grid cell IDs in the spatial index and continues with the next geometry.

6. The Spatial Extender enters the geometry ID and the intersecting grid cell IDs of the third level in the spatial index and continues with the next geometry.

The Spatial Extender does not actually create polygon grid structure of any sort. The Spatial Extender manifests each grid level parametrically by defining the origin as the x,y offsets of the column's spatial reference system extending into positive coordinate space. Using a parametric grid the Spatial Extender generates the intersections mathematically.

How the Spatial Extender uses the spatial index

The Spatial Extender uses a spatial index to improve the performance of a spatial query. Consider the box query—the most basic and probably the most popular spatial query. The box query returns geometries of a spatial column that intersect a user-defined box. If a spatial index does not exist, the Spatial Extender must compare all of a spatial column's geometries with the box.

Using the spatial index, the Spatial Extender identifies index grid entries that intersect the box. Since the spatial index is ordered on a grid, the Spatial Extender quickly obtains a list of candidate geometries. The process just described is referred to as the *first pass*.

A *second pass* disqualifies candidate geometries whose envelopes do not intersect the box.

A *third pass* compares the actual coordinates of the candidate geometry with the box to determine whether or not the geometry intersects the box. This last complex process of

comparison operates on a subset of the table rows, significantly reduced by the first two passes.

All spatial queries perform the three passes with the exception of the EnvelopesIntersect function. It performs only the first two passes and was designed for display operations that use display driver clipping routines and that don't require the granularity of the third pass.

Selecting the optimum grid cell sizes

Selecting the grid cell size is complicated by the fact that envelopes of irregularly shaped geometries do not fit neatly within a grid cell. Because of this irregularity, some geometry envelopes intersect several grids, while others fit inside a single grid cell. On the flip side, grid cells may intersect several geometry envelopes depending on the spatial distribution of the data.

A spatial index performs well when you enable the correct number of levels and their grid cell sizes to fit the data. To simplify this discussion, first consider a spatial column containing geometry whose size is uniform. In this case, it is not necessary to create a multileveled spatial index since a single grid level will suffice. Create a spatial index with a single grid level whose grid cell size is 1.5 times the size of the average geometry envelope. Since point data has a small envelope, the grid size could also be small.

The general rule is that the grid size should be about 1/10 of the typical query window size. For point data, only a single grid level should be necessary.

While testing your application, you may find that it performs better with a larger grid cell size because each grid cell references more geometries, enabling the *first pass* to discard nonqualifying geometries faster. However, if you continue to increase the grid cell size, performance deteriorates as the number of geometries filtered by the *second pass* increases.

DB2 Spatial Extender provides a utility, the *Index Advisor*, that lets you create a simulated grid index and tune this index into a model for a real index. It also determines whether to retain or replace an existing grid index.

Following is an example of how to use the *Index Advisor* to return detailed information about an existing grid index whose fully qualified name is mydb.myindex .

```
gseidx connect to mydb user test using test get geometry statistics for
index mydb.myindex detail
```

Both shp2sde and cov2sde use a similar algorithm to calculate the default spatial index for grid size level 1 when the optional -g option isn't present. The defaults for grid size level_2 and level_3 are always set to ZEROs, where shp2sde is based on the shapefile's extent and cov2sde is based on the ARC/INFO coverage's extent.

Selecting the number of levels

Few spatial columns contain geometry of the same relative size. However, geometries of most spatial columns can be grouped into size intervals. For instance, consider a spatial column of county parcels containing a vast number of small parcels clustered in the urban areas surrounded by a few large rural parcels. These situations are common and require the use of a multilevel spatial index. To select the grid cell sizes of each level, determine the intervals of geometry envelope sizes. Create a spatial index with gridlevel cell sizes slightly larger than each interval. Test the index by performing queries against the spatial column using your application. Try adjusting the grid sizes up or down slightly to determine if an appreciable improvement in performance can be obtained.

For further details on this topic refer to *Chapter 11 "Using indexes and views to access spatial data"* in the IBM *DB2 Spatial Extender User's Guide* and *Reference*.

CHAPTER 3

Configuring DBTUNE storage parameters

The DBTUNE storage parameters are stored in the DBTUNE table. The DBTUNE table, along with all other metadata tables, is created during the setup phase that follows the installation of the ArcSDE software. The ArcSDE software install creates a DBTUNE file under the etc directory from which the DBTUNE table is populated. If no DBTUNE file is present during setup, ArcSDE will populate the DBTUNE table with default values.

DBTUNE storage parameters allow you to control how ArcSDE clients create objects within an DB2 database. They allow you to determine such things as how to allocate space to a table or index, which tablespace a table or index is created in, and other DB2-specific storage attributes.

This chapter discusses the mechanism by which ArcSDE manages storage parameters that you provide and how ArcSDE applies them to specific statements submitted to DB2 when creating ArcSDE tables, indexes, and other objects in a DB2 database.

Many DBTUNE parameters include corresponding DB2 storage parameters.

The DBTUNE table and file

The DBTUNE storage parameters are maintained in the database in the DBTUNE metadata table. The DBTUNE table, along with all other metadata tables, is created during the setup phase that follows the installation of the ArcSDE software

The DBTUNE table is populated with specific default values, but these values may be changed. Several example versions of the dbtune file are provided for the installation media for ArcSDE.

DBTUNE parameters

Parameters define the storage configuration of simple objects, such as tables and indexes, as well as complex objects such as feature classes, networks classes, and raster columns. Many different parameters may be grouped together under a single *configuration keyword.*

ArcSDE client applications and some ArcSDE administration tools refererence one or more configuration keywords when creating an object.

When a configuration keyword is specified by an ArcSDE application or administration tool, the parameters within the associated *parameter group* are searched and the necessary configuration strings are incorporated into the CREATE TABLE or CREATE INDEX statement submitted to the DB2 database server.

The structure of the DBTUNE file

Storage parameters in a dbtune file occur as a combination of *parameter name* and *configuration string* delimited by white space. A configuration string value may span multiple lines and must be enclosed in double quotes. For example, a valid specification for the parameter named A_INDEX_ROWID might look like this:

```
A_INDEX_ROWID          ""
```

Storage parameters are grouped by *keyword*. Each parameter group is introduced by its keyword which is prefixed by two pound signs, "##". A line beginning with the word "END" terminates each parameter group. Double pound signs, "##", signal the presence of a keyword but are not part of the keyword itself.

For example, a group of parameters under the configuration keyword "WILSON_DATA" may look like this:

```
##WILSON_DATA

A_INDEX_ROWID   ""
```

```
A_INDEX_SHAPE   ""
A_INDEX_STATEID ""
B_STORAGE       "IN <TABLESPACE> INDEX IN <TABLESPACE> LONG IN <TABLESPACE>"
END
```

In special circumstances ArcSDE references a *compound keyword* when creating database objects. The compound keyword allows ArcSDE to create related database objects having different object creation parameters to accommodate different performance needs. A compound keyword consists of a configuration keyword plus a suffix delimited by a double colon, "`::`". For example:

```
##ELECTRIC::DESC
```

Comments within the dbtune file are indicated by a single pound sign, "#". Default versions of the dbtune file provided in the general software release contain lines that are commented out. Such lines are used as placeholders for certain storage parameters, such as tablespace name, and may be restored by removing the comment character and editing the line.

Any number of parameter groups may be specified in a dbtune file. However, certain groups and certain parameter names within groups are expected to exist and will be created in the DBTUNE table if they do not exist in the dbtune file.

The structure of the DBTUNE table

The DBTUNE table has the following definition:

```
Name             Null?     Datatype

keyword          not null  varchar(32)
parameter_name   not null  varchar(32)
config_string    null      varchar(2048)
```

The keyword field stores the configuration keyword for the group in which each parameter is found. For a single keyword, there may be many different parameter_name values, each one associated with a config_string value.

After creating the DBTUNE table, the setup phase of the ArcSDE installation populates the table with the contents of the dbtune.sde file, which it expects to find in the %SDEHOME%/etc directory under Windows or the $SDEHOME/etc under Unix. If the DBTUNE table already exists, the ArcSDE setup phase will not alter its contents.

Managing the DBTUNE table

Through the use of the sdedbtune utility you can initialize or alter the contents of the DBTUNE table. This utility guarantees that the DBTUNE table maintains a certain default set of keywords, parameters and parameter values.

In addition to the default keywords and parameters, you may add to the DBTUNE file keywords and configuration values of your choosing.

Note: ESRI does *not* recommend using SQL to directly alter the contents of the DBTUNE table. Doing so would bypass certain protections written into the sdedbtune utility, possibly leading to reduced performance.

Initializing the DBTUNE table

The dbtune.sde file provided with the install media contains default values, which are used to initialize the DBTUNE table.

On UNIX® systems, you can modify the dbtune.sde file prior to running the sdesetupdb2 command. On Windows NT® systems the setup phase is part of the install, so you will have to edit the file and use the sdedbtune import operation to customize the DBTUNE table.

If the dbtune.sde file is missing when the sdesetupdb2 command is executed, or if specific parameters are missing because the dbtune.sde file has been altered, the ArcSDE software will enter *software default* values into the dbtune table.

Customizing the DBTUNE file

Prior to creating ArcSDE objects, you should customize the dbtune.sde file by specifying the tablespace names for storage parameters. In the default dbtune.sde file, the tablespace entries in the dbtune.sde file have been commented out with the "#" character.

To customize the dbtune.sde file, remove the comment character preceding each tablespace specification and enter the names of the tablespaces where you wish to store your ArcSDE tables and indexes. Be careful not to remove the double quotation marks that surround the configuration strings.

Follow the procedure provided in the following example for updating the DB2 tablespace parameter in the dbtune.sde file.

The DEFAULTS configuration keyword in the dbtune.sde file contains the B_STORAGE storage parameter with the DB2 tablespace parameter commented out.

```
##DEFAULTS
```

```
A_INDEX_ROWID        ""
A_INDEX_SHAPE     ""
A_INDEX_STATEID      ""
A_INDEX_USER         ""
#B_STORAGE"IN <TABLESPACE> INDEX IN <TABLESPACE> LONG IN <TABLESPACE>"
```

Edit the dbtune.sde file, remove the "#" comment character, and enter the name of the tablespace you want to store business tables in by default.

```
##DEFAULTS
```

```
##DEFAULTS
A_INDEX_ROWID        ""
A_INDEX_SHAPE     ""
A_INDEX_STATEID      ""
A_INDEX_USER         ""
B_STORAGE  "IN SDEDB2 INDEX IN SDEINDEX LONG IN SDELOBS"
```

When the setup program loads your customized dbtune parameter, configuration keywords and storage parameters are written into the DBTUNE table.

Editing the DBTUNE table

If you need to change the contents of the DBTUNE table after it is loaded, you should use the sdedbtune utility and follow these steps.

1. Export the DBTUNE table to a text file using the sdedbtune –o export command.

2. Edit the resulting file with a UNIX file-based editor, such as "vi", or a Windows file-based editor such as notepad.

3. Import the edited file to the DBTUNE table using **the sdedbtune –o import** command.

In the following example, the DBTUNE table is exported to the "dbtune.out" file. Then, the file is edited with the UNIX "vi" file-based editor.

```
$ sdedbtune -o export -f dbtune.out -u sde -p sdepasswd
```

```
ArcSDE   9.0               Wed Oct  4 22:32:44 PDT 2003
Attribute        Administration Utility
-----------------------------------------------------

        Successfully exported to file SDEHOME\etc\dbtune.out
```

```
$ vi dbtune.out
```

```
$ sdedbtune -o import -f dbtune.out -u sde -p sdepasswd -N
```

```
ArcSDE   9.0                Wed Oct  4 22:32:44 PDT 2003
Attribute        Administration Utility
--------------------------------------------------

        Successfully imported from file SDEHOME\etc\dbtune.out
```

The sdedbtune administration tool always exports and imports from the $SDEHOME/etc directory. You cannot specify that files should be located into another directory. By not allowing the relocation of the file, the sdedbtune command ensures the dbtune parameters remain under the ownership of the ArcSDE administrator.

Adding keywords to the DBTUNE table

You may add parameter groups to the DBTUNE table for any special purpose. For instance, you may wish to create certain feature classes in a newly created tablespace that is segregated from the rest of the data.

To add keywords, follow the instructions above for editing the DBTUNE table. When you edit the export file, it is often a good idea to create a new parameter group as a cut and paste copy of an existing parameter group in order to avoid introducing syntax errors. You may then edit the configuration keyword and any of the strings to desired new values before saving the dbtune file and importing it back into the DBTUNE table.

Using the DBTUNE table

At its most basic level, the DBTUNE table provides configuration strings that ArcSDE appends to a CREATE TABLE or CREATE INDEX statement in SQL. Therefore, the configuration strings specify storage parameters that must be considered valid by the DB2 server.

Selecting the configuration string

The choice of configuration strings by an ArcSDE application depends on the operation being performed and the type of object it is being performed on, as well as the configuration keyword. For example, if the type of operation is CREATE TABLE and the type of table being created is a business table, the parameter_name of B_STORAGE will be used to determine the configuration string.

The ArcSDE application then searches the DBTUNE table for a configuration keyword that matches the one entered and uses the configuration string of the appropriate storage parameter.

If the application cannot find the requested configuration string within the specified parameter group, it searches the DEFAULT parameter group. If the requested configuration string cannot be located within the DEFAULT parameter group, ArcSDE uses the DB2 defaults to create the table or index.

Table parameters

Table parameters define the storage configuration of a DB2 table. ArcSDE appends the configuration string associated with the parameter to the CREATE TABLE statement prior to submitting the statement to DB2.

Valid entries for an ArcSDE table include any parameter allowable to the right of the column list in the CREATE TABLE statement, especially including the TABLESPACE and STORAGE clauses.

For example, if you define the B_STORAGE parameter in this manner:

B_STORAGE "IN SDEDB2 INDEX IN SDEINDEX LONG IN SDELOBS"

ArcSDE would execute the following DB2 CREATE TABLE command:

```
CREATE TABLE roads (road_id integer, name varchar2(32), surface_code
integer) in SDEDB2 index in SDEINDEX long in SDELOBS
```

Index parameters

Index parameters define the storage configuration of a DB2 index. ArcSDE appends the index parameter to a DB2 CREATE INDEX statement prior to submitting the statement to DB2.

Valid entries in an ArcSDE index parameter include any parameter allowable by the DB2 server to the right of the column list of the CREATE INDEX statement, especially including the TABLESPACE and STORAGE clauses.

Defining the storage parameters

Configuration keywords may include any combination of three basic types of storage parameters: metaparameters, table parameters, and index parameters.

Meta parameters define the way certain types of data will be stored, the environment of a configuration keyword, or a comment that describes the configuration keyword. Table and index parameters establish the storage characteristics of tables and indexes.

The business table storage parameter

A business table is any DB2 table created by an ArcSDE client, the sdetable administration command, or the ArcSDE C application programming interface (API) SE_table_create function.

Use the DBTUNE table's B_STORAGE storage parameter to define the storage configuration of a business table.

The business table index storage parameters

Three index storage parameters exist to support the creation of business table indexes.

The B_INDEX_USER storage parameter holds the storage configuration for user-defined indexes created with the C API function SE_table_create_index and the create_index operation of the sdetable command.

The B_INDEX_ROWID storage parameter holds the storage configuration of the index ArcSDE creates on a register table's object ID column, commonly referred to as the ROWID.

Note: ArcSDE registers all tables that it creates. Tables not created by ArcSDE can also be registered with the alter_reg operation of the sdetable command or with ArcCatalog. The SDE.TABLE_REGISTRY system table maintains a list of the currently registered tables.

The B_INDEX_SHAPE storage parameter holds the storage configuration of the spatial column index that ArcSDE creates when a spatial column is added to a business table. This index is created by the ArcSDE C API function SE_layer_create. This function is called by ArcInfo™ when it creates a feature class and by the add operation of the sdelayer command.

Multiversioned table storage parameters

Registering a business table as multiversioned allows multiple users to maintain and edit their copy of the object. At appropriate intervals, each user merges the changes made to the copy along with the changes made by other users and reconciles any conflicts that arise when the same rows are modified.

ArcSDE creates two tables—the adds table and the deletes table—for each table that is registered as multiversioned.

The A_STORAGE storage parameter maintains the storage configuration of the adds table. Four other storage parameters hold the storage configuration of the indexes of the adds table. The adds table is named A<n>, where <n> is the registration ID listed in the SDE.TABLE_REGISTRY system table. For instance, if the business table ROADS is listed with a registration ID of 10, ArcSDE creates the adds table as A10.

The A_INDEX_ROWID storage parameter holds the storage configuration of the index that ArcSDE creates on the multiversion object ID column, commonly referred to as the ROWID. The adds table ROWID index is named A<n>_ROWID_IX1, where <n> is the business table's registration ID, which the adds table is associated with.

The A_INDEX_STATEID storage parameter holds the storage configuration of the index that ArcSDE creates on the adds table's SDE_STATE_ID column. The SDE_STATE_ID column index is called A<n>_STATE_IX2, where <n> is the business table's registration ID, which the adds table is associated with.

The A_INDEX_SHAPE storage parameter holds the storage configuration of the index that ArcSDE creates on the adds table's spatial column. If the business table contains a spatial column, the column and the index on it are duplicated in the adds table. The adds table's spatial column index is called A<n>_IX1_A, where <n> is the layer ID of the feature class as it is listed in the SDE.LAYERS table.

The A_INDEX_USER storage parameter holds the storage configuration of user-defined indexes that ArcSDE creates on the adds table. The user-defined indexes on the business tables are duplicated on the adds table.

The D_STORAGE storage parameter holds the storage configuration of the deletes table. Two other storage parameters hold the storage configuration of the indexes that ArcSDE creates on the deletes table. The deletes table is named D<n>, where <n> is the registration ID listed in the SDE.TABLE_REGISTRY system table. For instance, if the business table ROADS is listed with a registration ID of 10, ArcSDE creates the deletes table as D10.

The D_INDEX_STATE_ROWID storage parameter holds the storage configuration of the D<n>_IDX1 index that ArcSDE creates on the deletes table's SDE_STATE_ID and SDE_DELETES_ROW_ID columns.

The D_INDEX_DELETED_AT storage parameter holds the storage configuration of the D<n>_IDX2 index that ArcSDE creates on the deletes table's SDE_DELETED_AT column.

Note: If a configuration keyword is not specified when the registration of a business table is converted from single-version to multiversion, the adds and deletes tables and

their indexes are created with the storage parameters of the configuration keyword the business table was created with.

Raster table storage parameters

A raster column added to a business table is actually a foreign key reference to raster data stored in a schema consisting of four tables and five supporting indexes.

The RAS_STORAGE storage parameter holds the DB2 CREATE TABLE storage configuration of the RAS table.

The RAS_INDEX_ID storage parameter holds the DB2 CREATE TABLE storage configuration of the RAS table index.

The BND_STORAGE storage parameter holds the DB2 CREATE TABLE storage configuration of the BND table index.

The BND_INDEX_ COMPOSITE storage parameter holds the DB2 CREATE INDEX storage configuration of the BND table's composite column index.

The BND_INDEX_ID storage parameter holds the DB2 CREATE INDEX storage configuration of the BND table's rid column index.

The AUX_STORAGE storage parameter holds the DB2 CREATE TABLE storage configuration of the AUX table.

The AUX_INDEX_COMPOSITE storage parameter holds the DB2 CREATE INDEX storage configuration of the AUX table's index.

The BLK_STORAGE storage parameter holds the DB2 CREATE TABLE storage configuration of the BLK table.

The BLK_INDEX_COMPOSITE storage parameter holds the DB2 CREATE TABLE storage configuration of the BLK table's index.

Arranging storage parameters by keyword

Storage parameters of the DBTUNE table are grouped by keyword. The following keywords are present by default in the DBTUNE table.

- DEFAULTS
- DATA_DICTIONARY
- IMS_METADATARELATIONSHIPS
- IMS_METADATA
- IMS_METADATATAGS
- IMS_METADATATHUMBNAILS
- IMS_METADATAUSERS
- IMS_METADATAVALUES
- IMS_METADATAWORDINDEX
- IMS_METADATAWORD
- LOGFILE_DEFAULTS
- NETWORK_DEFAULTS
- NETWORK_DEFAULTS::DESC
- NETWORK_DEFAULTS::NETWORK
- SURVEY_MULTI_BINARY
- TOPOLOGY_DEFAULTS
- TOPOLOGY_DEFAULTS::DIRTYAREAS

DEFAULTS keyword

Each DBTUNE table has a fully populated DEFAULTS keyword.

The DEFAULTS keyword can be selected whenever you create a table, index, feature class, or raster column. If you do not select a keyword for one of these objects, the DEFAULTS keyword is used. If you do not include a storage parameter in a keyword you have defined, ArcSDE substitutes the storage parameter from the DEFAULTS keyword.

The DEFAULTS keyword relieves you of the need to define all the storage parameters for each of your keywords. The storage parameters of the DEFAULTS keyword should be populated with values that represent the average storage configuration of your data.

During installation, if the ArcSDE software detects a missing DEFAULTS keyword storage parameter in the dbtune.sde file, it automatically adds the storage parameter. If you import a DBTUNE file with the sdedbtune command, the command automatically adds default storage parameters that are missing. ArcSDE will detect the presence of the following list of storage parameters and insert the storage parameter and the default configuration string.

```
##DEFAULTS

A_INDEX_ROWID           ""
A_INDEX_SHAPE     ""
A_INDEX_STATEID         ""
A_INDEX_USER            ""
#A_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE> LONG IN <TABLESPACE>"
B_INDEX_ROWID           ""
B_INDEX_SHAPE     ""
B_INDEX_USER            ""
B_RUNSTATS     "YES"
#B_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE> LONG IN <TABLESPACE>"
#BLK_STORAGE "IN <TABLESPACE> INDEX IN <TABLESPACE> LONG IN <TABLESPACE>"
#AUX_STORAGE "IN <TABLESPACE> INDEX IN <TABLESPACE>"
#BND_STORAGE "IN <TABLESPACE> INDEX IN <TABLESPACE>"
#RAS_STORAGE "IN <TABLESPACE> INDEX IN <TABLESPACE>"
D_INDEX_DELETED_AT""
D_INDEX_STATE_ROWID             ""
#D_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE> LONG IN <TABLESPACE>"
BLOB_OPTION             "LOGGED NOT COMPACT"
BLOB_SIZE       "1M"
CLOB_OPTION             "LOGGED NOT COMPACT"
CLOB_SIZE       "32K"
#MAX_CACHED_CURSORS "80"
UI_TEXT         ""
#XML_DOC_STORAGE        "IN <TABLESPACE> INDEX IN <TABLESPACE>"
#XML_IDX_STORAGE        "IN <TABLESPACE> INDEX IN <TABLESPACE>"
XML_IDX_INDEX_ID     ""
XML_IDX_INDEX_TAG     ""
XML_IDX_INDEX_DOUBLE  ""
XML_IDX_FULLTEXT_UPD_FREQUENCY   ""
XML_IDX_FULLTEXT_UPD_MINIMUM     ""
XML_IDX_FULLTEXT_UPD_COMMIT      ""
XML_IDX_FULLTEXT_IDXDIRECTORY    ""
XML_IDX_FULLTEXT_WKDIRECTORY     ""
XML_IDX_FULLTEXT_LANGUAGE       ""
XML_IDX_FULLTEXT_CCSID          ""
END
```

Setting the default BLOB size

DB2 requires a size on BLOB column creation.

If a BLOB column is to be created and it has a size of greater than 2 GB, this size will be ignored and the default LOB_SIZE parameter of 1 MB will be used. This will allow the DBA to carefully craft the database parameters.

Setting the default CLOB size

DB2 requires a size on CLOB column creation.

If a CLOB column is to be created and it has a size of greater than 2 GB, the size will be ignored and the default CLOB_SIZE parameter of 32K will be used. This will allow the DBA to carefully craft the database parameters.

Setting the B_RUNSTATS parameter

This parameter will be used at the end of a data load, after all the records are inserted and the layer is being readied to put into normal_io mode. The last part of switching to normal_io mode will be the checking of B_RUNSTATS. "YES" will be the default if no B_RUNSTATS parameter is present in the DEFAULTS keyword of the dbtune.sde file. B_RUNSTATS only applies to the business table If B_RUNSTATS is equal to "YES" or "yes", then a full runstats will be performed on the table automatically. If it is set to anything else, then a runstats will not happen. The vast majority of users will want to have the full runstats done on the table. For those who wish to do something special with it for some reason, such as only do indexes, they can set B_RUNSTATS to "NO" and perform a manual RUNSTATS command with any options they choose.

Setting the MAX_CACHED_CURSORS parameter

Some control should be available over how many cursors per user can be allocated to the cache. While there are database tuning parameters related to the maximum number of cursors (SQL_MAX_CONCURRENT_ACTIVITIES for DB2), these are of limited use or are often are not set - effectively limited only by available resources and complexity of query. Simply applying the default max cursor value may cause issues on heavily loaded systems. To better control this, or to disable caching entirely the dbtune parameter MAX_CACHED_CURSORS was added as a DEFAULTS keyword. The current default value is "80". To disable caching, set it to "0".

Setting the system table DATA_DICTIONARY keyword

During the execution of sdesetupdb2 the ArcSDE and geodatabase system tables and indexes are created with the storage parameters of the DATA_DICTIONARY keyword. You may customize the keyword in the dbtune.sde file prior to running the sdesetupdb2

tool. In this way you can change default storage parameters of the DATA_DICTIONARY keyword.

Edits to all of the geodatabase system tables and most of the ArcSDE system tables occur when schema change occurs. As such, edits to these system tables and indexes usually happen during the initial creation of an ArcGIS database with infrequent modifications occurring whenever a new schema object is added.

Four of the ArcSDE system tables—VERSION, STATES, STATE_LINEAGES, and MVTABLES_MODIFIED—participate in the ArcSDE versioning model and record events resulting from changes made to multiversioned tables. If your site makes extensive use of a multiversioned database, these tables and their associated indexes are quite active. Separating these objects into their own tablespace allows you to position their data files with data files that experience low I/O activity and thus minimize disk I/O contention.

If the dbtune.sde file does not contain the DATA_DICTIONARY keyword, or if any of the required parameters are missing from the keyword, the following records will be inserted into the DATA_DICTIONARY when the table is created. Note that the DBTUNE file entries are provided here for readability.

```
##DATA_DICTIONARY

B_INDEX_ROWID             ""
B_INDEX_USER              ""
#B_STORAGE      "IN <TABLESPACE> INDEX IN <TABLESPACE> LONG IN <TABLESPACE>"
#STATES_TABLE    "IN <TABLESPACE> INDEX IN <TABLESPACE>"
STATES_INDEX     ""
#STATE_LINEAGES_TABLE    "IN <TABLESPACE> INDEX IN <TABLESPACE>"
#VERSIONS_TABLE         "IN <TABLESPACE> INDEX IN <TABLESPACE>"
VERSIONS_INDEX          ""
#MVTABLES_MODIFIED_TABLE "IN <TABLESPACE> INDEX IN <TABLESPACE>"
MVTABLES_MODIFIED_INDEX  ""
#XML_INDEX_TAGS_TABLE    "IN <TABLESPACE> INDEX IN <TABLESPACE>"
#XML_TAGS_TABLE         "IN <TABLESPACE> INDEX IN <TABLESPACE>"
END
```

The SURVEY MULTIBINARY keyword

This keyword is used to support BLOB columns on the SDB_<n>_Surveys table. However, it is mainly meant for Oracle since it cannot have multiple LONG RAW columns in the same business table.

```
##SURVEY_MULTI_BINARY

UI_TEXT ""
END
```

The TOPOLOGY keyword

The TOPOLOGY keyword controls the storage of topology tables, which are named POINTERRORS, LINEERRORS, POLYERRORS and DIRTYAREAS. An SDE instance must have a valid topology keyword in the dbtune table, or topology will not be built.

The DIRTYAREAS table maintains information on areas within a layer that have been changed. Because it tracks versions, data will be inserted or updated but not deleted during normal use. The DIRTYAREAS table will reduce in size only when database versions are compressed.

Because the DIRTYAREAS table is much more active than the remaining topology tables, the TOPOLOGY keyword may be compound. You may specify the DIRTYAREAS suffix to list configuration string to be used to create the topology tables.

For DB2, the default values for TOPOLOGY and TOPOLOGY::DIRTYAREAS are

```
##TOPOLOGY_DEFAULTS

UI_TOPOLOGY_TEXT "The topology default configuration"
A_INDEX_ROWID   ""
A_INDEX_SHAPE   ""
A_INDEX_STATEID ""
A_INDEX_USER    ""
#A_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
B_INDEX_ROWID   ""
B_INDEX_SHAPE   ""
B_INDEX_USER    ""
#B_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
D_INDEX_DELETED_AT  ""
D_INDEX_STATE_ROWID ""
#D_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
END

##TOPOLOGY_DEFAULTS::DIRTYAREAS
A_INDEX_ROWID   ""
```

```
A_INDEX_SHAPE   ""
A_INDEX_STATEID ""
A_INDEX_USER    ""
#A_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
B_INDEX_ROWID   ""
B_INDEX_SHAPE   ""
B_INDEX_USER    ""
#B_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
D_INDEX_DELETED_AT  ""
D_INDEX_STATE_ROWID ""
#D_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
END
```

The IMS METADATA keywords

The IMS METADATA keywords control the storage of the IMS Metadata tables. These keywords are a standard part of the dbtune table. If the IMS_METADATA storage parameters are not present in the dbtune file when it is imported into the DBTUNE table, ArcSDE applies software defaults.

The software defaults have the same settings as the keyword parameters listed in the dbtune.sde table that is shipped with ArcSDE. You will need to edit the storage parameters tablespace names. As always try to separate the tables and indexes into different tablespaces.

For more information about installing IMS Metadata and the associated tables and indexes refer to ArcIMS Metadata Server documentation.

The IMS_METADATA keyword controls the storage of the ims_metadata feature class. Four indexes are created on the ims_metadata business table. ArcSDE creates the following default IMS_METADATA keyword in the DBTUNE table if the keyword is missing from the dbtune file when it is imported.

The IMS metadata keywords are as follows:

```
##IMS_METADATA

B_INDEX_ROWID   ""
B_INDEX_SHAPE   ""
B_INDEX_USER    ""
#B_STORAGE      "IN <TABLESPACE> INDEX IN <TABLESPACE> LONG IN <TABLESPACE>"
BLOB_OPTION     "LOGGED NOT COMPACT"
BLOB_SIZE       "1M"
COMMENT         "The IMS metadata feature class"
UI_TEXT         ""
END
```

The IMS_METADATARELATIONSHIPS keyword controls the storage of the ims_metadatarelationships business table. Three indexes are created on the

ims_metadatarelationships business table. ArcSDE creates the following default IMS_METADATARELATIONSHIPS keyword in the DBTUNE table if the keyword is missing from the dbtune file when it is imported.

```
##IMS_METADATARELATIONSHIPS

B_INDEX_ROWID   ""
B_INDEX_USER    ""
#B_STORAGE      "IN <TABLESPACE> INDEX IN <TABLESPACE>"
END
```

The IMS_METADATATAGS keyword controls the storage of the ims_metadatatags business table. Two indexes are created on the ims_metadatatags business table. ArcSDE creates the following default IMS_METADATATAGS keyword in the DBTUNE table if the keyword is missing from the dbtune file when it is imported.

```
##IMS_METADATATAGS

B_INDEX_ROWID   ""
B_INDEX_USER    ""
#B_STORAGE      "IN <TABLESPACE> INDEX IN <TABLESPACE>"
END
```

The IMS_METADATATHUMBNAILS keyword controls the storage of the ims_metadatathumbnails business table. One index is created on the ims_metadatathumbnails business table. ArcSDE creates the following default IMS_METADATATHUMBNAILS keyword in the DBTUNE table if the keyword is missing from the dbtune file when it is imported.

```
##IMS_METADATATHUMBNAILS

B_INDEX_USER    ""
#B_STORAGE      "IN <TABLESPACE> INDEX IN <TABLESPACE> LONG IN <TABLESPACE>"
BLOB_OPTION     "LOGGED NOT COMPACT"
BLOB_SIZE       "1M"
END
```

The IMS_METADATAUSERS keyword controls storage of the ims_metadatausers business table. One index is created on the ims_metadatausers business table. ArcSDE creates the following default IMS_METADATAUSERS keyword in the DBTUNE table if the keyword is missing from the dbtune file when it is imported.

```
##IMS_METADATAUSERS

B_INDEX_ROWID   ""
B_INDEX_USER    ""
#B_STORAGE      "IN <TABLESPACE> INDEX IN <TABLESPACE>"
END
```

The IMS_METADATAVALUES keyword controls the storage of the ims_metadatavalues business table. Two indexes are created on ims_metadatavalues business table. ArcSDE creates the following default IMS_METADATAVALUES keyword in the DBTUNE table if the keyword is missing from the dbtune file when it is imported.

```
##IMS_METADATAVALUES

B_INDEX_USER    ""
#B_STORAGE      "IN <TABLESPACE> INDEX IN <TABLESPACE>"
END
```

The IMS_METADATAWORDINDEX keyword controls the storage of the ims_metadatawordindex business table. Three indexes are created on the ims_metadatawordindex business table. ArcSDE creates the following default IMS_METADATAWORDINDEX keyword in the DBTUNE table if the keyword is missing from the dbtune file when it is imported.

```
##IMS_METADATAWORDINDEX
B_INDEX_USER    ""
#B_STORAGE      "IN <TABLESPACE> INDEX IN <TABLESPACE>"
END
```

The IMS_METADATAWORDS keyword controls the storage of the ims_metadatawords business table. One index is created on the ims_metadatawords business table. ArcSDE creates the following default IMS_METADATAWORDS keyword in the DBTUNE table if the keyword is missing from the dbtune file when it is imported.

```
##IMS_METADATAWORDS
B_INDEX_ROWID   ""
B_INDEX_USER    ""
#B_STORAGE      "IN <TABLESPACE> INDEX IN <TABLESPACE>"
END
```

Changing the appearance of DBTUNE keywords in the ArcInfo user interface

ArcSDE provides UI_TEXT and UI_NETWORK_TEXT storage parameters that allow you to change the appearance of the configuration keywords in the ArcGIS® user interface.

ArcSDE administrators can add one of these storage parameters to each keyword to communicate to the ArcInfo™ schema builders the intended use of the keyword. The configuration string of these storage parameters will appear in ArcInfo interface DBTUNE keyword scrolling lists.

The UI_TEXT storage parameter should be added to keywords that will be used to build tables, feature classes, and indexes.

The UI_NETWORK_TEXT storage parameter should be added to parent network keywords.

Adding a comment to a keyword

The COMMENT storage parameter allows you to add informative text that describes such things as a keyword's intended use, the last time it was changed, or who created it.

LOGFILE configuration keywords

Log files are used by ArcSDE to maintain temporary and persistent sets of selected records.

The LOGFILE_DEFAULTS keyword holds the parameter group for all users that do not have their own keyword created. Alternatively, you may create individual log file keywords for specific users by appending the user's name to the LOGFILE_ prefix to form the keyword name. For example, if the user's name is STANLEY, ArcSDE will search the DBTUNE table for the LOGFILE_STANLEY configuration keyword. If this configuration keyword is not found, ArcSDE will use the storage parameters of the LOGFILE_DEFAULTS configuration keyword.

ArcSDE always creates the DBTUNE table with a LOGFILE_DEFAULTS configuration keyword. If you do not specify this configuration keyword in a DBTUNE file imported by the sdedbtune command, ArcSDE will populate the DBTUNE table with software default LOGFILE_DEFAULTS storage parameters. Further, if the DBTUNE file lacks some of the LOGFILE_DEFAULTS configuration keyword storage parameters, ArcSDE supplies the rest. Therefore, the LOGFILE_DEFAULTS configuration keyword is always fully populated.

If a user-specific configuration keyword exists but some of the storage parameters are not present, the storage parameters of the LOGFILE_DEFAULTS configuration keyword are used.

The storage parameters that are used depend on which type of log files the server has been configured to use. If the ArcSDE server is configured to use shared log files, ArcSDE creates the log file tables SDE_LOGFILES and SDE_LOGFILE_DATA and indexes the first time the user connects.

For the creation of shared log file tables the LD_STORAGE and LF_STORAGE parameters control the storage of the SDE_LOGFILE_DATA and SDE_LOGFILES tables.

The LF_INDEXES parameter defines the storage of the indexes of the SDE_LOGFILES table, while the LD_INDEX_DATA_ID and LD_INDEX_ROWID parameters define the storage of the SDE_LOGFILE_DATA table.

Creating a log file configuration keyword for each user allows you to position the SDE user's log files on separate devices by specifying the tablespace the log file tables and indexes are created in. Most installations of ArcSDE will function well using the LOGFILE_DEFAULTS storage parameters supplied with the installed dbtune.sde file. However, for applications making use of SDE log files, such as ArcGIS Desktop, it may help performance to spread the log files across the file system. Typically logfiles are updated whenever a selection set exceeds 100 records.

If you have configured the server to use session based or stand-alone logfiles in addition to shared logfiles, ArcSDE will use a different set of storage parameters when it creates the session-based and stand-alone logfiles tables.

The SESSION_STORAGE parameter defines the storage of the session-based and stand-alone logfile tables which include both session and standalone types.

The SESSION_INDEX parameter defines the storage of the session-based and stand-alone logfile table indexes.

If the imported DBTUNE file does not contain a LOGFILE_DEFAULTS configuration keyword or if any of the logfile storage parameters are missing, ArcSDE will insert the following records:

```
##LOGFILE_DEFAULTS

LD_INDEX_DATA_ID    ""
LD_INDEX_ROWID          ""
#LD_STORAGE             "IN <TABLESPACE> INDEX IN <TABLESPACE>"
#LF_STORAGE             "IN <TABLESPACE> INDEX IN <TABLESPACE>"
UI_TEXT             "LOGFILES"
#SESSION_STORAGE    "IN <TABLESPACE> INDEX IN <TABLESPACE>"
#SESSION_INDEX      "IN <TABLESPACE> INDEX IN <TABLESPACE>"
SESSION_TEMP_TABLE  "0"
END
```

Network class composite keywords

The composite keyword is a unique type of keyword designed to accommodate the tables of the ArcGIS network class. The network table's size variation requires a

keyword that provides configuration storage parameters for both large and small tables. Typically the network descriptions table is very large in comparison with the others.

To accommodate the vast difference in the size of the network tables, the network composite keyword is subdivided into elements. A network composite keyword has three elements: the parent element defines the general characteristic of the keyword and the junctions feature class, the description element defines the configuration of the DESCRIPTIONS table and its indexes, and the network element defines the configuration of the remaining network tables and their indexes.

The parent element does not have a suffix, and its keyword looks like any other keyword. The description element is demarcated by the addition of the ::DESC suffix to the parent element's keyword, and the network element is demarcated by addition of the ::NETWORK suffix to the parent element's keyword.

For example, if the parent element keyword is ELECTRIC, the network composite keyword would appear in a DBTUNE file as follows:

```
##ELECTRIC

COMMENT       This keyword is dedicated to the electrical geometric network class

UI_NETWORK_TEXT       "The electrical geometrical network class keyword"

B_STORAGE    "IN SDEDB2 INDEX IN SDEINDEX LONG IN SDELOBS"

B_INDEX_ROWID          "PCTFREE 15 DISALLOW REVERSE SCANS"

B_INDEX_USER           "PCTFREE 10 MINPCTUSED 25 DISALLOW REVERSE SCANS"

A_STORAGE              "IN SDEDB2 INDEX IN SDEINDEX LONG IN SDELOBS"

A_INDEX_ROWID          "PCTFREE 15 DISALLOW REVERSE SCANS"

A_INDEX_USER           "PCTFREE 10 MINPCTUSED 25 DISALLOW REVERSE SCANS"

A_INDEX_STATEID        ""

D_STORAGE              "IN SDEDB2 INDEX IN SDEINDEX LONG IN SDELOBS"

D_INDEX_DELETED_AT  ""

D_INDEX_STATE_ROWID  ""

END


##ELECTRIC::DESC

B_STORAGE              "IN SDEDB2 INDEX IN SDEINDEX LONG IN SDELOBS"
```

```
B_INDEX_ROWID          "PCTFREE 15 DISALLOW REVERSE SCANS"

B_INDEX_USER           "PCTFREE 10 MINPCTUSED 25 DISALLOW REVERSE SCANS"

A_STORAGE              "IN SDEDB2 INDEX IN SDEINDEX LONG IN SDELOBS"

A_INDEX_ROWID          "PCTFREE 15 DISALLOW REVERSE SCANS"

A_INDEX_USER           "PCTFREE 10 MINPCTUSED 25 DISALLOW REVERSE SCANS"

A_INDEX_STATEID        ""

D_STORAGE              "IN SDEDB2 INDEX IN SDEINDEX LONG IN SDELOBS"

D_INDEX_DELETED_AT     ""

D_INDEX_STATE_ROWID    ""

END

##ELECTRIC::NETWORK

B_STORAGE              "IN SDEDB2 INDEX IN SDEINDEX LONG IN SDELOBS"

B_INDEX_ROWID          "PCTFREE 15 DISALLOW REVERSE SCANS"

B_INDEX_USER           "PCTFREE 10 MINPCTUSED 25 DISALLOW REVERSE SCANS"

A_STORAGE              "IN SDEDB2 INDEX IN SDEINDEX LONG IN SDELOBS"

A_INDEX_ROWID          "PCTFREE 15 DISALLOW REVERSE SCANS"

A_INDEX_USER           "PCTFREE 10 MINPCTUSED 25 DISALLOW REVERSE SCANS"

A_INDEX_STATEID        ""

D_STORAGE              "IN SDEDB2 INDEX IN SDEINDEX LONG IN SDELOBS"

D_INDEX_DELETED_AT     ""

D_INDEX_STATE_ROWID    ""

END
```

Following the import of the DBTUNE file, these records would be inserted into the DBTUNE table.

```
DB2> select keyword, parameter_name from DBTUNE;

KEYWORD             PARAMETER_NAME
----------------    ----------------
ELECTRIC            COMMENT
ELECTRIC            UI_NETWORK_TEXT
ELECTRIC            B_STORAGE
ELECTRIC            B_INDEX_ROWID
```

```
ELECTRIC            B_INDEX_SHAPE
ELECTRIC            B_INDEX_USER
ELECTRIC            A_STORAGE
ELECTRIC            A_INDEX_ROWID
ELECTRIC            A_INDEX_SHAPE
ELECTRIC            A_INDEX_USER
ELECTRIC            A_INDEX_STATEID
ELECTRIC            D_STORAGE
ELECTRIC            D_INDEX_DELETED_AT
ELECTRIC            D_INDEX_STATE_ROWID
ELECTRIC::DESC      B_STORAGE
ELECTRIC::DESC      B_INDEX_ROWID
ELECTRIC::DESC      B_INDEX_USER
ELECTRIC::DESC      A_STORAGE
ELECTRIC::DESC      A_INDEX_ROWID
ELECTRIC::DESC      A_INDEX_STATEID
ELECTRIC::DESC      A_INDEX_USER
ELECTRIC::DESC      D_STORAGE
ELECTRIC::DESC      D_INDEX_DELETE_AT
ELECTRIC::DESC      D_INDEX_STATE_ROWID
ELECTRIC::NETWORK   B_STORAGE
ELECTRIC::NETWORK   B_INDEX_ROWID
ELECTRIC::NETWORK   B_INDEX_USER
ELECTRIC::NETWORK   A_STORAGE
ELECTRIC::NETWORK   A_INDEX_ROWID
ELECTRIC::NETWORK   A_INDEX_STATEID
ELECTRIC::NETWORK   A_INDEX_USER
ELECTRIC::NETWORK   D_STORAGE
ELECTRIC::NETWORK   D_INDEX_DELETE_AT
ELECTRIC::NETWORK   D_INDEX_STATE_ROWID
```

The network junctions feature class is created with the ELECTRIC configuration keyword storage parameters, the network descriptions table is created with the storage parameters of the ELECTRIC::DESC keyword, and the remaining smaller network tables are created with the ELECTRIC::NETWORK keyword.

The NETWORK_DEFAULTS keyword

The NETWORK_DEFAULTS keyword contains the default storage parameters for the ArcGIS network class. If the user does not select a network class composite keyword from the ArcCatalog interface, the ArcGIS network is created with the storage parameters within the NETWORK_DEFAULTS keyword.

Whenever a network class composite keyword is selected, its storage parameters are used to create the feature class, table, and indexes of the network class. If a network composite keyword is missing any storage parameters, ArcGIS substitutes the storage parameters of the DEFAULTS keyword rather than the NETWORK_DEFAULTS keyword. The storage parameters of the NETWORK_DEFAULTS keyword are used when a network composite keyword has not been specified.

If a NETWORK_DEFAULTS keyword is not present in a DBTUNE file imported into the DBTUNE table, the following NETWORK_DEFAULTS keyword is created.

##NETWORK_DEFAULTS

```
A_INDEX_ROWID          ""
A_INDEX_SHAPE   ""
A_INDEX_STATEID        ""
A_INDEX_USER           ""
#A_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
B_INDEX_ROWID          ""
B_INDEX_SHAPE   ""
B_INDEX_USER           ""
#B_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
COMMENT      "The base system initialization parameters for NETWORK_DEFAULTS"
D_INDEX_DELETED_AT""
D_INDEX_STATE_ROWID            ""
#D_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
UI_NETWORK_TEXT      "The network default configuration"
END

##NETWORK_DEFAULTS::DESC
A_INDEX_ROWID          ""
A_INDEX_STATEID        ""
A_INDEX_USER           ""
#A_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
B_INDEX_ROWID          ""
B_INDEX_USER           ""
#B_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
D_INDEX_DELETED_AT""
D_INDEX_STATE_ROWID            ""
#D_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
END

##NETWORK_DEFAULTS::NETWORK
A_INDEX_ROWID          ""
A_INDEX_STATEID        ""
A_INDEX_USER           ""
#A_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
B_INDEX_ROWID          ""
B_INDEX_USER           ""
#B_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
D_INDEX_DELETED_AT""
D_INDEX_STATE_ROWID            ""
#D_STORAGE  "IN <TABLESPACE> INDEX IN <TABLESPACE>"
END
```

DB2 default parameters

By default, DB2 stores tables and indexes in the user's default tablespace using the tablespace's default storage parameters. This tablespace is called USERSPACE1 in DB2.

The complete list of ArcSDE storage parameters

Parameter Name	Value	Parameter Description	Default Value
STATES_LINEAGES_TABLE	<string>	State_lineages table	B_STORAGE
STATES_TABLE	<string>	States table	B_STORAGE
STATES_INDEX	<string>	States indexes	B_INDEX_USER
MVTABLES_MODIFIED_TABLE	<string>	Mvtables_modified table	B_STORAGE
MVTABLES_MODIFIED_INDEX	<string>	Mvtables_modified index	B_INDEX_USER
VERSIONS_TABLE	<string>	Versions table	B_STORAGE
VERSIONS_INDEX	<string>	Version index	B_INDEX_USER
B_STORAGE	<string>	Business table	DB2 defaults
B_INDEX_ROWID	<string>	Business table object ID column index	DB2 defaults
B_INDEX_SHAPE	<string>	Business table spatial column index	DB2 defaults
B_INDEX_USER	<string>	Business table user index(s)	DB2 defaults
B_RUNSTATS	<string>	Default value for RUNSTATS	YES
A_STORAGE	<string>	Adds table	DB2 defaults
A_INDEX_ROWID	<string>	Adds table object ID column index	DB2 defaults
A_INDEX_STATEID	<string>	Adds table sde_state_id column index	DB2 defaults
A_INDEX_USER	<string>	Adds table index	DB2 defaults
A_INDEX_SHAPE	<string>	Adds table spatial column index	DB2 defaults

Parameter Name	Value	Parameter Description	Default Value
D_STORAGE	<string>	Deletes table	DB2 defaults
D_INDEX_ STATE_ROWID	<string>	Deletes table sde_states_id and sde_deletes_row_id column index	DB2 defaults
D_INDEX_DELETED_AT	<string>	Deletes table sde_deleted_at column index	DB2 defaults
BLOB_SIZE	<string>	Size of BLOB column	1 MB
BLOB_OPTION	<string>	Storage configuration properties of the BLOB column	DB2 defaults
CLOB_SIZE	<string>	Size of CLOB column	32 K
CLOB_OPTION	<string>	Storage configuration properties of the CLOB column	DB2 defaults
LF_STORAGE	<string>	Sde_logfiles table	DB2 defaults
LF_INDEXES	<string>	Sde_logfile table column indexes	DB2 defaults
LD_STORAGE	<string>	Sde_logfile_data table	DB2 defaults
LD_INDEX_DATA_ID	<string>	Sde_logfile_data table	DB2 defaults
LD_INDEX_ROWID	<string>	Sde_logfile_data table sde_row_id column index	DB2 defaults
SESSION_STORAGE	<string>	SDE session-based and stand alone log file tables	DB2 defaults
SESSION_INDEX	<string>	Sde session-based and stand-alone log file indexes	DB2 defaults

Parameter Name	Value	Parameter Description	Default Value
RAS_STORAGE	<string>	Raster RAS table	DB2 defaults
BND_STORAGE	<string>	Raster BND table	DB2 defaults
AUX_STORAGE	<string>	Raster AUX table	DB2 defaults
BLK_STORAGE	<string>	Raster BLK table	DB2 defaults
UI_TEXT	<string>	User interface name of the configuration keyword	DB2 defaults
MX_CACHED_CURSORS	<string>	Maximum number of cached cursors	80
UI_NETWORK_TEXT	<string>	User interface name of the network configuration keyword	DB2 defaults
COMMENT	<string>	Comments	none
XML_IDX_INDEX_ID	<string>	Index storage info	DB2 defaults
XML_IDX_INDEX_TAG	<string>	Index storage info	DB2 defaults
XML_IDX_INDEX_DOUBLE	<string>	Index storage info	DB2 defaults
XML_IDX_FULLTEXT_UPD_FREQUENCY	<string>	Index update frequency	DB2 defaults
XML_IDX_FULLTEXT_UPD_MINIMUM	<string>	Index update minimum rows	DB2 defaults
XML_IDX_FULLTEXT_UPD_COMMIT	<string>	Index update commit_count	DB2 defaults
XML_IDX_FULLTEXT_UPD_IDXDIRECTORY	<string>	Path to text index directory	DB2 defaults
XML_IDX_FULLTEXT_UPD_WKDIRECTORY	<string>	Path to text index working directory	DB2 defaults

Parameter Name	Value	Parameter Description	Default Value
XML_IDX_FULLTEXT_UPD_LANGUAGE	<string>	Text index language	DB2 defaults
XML_IDX_FULLTEXT_UPD_CCSID	<string>	Text index CCSID	DB2 defaults
XML_INDEX_TAGS_TABLE	<string>	Index tags table (data dictionary)	DB2 defaults
XML_TAGS_TABLE	<string>	Tags table (data dictionary)	DB2 defaults

CHAPTER 4

Managing tables, feature classes, and raster columns

A fundamental part of any database is creating and loading the tables. Tables with spatial columns are called standalone feature classes. Attribute-only (nonspatial) tables are also an important part of any database. This chapter will describe the table and feature class creation and loading process.

Data creation

There are numerous applications that can create and load data within an ArcSDE DB2 database. These include:

1. ArcSDE administration commands located in the bin directory of SDEHOME:

   - sdelayer—Creates and manages feature classes.
   - sdetable—Creates and manages tables.
   - sdeimport—Takes an existing sdeexport file and loads the data into a feature class.
   - shp2sde—Loads an ESRI shapefile into a feature class.
   - cov2sde—Loads a coverage, Map LIBRARIAN layer, or ArcStorm™ layer into a feature class.
   - tbl2sde—Loads an attribute-only dBASE® or INFO™ file into a table.

- sdegroup—A specialty feature class creation command that combines the features of an existing feature class into single multipart features and stores them in a new feature class for background display. The generated feature class is used for rapid display of a large amount of geometry data. The attribute information is not retained, and spatial searches cannot be performed on these feature classes.

- sderaster—creates, inserts, modifies, imports, and manages rasters stored in an ArcSDE database.

These are all run from the operating system prompt. Command references for these tools are in the ArcSDE Developer Help.

Other applications include:

2. ArcGIS Desktop—Use ArcCatalog or ArcToolbox to manage and populate your database.

3. ArcInfo Workstation—Use the Defined Layer interface to create and populate the database.

4. ArcView® 3.2—Use the Database Access extension.

5. MapObjects®—Custom Component Object Model (COM) applications can be built to create and populate databases.

6. ArcSDE CAD Client extension—For AutoCAD® and MicroStation® users.

7. Other third party applications built with either the C or Java™ APIs.

This document focuses primarily on the ArcSDE administration tools but does provide some ArcGIS Desktop examples as well. In general, most people prefer an easy-to-use graphical user interface like the one found in ArcGIS Desktop. For details on how to use ArcCatalog or ArcToolbox (another desktop data loading tool), please refer to the ArcGIS books:

- *Using ArcCatalog*

- *Building a Geodatabase*

Creating and populating a feature class

The general process involved with creating and loading a feature class is to:

1. Create the business table.
2. Record the business table and the spatial column in the ArcSDE LAYERS and GEOMETRY_COLUMNS system tables, thus adding a new feature class to the database.
3. Switch the feature class to load_only_io mode, an optional step to improve bulk data loading performance. It is OK to leave feature class in normal_io mode to load data.
4. Insert the records (load data).
5. Switch the feature class to normal_io mode which builds the indexes.
6. Version the data (optional).
7. Grant privileges on the data (optional).

In the following sections, this process is discussed in more detail and illustrated with some examples of ArcSDE administration commands usage and ArcInfo data loading utilities through the ArcCatalog and ArcToolbox interfaces.

Creating a feature class "from scratch"

There are two basic ways to create a feature class. You can create a feature class from scratch (requiring considerably more effort), or you can create a feature class from existing data such as a coverage or ESRI shapefile. Both methods are reviewed below with the "from scratch" method presented first.

Creating a business table

You may create a business table with either the SQL CREATE TABLE statement or the ArcSDE sdetable command. The sdetable command allows you to include a dbtune configuration keyword containing the storage parameters of the table.

Although the table may contain up to 1012 columns, ArcSDE requires that only one of those columns be defined as a spatial column.

In this example, the sdetable command is used to create the roads business table.

```
sdetable -o create -t roads -d 'road_id integer, name string(32), shape
integer' -k roads -u beetle -p bug
```

The table is created using the dbtune configuration keyword (-k) "roads" by the user "beetle".

The same table could be created with a SQL CREATE TABLE statement using the DB2 SQL interface.

```
create table roads
(road_id     integer,
name         varchar(32),
shape        integer);
```

At this point you have created a table in the database. ArcSDE does not yet recognize it as a feature class. The next step is to record the spatial column in the ArcSDE LAYERS and GEOMETRY_COLUMNS system tables and thus add a new feature class to the database.

Adding a feature class

After creating a business table, you must add an entry for the spatial column in the ArcSDE LAYERS system tables before the ArcSDE server can reference it. Use the sdelayer command with the "-o add" operation to add the new feature class.

In the following example, the roads feature class is added to the ArcSDE database. Note that to add the feature class, the roads table name and the spatial column are combined to form a unique feature class reference. To understand the purpose of the –e, –g, and –x options, refer to the sdelayer command reference in the *ArcSDE Developer Help* system.

```
sdelayer -o add -l roads,shape -e l+ -g 256,0,0 -x 0,0,100 -u beetle -p
bug -k roads
```

The feature class tables and indexes are stored according to the storage parameters of the "roads" configuration keywords in the DBTUNE table. Upon successful completion of the previous sdetable command—to create a table—and the sdelayer command—to record the feature class in the ArcSDE system tables—you have an empty feature class in normal_io mode.

Switching to load-only mode

Switching the feature class to load-only mode drops the spatial index and makes the feature class unavailable to ArcSDE clients. Bulk loading data into the feature class in this state is much faster due to the absence of index maintenance. Use the sdelayer command to switch the feature class to load-only mode by specifying the "-o load_only_io" operation.

```
sdelayer -o load_only_io -l roads,shape -u beetle -p bug
```

Note: A feature class registered as multiversioned cannot be placed in the load-only I/O mode.

Inserting records into the feature class

Once the empty feature class exists, the next step is to populate it with data. There are several ways to insert data into a feature class, but probably the easiest method is to

convert an existing shapefile or coverage or import a previously exported ArcSDE sdeexport file directly into the feature class.

In this first example, shp2sde is used with the init operation. The init operation is used on newly created feature classes or can be used on feature classes when you want to "overwrite" data that's already there. Don't use the init operation on feature classes that already contain data unless you want to remove the existing data. Here, the shapefile, "rdshp", will be loaded into the feature class, "roads". Note that the name of the spatial column ("shape" in this case) is included in the feature class (-l) option.

```
shp2sde -o init -l roads,shape -f rdshp -u beetle -p bug
```

Similarly, you can also use the cov2sde command:

```
cov2sde -o init -l roads,shape -f rdcov -u beetle -p bug
```

Switching the table to normal I/O mode

After data has been loaded into the feature class, you must switch the feature class to normal_io mode to re-create all indexes and make the feature class available to clients. For example:

```
sdelayer -o normal_io -l roads,shape -u beetle -p bug
```

Versioning your data

Optionally, you may enable your feature class as multiversioned. Versioning is a process that allows multiple representations of your data to exist without requiring duplication or copies of the data. ArcMap requires data to be multiversioned to edit it. For further information on versioning data, refer to the *Building a Geodatabase* book.

In this example, the feature class, states will be registered as multiversioned using the sdetable alter_reg operation.

```
sdetable -o alter_reg -t states -c ver_id -C SDE -V multi -k
GEOMETRY_TYPE
```

Granting privileges on the data

Once you have the data loaded, it is often necessary for other users to have access to the data for update, query, insert, or delete operations. Initially, only the user who has created the business table has access to it. In order to make the data available to others, the owner of the data must grant privileges to other users. The owner can use the sdelayer command to grant privileges. Privileges can be granted to either another user or to a group.

In this example, a user called "beetle" gives a user called "spider" SELECT privileges on a feature class called "states".

```
sdelayer -o grant -l states,feature -U spider -A SELECT -u beetle -p bug
```

The full list of -A keywords are:

SELECT The user may query the selected object data.

DELETE The user may delete the selected object data.

UPDATE The user may modify the selected object data.

INSERT The user may add new data to the selected object data.

If you include the -I grant option, you also grant the recipient the privilege of granting other users and groups the initial privilege.

In this example, a user, "beetle" gives a GROUP, "arcsde" SELECT privileges on table "RIVERS".

sdetable -o grant –t RIVERS -U group:arcsde -A SELECT -u beetle -p bug

To distinguish a GROUP from a USER the prefix "group:" is attached to the –U option.

Creating and loading feature classes from existing data

The "from scratch" method of creating a schema and loading it has been reviewed. This next section reviews how to create feature classes from existing data. This method is simpler since the creation and load process is completed at once.

Each of the ArcSDE administration commands, shp2sde, cov2sde, and sdeimport, includes a "-o create" operation, which allows you to create a new feature class within the ArcSDE database. The create operation does all of the following:

- Creates the business table using the input data as the template for the schema
- Adds the feature class to the ArcSDE system tables
- Puts the feature class into load-only mode
- Inserts data into the feature class
- When all the records are inserted, puts the feature class into normal_io mode

shp2sde

The shp2sde command converts shapefiles into ArcSDE feature classes. The spatial column definition is read directly from the shapefile. You can use the shpinfo command to display the shapefile column definitions.

In this example, the -k option references the DBTUNE configuration keyword ROADS. The ROADS keyword contains storage parameters for storing the tables and indexes of the roads feature class.

```
shp2sde -o create -f rdshp -l roads,shape -k ROADS -u beetle -p bug
```

cov2sde

The cov2sde command converts ArcInfo coverages, ArcInfo Librarian™ library feature classes, and ArcStorm library feature classes into ArcSDE feature classes. The create operation derives the spatial column definition from the coverage's feature attribute table. Use the ArcInfo describe command to display the ArcInfo data source column definitions.

In this example, an ArcStorm library, "roadlib", is converted into the feature class, "roads".

```
cov2sde -o create -l roads,shape -f roadlib,arcstorm -g 256,0,0 -x
0,0,100 -e 1+ -u beetle -p bug
```

sdeimport

The sdeimport command converts ArcSDE export files into ArcSDE feature classes. In this example, the "roadexp" ArcSDE export file is converted into the feature class 'roads'.

```
sdeimport -o create -l roads,shape -f roadexp -u beetle -p bug
```

After using these commands to create and load data, you may optionally need to enable multiversioning on the feature class and grant privileges on the feature class to other users.

Appending data to an existing feature class

A common requirement for data management is to be able to append data to existing feature classes. The data loading commands described thus far have a -o append operation for appending data. A feature class must exist prior to using the append operation. If the feature class is multiversioned, it must be in an "open" state. It is also advisable to change the feature class to load-only I/O mode and pause the spatial indexing operations before loading the data to improve the data loading performance. The spatial indexes will be re-created when the feature class is put back into normal I/O mode. Because the feature class has been defined, the metadata exists and is not altered by the append operation.

In the shp2sde example below, the previously created "roads" feature class appends features from a shapefile, "rdshp2". All existing features, loaded from the rdshp shapefile, remain intact, and ArcSDE updates the feature class with the new features from the rdshp2 shapefile.

```
sdelayer -o load_only_io -l roads,shape -u beetle -p bug
```

```
shp2sde -o append -f rdshp2 -l roads,shape -u beetle -p bug
sdelayer -o normal_io -l roads,shape -u beetle -p bug
sdetable -o update_dbms_stats -t roads -u beetle -p bug
```

Note the last command in the sequence. The sdetable update_dbms_stats operation updates the table and index statistics required by the DB2 optimizer. Without the statistics the optimizer may not be able to select the best execution plan when you query the table. For more information on updating statistics, see Chapter 2, 'Essential configuring and tuning'.

Creating and populating raster columns

Raster columns are created from ArcGIS Desktop using ArcCatalog or ArcMap™. To create a raster column, you will first need to convert the image file into a format acceptable to ArcSDE. Then after the image has been converted to the ESRI raster file format, you can convert it into a raster column.

For more information on creating raster columns using either ArcCatalog or ArcToolbox, refer to *Building a Geodatabase.*

To understand how ArcSDE stores rasters in DB2, refer to Appendix A, 'Storing raster data'.

Creating views

There are times when a (DBMS) view is required in your database schema. ArcSDE provides the sdetable create_view operation to accommodate this need. The view creation is much like any other DB2 view creation. If you want to create a view using a layer and you want the resulting view to appear as a feature class to client applications, include the feature class's spatial column in the view definition. As with the other ArcSDE commands, see ArcSDE Developer Help for more information.

Exporting data

As with importing data, there are client applications that export data from ArcSDE as well. With ArcSDE, the following command line tools exist:

sdeexport—creates an ArcSDE export file to easily move feature class data between DB2 instances and other supported DBMSs

sde2shp—creates an ESRI shapefile from an ArcSDE feature class

sde2cov—creates a coverage from an ArcSDE feature class

sde2tbl—creates a dBASE or INFO file from a DBMS table

Schema modification

There will be occasions when it is necessary to modify the schema of some tables. You may need to add columns from a table. The ArcSDE command to do this is sdetable with the -o alter option. ArcCatalog offers an easy-to-use tool for this and other schema operations such as modifying the spatial index (grids) and adding and dropping column indexes.

Choosing an ArcSDE logfile configuration

ArcSDE allows you to configure the allocation of ArcSDE logfiles to your users. You can allow your users to own their own logfiles or they can check out a logfile from a pool of logfiles owned by the sde user. Logfiles can be shared, session-based or stand-alone. A shared logfile is the default and is used by all sessions that connect as a given user. Also if the ArcSDE server is configured to use stand-alone logfiles and all available logfiles of this type is exhausted, ArcSDE will attempt to create a session-based logfile if they are allowed; otherwise a shared logfile is created. If the shared logfile cannot be created, ArcSDE returns an error.

Shared ArcSDE logfiles

Shared logfiles are shared by all sessions that connect as the same user. Essentially, all sessions are inserting and deleting records from the same logfile data table. The logfiles are created the first time any session connects and remain in user's schema. To configure your server to use only shared logfiles; set the logfile server configuration parameters as follows:

```
MAXSTANDALONELOGS       0
ALLOWSESSIONLOGFILE     FALSE
```

Session-based ArcSDE logfiles

For session-based logfiles, each session that connects to the server creates a logfile.

A session-based logfile is dropped when a sessions disconnects. To configure your server to use session-based logfiles, set the server configuration parameter ALLOWSESSIONLOGFILE to true.

```
ALLOWSESSIONLOGFILE     TRUE
```

You need to make sure that you configure enough space for the tables and indexes of the session-based logfiles. The DBTUNE SESSION_STORAGE and SESSION_INDEX storage parameters control the storage of session-based logfiles.

Stand-alone ArcSDE logfiles

Stand-alone logfiles are created by a session for each logfile the application needs to store. When an application deletes the logfile, the stand-alone logfile is truncated. The stand-alone logfiles are dropped when the session disconnects. To configure your server to use stand-alone logfiles, set the server configuration parameter MAXSTANDALONELOGS to the number of stand alone logfiles you want them to be able to create.

For instance, set MAXSTANDALONELOGS to 6 if you want to allow each ArcSDE session to create a maximum of 6 stand-alone logfiles.

```
MAXSTANDALONELOGS 6
```

Keep in mind that you need to configure enough space to store all of these logfiles. The DBTUNE parameters, SESSION_STORAGE and SESSION_INDEX, allocate space for the tables and indexes of stand-alone logfiles.

If the application exhausts the number of allowable standalone logfiles—if the application needs to simultaneously create more logical logfiles than MAXSTANDALONELOGS allows—ArcSDE will attempt to create a session-based logfile, but only if ALLOWSESSIONLOGFILE is set to TRUE; otherwise ArcSDE will use a shared logfile. The shared logfile is created if it does not already exist. If the shared logfile cannot be created, ArcSDE returns an error.

Using an sde user pool of ArcSDE logfiles

The sde user can create a pool of logfiles that can be checked out and used as either session-based or stand-alone logfiles by other users. Using a pool of sde owned logfiles avoids having to grant users CREATE TABLE privileges. Shared logfiles cannot be checked out from an sde owned logfile pool.

To create a pool of logfiles, set the configuration parameter LOGPOOLSIZE to the number of logfiles that need to be created. This number should reflect the number of sessions that will connect to your server, in addition to the stand-alone logfiles if allowed. To calculate the total number of logfiles that could be checked out of the pool, use the following formulae:

If session logfiles are allowed, but not stand-alone logfiles:

```
LOGPOOLSIZE = total sessions expected
```

If stand-alone logfiles are allowed, but not session logfiles:

```
LOGPOOLSIZE = MAXSTANDALONELOGS * total sessions expected
```

If both stand-alone logfiles and session logfiles are allowed:

```
LOGPOOLSIZE = (MAXSTANDALONELOGS + 1) * total sessions expected
```

For instance, if you compute that 100 logfiles are needed, the LOGPOOLSIZE parameter would be set as follows:

```
LOGPOOLSIZE 100
```

If the pool is exhausted and another logfile is needed, ArcSDE will attempt to create it in the users schema. If the logfile cannot be created, an error is returned.

The pooled logfile tables are created or dropped whenever the LOGFILESIZE parameter is changed.

Set the HOLDLOGPOOLTABLES server configuration parameter to TRUE if you want the sessions to retain checked out logfiles. If set to false, the logfiles are released whenever the application deletes all of its logfiles in the case of a session logfile or whenever the logfile occupying a stand alone logfile is deleted.

The storage of the tables and indexes of the logfile pool is controlled by the DBTUNE storage parameters SESSION_STORAGE and SESSION_INDEX.

Using the ArcGIS Desktop ArcCatalog and ArcToolbox applications

So far the discussion has focused on ArcSDE command line tools that create feature class schemas and load data into them. While robust, these commands can be daunting for the first-time user. In addition, if you are using ArcGIS Desktop, you may have to use ArcCatalog to create feature datasets and feature classes within those feature datasets to use specific ArcGIS Desktop functionality. For that reason, a glimpse of how to use ArcToolbox and ArcCatalog to load data is provided. Please refer to the ArcGIS Desktop documentation on ArcCatalog, ArcToolbox, and the geodatabase for a full discussion of these tools.

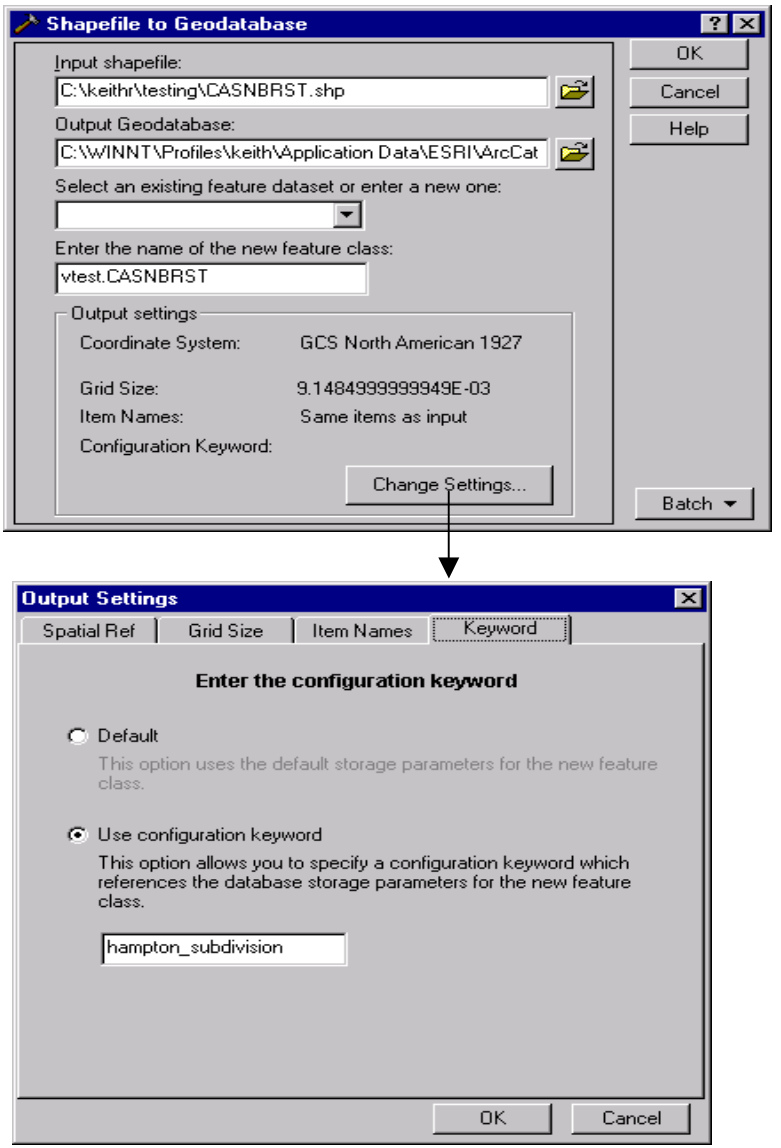
Loading data

You can convert ESRI shapefiles, coverages, Map LIBRARIAN layers, and ArcStorm™ layers into geodatabase feature classes with the ArcToolbox and ArcCatalog applications. ArcToolbox provides a number of tools that enable you to convert data from one format to another.

ArcToolbox operations, such as the ArcSDE administration commands shp2sde, cov2sde, and sdeimport, accept configuration keywords.

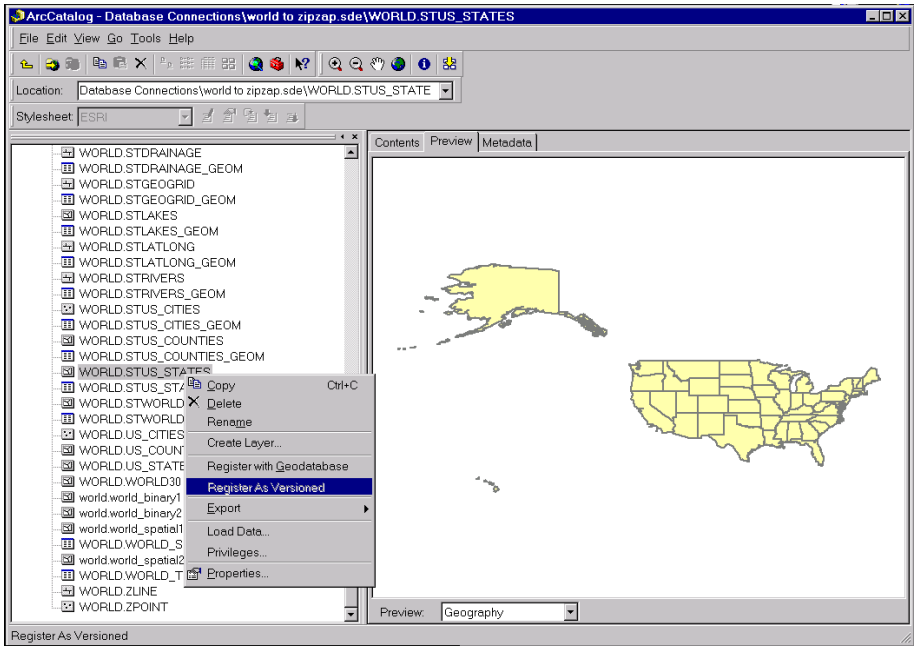
In the ArcToolbox Shapefile to Geodatabase wizard, you can see that a configuration keyword has been specified for the loading of the hampton_streets shapefile into the geodatabase.

The shapefile CASNBRST.shp is converted to feature class vtest.CASNBRST using ArcToolbox.



Versioning your data

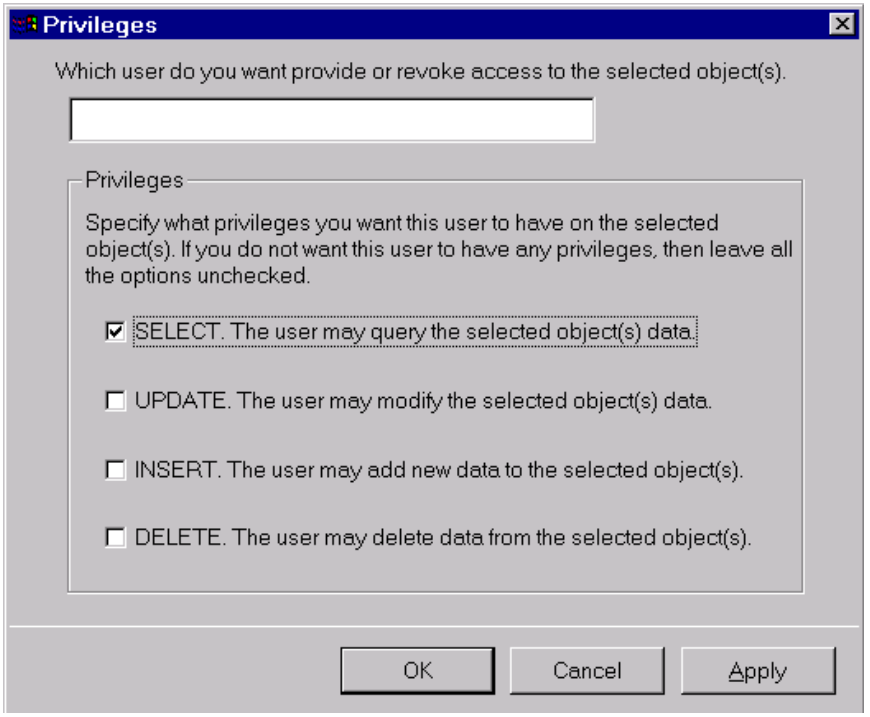
ArcCatalog also provides a means for registering data as multiversioned. Simply right-click the feature class to be registered as multiversioned and select the Register As Versioned context menu item.



A feature class is registered as multiversioned from within ArcCatalog.

Granting privileges

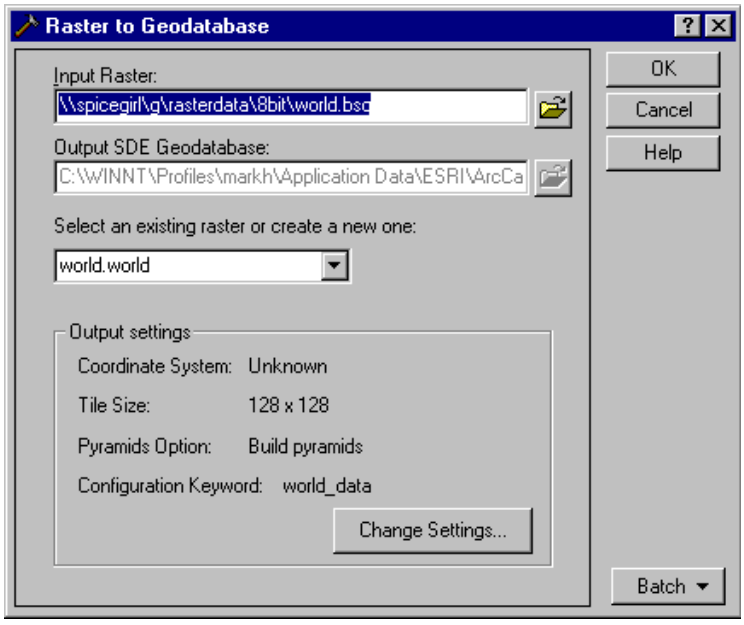
Using ArcCatalog, right-click the data object class and click the Privileges context menu. From the Privileges context menu assign privileges specifying the username or group, for example, PUBLIC and the privilege you wish to grant to or revoke from a particular user or group.



The ArcCatalog Privileges menu allows the owner of an object class, such as a feature dataset, feature class, or table, to assign privileges to other users or roles.

Creating a raster column with ArcCatalog

Using ArcCatalog, right-click the database connection, point to Import, and click Raster to Geodatabase. Navigate to the raster file to import. Click Change Settings if you want to change the coordinate reference system, tile size, pyramids option, or configuration keyword. Click OK to import the raster file into the DB2 database.



Efficiently registering large business tables with ArcSDE

When you create a business table in an ArcSDE database using an ArcSDE client application, for example, ArcCatalog or sdetable, ArcSDE registers the business table in the SDE.TABLE_REGISTRY. During the registration process ArcSDE performs a number of tasks depending on the type of registration requested. The duration of the registration process is dependent on the type of registration, whether the business table has a spatial column, and the table's number of rows. For large business tables, the registration process can take a long time to complete. This section provides the most efficient method to register tables with a large number of records.

Registering a table as NONE or USER maintained

Tables registered as NONE are registered without a row ID column.

Tables registered as USER are registered with a row ID column whose values you must maintain.

If the registration type is NONE or USER, ArcSDE merely adds a record to the SDE.TABLE_REGISTRY that references the business table. For tables registered as type USER the name of the row id is also added to the SDE.TABLE_REGISTRY entry. In the case of user maintained row IDs, ArcSDE will ensure that the column exists in the business table before completing the registration.

Registration of these two registration types happens rather quickly.

Registering a table with an SDE maintained row ID

Tables registered as type SDE, must have a row id column that uniquely identifies the rows of the table.

Note: Tables registered by the geodatabase must be registered by ArcSDE with an SDE maintained row ID. If the geodatabase determines that the table has been registered with an SDE maintained row ID, the geodatabase registration process is relatively inexpensive.

If a table was registered with a USER maintained row ID, the geodatabase alters its row ID registration to be SDE maintained.

By default the geodatabase adds a column called objectid to the table and registers it as an SDE maintained row ID. If the objectid column already exists, and is not currently registered as SDE maintained, the geodatabase will add a new column to the table called objectid_1.

Creating a new SDE maintained row ID column

If the row id column does not exist when the table is registered, ArcSDE adds a column of type INTEGER, with a NOT NULL constraint. If the table contains rows, ArcSDE populates the column with unique ascending values starting at your specified minimum ID value. The minimum ID value defaults to 1 if left unspecified. It then creates a unique index on the column called R<registration_id>_SDE_ROWID_UK, where registration_id is the registration identifier ArcSDE assigns the table when it was registered.

ArcSDE creates a sequence generator called R<registration_id> and uses it to generate the next value of the row id column whenever a value is added to the column.

Using an existing column

If the row ID column already exists, ArcSDE confirms that the column was defined as an integer. If it does not, the registration fails.

Next, ArcSDE confirms that the column has a unique index. If the column was defined with a non-unique index, ArcSDE drops the index.

In the event that the column does not have a unique index, ArcSDE attempts to create a unique index on the column. If the index creation fails because the column contains non-unique values, ArcSDE repopulates the column with ascending values beginning at 1 and then creates the unique index. ArcSDE names the unique index R<registration_id>_SDE_ROWID_UK.

Next, ArcSDE verifies that the column has been defined as NOT NULL.

If the column was defined as NULL, ArcSDE attempts to redefine it as NOT NULL. If this action fails, ArcSDE repopulates the column and defines it as NOT NULL.

Repopulating the column either because it contained null values, or because it contained non-unique values is an expensive process, especially if the table contains more than a 100,000 records.

Therefore, if at all possible, you should not rely on ArcSDE to perform this operation. Instead, define the row ID column as not null when the table is created and create your own unique index on it. At the very least, you should ensure that the column is populated with unique integer values.

Registering a table as multiversioned

To perform versioned edits on a business table, the table must be registered as multiversioned. These tables, as the name implies, store the records of the business that are added and deleted. They are named A<registration_id> and D<registration_id>.When tables are registered as multiversioned, the associated adds and deletes tables are created, and along with the business table, ArcSDE updates their DBMS statistics.

Multiversioned views are available for SQL access to the multiversioned database. See the *ArcSDE Developers Guide* for more information.

How does ArcSDE use existing DB2 tables?

Tables with spatial columns can be created by other applications. ArcSDE has been designed to use tables containing spatial columns that were created by other applications.

Automatic discovery of tables with spatial columns

Whenever an ArcSDE client lists the feature classes stored in the database, ArcSDE can automatically search the DB2 system tables for new tables with spatial columns. When a new table is discovered, it is registered with ArcSDE and made available to applications.

ArcSDE uses the first record in a newly discovered table to establish the ArcSDE geometry type. If the table contains multiple geometry types, the sdelayer administration utility can be used to alter the geometry type definition.

ArcSDE searches for a column in the table to use as a row_id column. To qualify, the column must be defined as INTEGER, NOT NULL, and UNIQUE constraints. If such a column is found, it is recorded in the ArcSDE table registry along with the table. If a row_id column is not found, the table is registered, but operations requiring a row_id are unavailable.

By default, automatic discovery of tables with spatial columns , often referred to as auto-registration, is disabled. In order to enable this feature use the "sdeconfig –o alter" command and set DISABLEAUTOREG to FALSE. For example

sdeconfig –o alter –v DISABLEAUTOREG=FALSE –u sde –p sde –D mydb –I 5151

ArcSDE to DB2 Server Data Type Mapping

ArcSDE uses 12 general data types. These types are mapped to DB2 Server types in the following matrix.

ArcSDE Data Type	DB2 Server Data type
SE_STRING_TYPE	CHAR, VARCHAR
SE_NSTRING_TYPE	VARGRAPHIC, GRAPHIC
SE_NCLOB_TYPE	LONG VARGRAPHIC, DBLOB
SE_INT16_TYPE (SE_SMALLINT_TYPE)	SMALLINT
SE_INT32_TYPE (SE_INTEGER_TYPE)	INTEGER
SE_INT64_TYPE	BIGINT

SE_FLOAT32_TYPE (SE_FLOAT_TYPE)	FLOAT
SE_FLOAT64_TYPE (SE_DOUBLE_TYPE)	DOUBLE
SE_DATE_TYPE	TIMESTAMP
SE_UUID_TYPE	CHAR, (UUID LEN)
SE_BLOB_TYPE	BLOB

CHAPTER 5

National language support

Storing data in an ArcSDE DB2 database using character sets other than the DB2 default requires some extra configuration on both the client and the server. This section provides guidelines for configuring both the DB2 database and the ArcSDE client environment to enable the use of character sets other than the default.

DB2 database character sets

If you are using Solaris™ and AIX, a DB2 database is created by default with CODESET ISO8859-1. On HP® 64-bit the default CODESET is roman8 and on Linux® the default CODESET is ISO-8859-1 On Windows®, the default CODESET is IBM-1252 The CODESET is selected when the database is created with the CREATE DATABASE statement and cannot be changed afterwards. To change the CODESET and TERITORRY, the database must be re-created and the data reloaded. Consult the *DB2 National Language Support Guide* for your DB2 release to determine the character set that is right for your data.

Setting the DB2CODEPAGE

Once the ArcSDE DB2 database has been created with the proper CODESET and TERITORRY, data can be loaded into it using a variety of applications such as ArcGIS Desktop and the ArcSDE administration tools shp2sde and cov2sde.

The application code page is derived from the active environment when the database connection is made. If the DB2CODEPAGE registry variable is set, its value is taken as the application code page. However, it is not necessary to set the DB2CODEPAGE registry variable because DB2 will determine the appropriate code page value from the operating system. Setting the DB2CODEPAGE registry variable to incorrect values may cause unpredictable results.

The database code page is derived from the value specified (explicitly or by default) at the time the database is created. For example, the following defines how the active environment is determined in different operating environments:

UNIX On UNIX-based operating systems, the active environment is determined from the locale setting, which includes information about language, territory and code set.

Windows operating systems For all Windows operating systems, if the DB2CODEPAGE environment variable is not set, the code page is derived from the ANSI code page setting in the Registry.

client code page If the DB2CODEPAGE variable is set, the client code page is the value of DB2CODEPAGE. Otherwise, the client code page is the client's operating system locale.

server code page also called the server operating system locale code page. It is the operating system locale on which the DB2 database is installed.

For example, if the DB2 database, installed on Windows, has been created with the Shift JIS, DB2 CODEPAGE notation 932, character encoding and you want to access this database from a Unix client running in the EUC JP, DB2 CODEPAGE notation 954, character encoding, you will have to set the DB2CODEPAGE variable to 954. This ensures that all character data transferred between the ArcSDE server and the ArcSDE client uses the 954 CODEPAGE setting.

For C API Client applications, set the DB2CODEPAGE as an environment variable.

```
setenv DB2CODEPAGE  954
```

If the situation were reversed, and the DB2 database was on Unix and created with the EUC JP character encoding and the client is running on Windows, you will have to set the DB2CODEPAGE environment variable on the Windows client to 932.

For Windows clients such as ArcCatalog or ArcMap, click Start, Settings, and Control Panel. Double-click the System icon and click on the Environment tab after the System menu appears. Click the System Variables scrolling list and enter DB2CODEPAGE in the Variable: input line 932 in the Value: input line. Click Set and OK.

For C API and ArcSDE client utility applications like shp2sde that run from the DOS environment, set the DB2CODEPAGE as an environment variable.

```
set DB2CODEPAGE=932
```

Setting the DB2CODEPAGE variable for windows

Be careful setting the DB2CODEPAGE on Windows because there are actually two different CODEPAGE environments on this platform. The character encoding standards supported by the Windows environment are different from that supported by the MS–DOS environment. Windows applications such as ArcGIS Desktop run in the Windows American National Standards Institute (ANSI) CODEPAGE environment, while ArcSDE administration tools and C and Java API applications invoked from the MS–DOS Command Prompt run in the original equipment manufacturer (OEM) CODEPAGE environment.

Setting the DB2CODEPAGE variable for a remote ArcSDE setup

If you use direct connect to connect to a DB2 server from a remote computer or if you start the ArcSDE application server on a computer that is remote from the DB2 server you will need to set the DB2CODEPAGE variable for ArcSDE.

In the case of the direct connect client the DB2CODEPAGE variable must be set in the environment. If you are using an ArcSDE admin tool such as shp2sde, that is started from a MS–DOS command tool, set the DB2CODEPAGE variable in the MS DOS environment before executing the command.

Direct connections from a Windows application such as ArcMap require that the DB2CODEPAGE variable be set in the Windows environment. Click Start, Settings, Control Panel and double click the System icon. Click the Advanced tab and click the Environment Variables button. Click New to enter the DB2CODEPAGE variable. The application must be restarted to consume the DB2CODEPAGE variable.

To set the DB2CODEPAGE for a remote application server, set the variable in the dbinit.sde file. When the application server is started the variable is read from the file. To ensure that the variable is set in the ArcSDE application server environment check the ArcSDE application server's variable settings with the ArcSDE `sdemon –o info –I vars command`.

The following ArcSDE/DB2 configuration illustrates how the DB2CODEPAGE variable should be set in a remote setup. Consider the case where the language environment is Eastern European, DB2 is installed on a UNIX server, the ArcSDE application server is running on a Windows server and a user is connecting from Windows on yet another computer. The DB2 database was created with the ISO8859-2 CODESET. Before the administrator starts the ArcSDE application server the following windows ANSI cp1250 CODEPAGE DB2CODEPAGE variable must be added to the dbinit.sde file:

```
set DB2CODEPAGE=912
```

The user connects to the application server with ArcMap, but before doing so sets the DB2CODEPAGE variable to the windows ANSI cp1250 code page value as:

```
DB2CODEPAGE=912
```

The user wishes to use SQLPLUS from the DB2 command window to query a table. In the command window the user enters the following DOS OEM 852 DB2CODEPAGE variable:

```
set DB2CODEPAGE=852
```

Character encoding standards supported by ArcSDE

For a complete list of character encoding standards supported by your DB2 database and their naming conventions, please refer to Supported territory codes and code pages in the application development document for your version of DB2. Currently ArcSDE only supports conversions between the character encoding standards listed in the table below.

Encoding name	Description
950	BIG5 16-bit Traditional Chinese
964	EUC 32-bit Traditional Chinese
932	Shift-JIS 16-bit Japanese
954	EUC 24-bit Japanese
949	KSC5601 16-bit Korean
819	ISO 8859-1 West European
912	ISO 8859-2 East European
915	ISO 8859-5 Latin/Cyrillic
1089	ISO 8859-6 Latin/Arabic
813	ISO 8859-7 Latin/Greek
916	ISO 8859-8 Latin/Hebrew

920	ISO 8859-9 West European & Turkish
437	IBM-PC Code Page 437 8-bit American
850	IBM-PC Code Page 850 8-bit West European
851	IBM-PC Code Page 851 8-bit Greek/Latin
852	IBM-PC Code Page 852 8-bit East European
855	IBM-PC Code Page 855 8-bit Latin/Cyrillic
857	IBM-PC Code Page 857 8-bit Turkish
860	IBM-PC Code Page 860 8-bit West European
861	IBM-PC Code Page 861 8-bit Icelandic
863	IBM-PC Code Page 863 8-bit Canadian French
865	IBM-PC Code Page 865 8-bit Norwegian
866	IBM-PC Code Page 866 8-bit Latin/Cyrillic
869	IBM-PC Code Page 869 8-bit Greek/Latin
737	IBM-PC Code Page 737 8-bit Greek/Latin
775	IBM-PC Code Page 775 8-bit Baltic
1250	MS Windows Code Page 1250 8-bit East European
1251	MS Windows Code Page 1251 8-bit Latin/Cyrillic
1252	MS Windows Code Page 1252 8-bit West European
1253	MS Windows Code Page 1253 8-bit Latin/Greek
1254	MS Windows Code Page 1254 8-bit Turkish
1255	MS Windows Code Page 1255 8-bit Latin/Hebrew

1256	MS Windows Code Page 1256 8-Bit Latin/Arabic
1257	MS Windows Code Page 1257 8-bit Baltic
1258	MS Windows Code Page 1258 8-bit Vietnamese

Unicode support

The Unicode project for ArcSDE involves support for the new SDE attribute datatypes SE_NSTRING_TYPE and SE_NCLOB_TYPE and internal datatype SE_WCHAR.

In DB2, no data type exists which can be used specifically for WCHAR column types. You can retrieve, insert, or modify data as an WCHAR type, however, you cannot create a column specifically as an WCHAR type, nor can you describe and detect that a column is meant to be handled exclusively as WCHAR. The application must either handle all strings as WCHAR, or must know enough to force specific columns to be WCHAR.

In order to fully support unicode for DB2, it appears that we will need additional metadata associated with a column flagging it to be handled exclusively as an NSTRING or NCLOB.

Character columns are stored in the DB2 database in the locale specific codepage that the database was created in. Conversions occur at the server between database code page and client application code page - with possible data loss depending on code pages involved.

In order to store character data as unicode (UTF-8 & UCS-2 - where CHAR, VARCHAR, LONG VARCHAR, CLOB is stored as UTF-8 and GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, DBCLOB is stored as UCS-2), the database must be created using the UTF-8 (or codepage# 1208) codepage, with appropriate territory.

An application can retrieve character data as unicode either by setting the client (SDE) local code page environment to UTF-8, or by connecting as a unicode application (SQLConnectW or SQLSetConnectAttr SQL_ATTR_ANSI_APP = SQL_AA_FALSE) and then binding to ANSI or UNICODE buffers. In the ArcSDE implementation, we connect as a UNICODE application using SQLConnectW and then bind to ANSI (SQL_C_CHAR) or UNICODE (SQL_C_WCHAR) buffers.

At one point it was suggested that the GRAPHIC datatypes should be used to store UNICODE data. However, this may not be practical. The GRAPHIC types are meant to store double-byte character data and are not available unless the database was created using a double-byte character set code page. There are also restrictions on operations that can be performed on the GRAPHIC types vs. CHAR types or UTF-8/UCS-8 data making it impractical to use.

APPENDIX A

Storing raster data

A raster is a rectangular array of equally spaced cells that, taken as a whole, represent thematic, spectral, or picture data. Raster data can represent everything from qualities of land surface, such as elevation or vegetation, to satellite images, scanned maps, and photographs.

You are probably familiar with raster formats, such as tagged image file format (TIFF), Joint Photographic Experts Group (JPEG), and Graphics Interchange Format (GIF), that your Internet browser renders. These rasters are composed of one or more bands. Each band is segmented into a grid of square pixels. Each pixel is assigned a value that reflects the information it represents at a particular position.

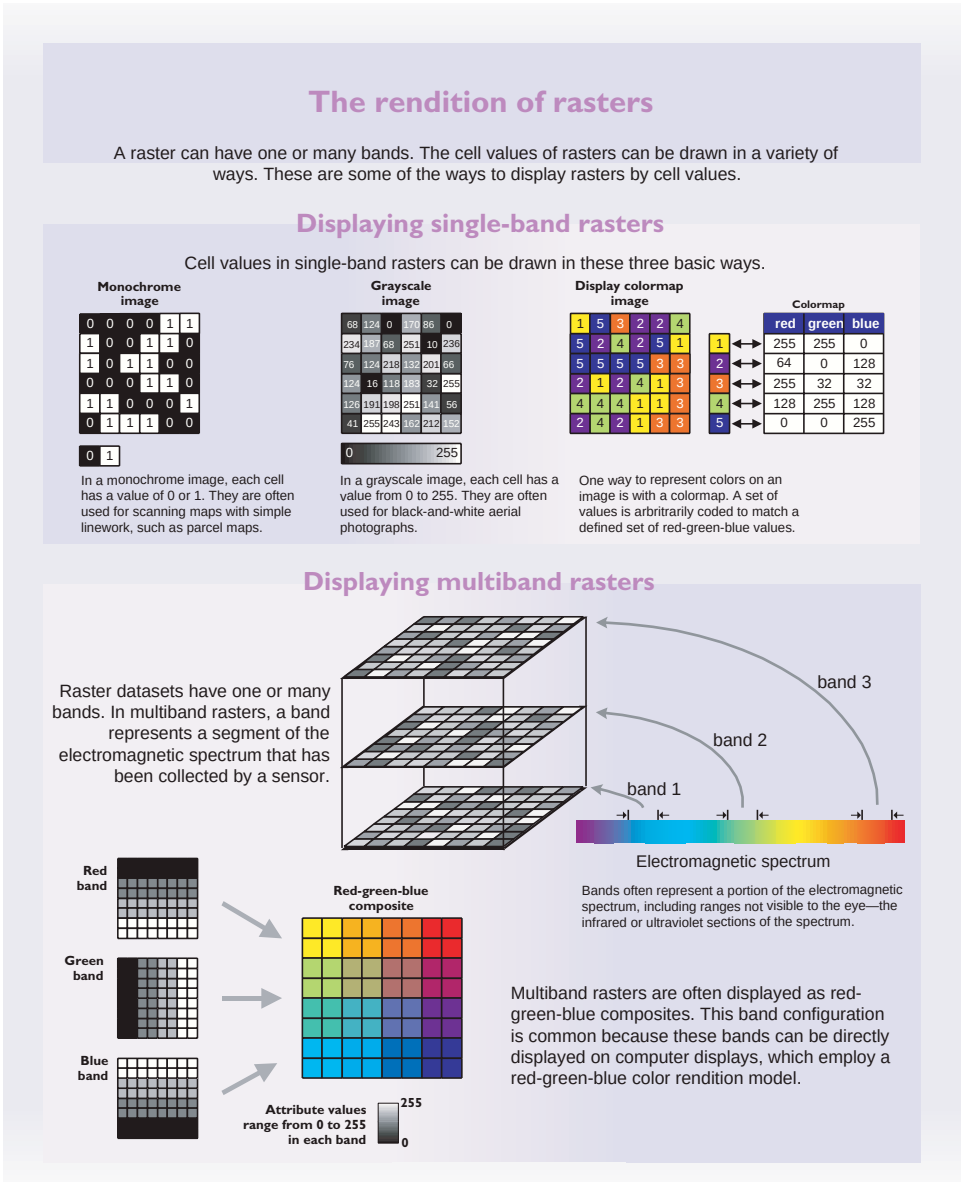
For an expanded discussion of the type of raster data supported by ESRI products, review Chapter 9, 'Cell-based modeling with rasters', in *Modeling Our World*.

A raster column is added to a business table, and each cell of the raster column contains a reference to a raster stored in a separate raster table. Therefore, each row of a business table references an entire raster.

ArcSDE stores the raster bands in the raster band table. ArcSDE joins the raster band table to the raster table on the raster_id column. The raster band table's raster_id column is a foreign key reference to the raster table's raster_id primary key.

ArcSDE automatically stores any existing image metadata, such as image statistics, color maps, or bitmasks, in the raster auxiliary table. The rasterband_id column of the raster auxiliary table is a foreign key reference to the primary key of the raster band table.

ArcSDE joins the two tables on this primary/foreign key reference when accessing a raster band's metadata.



The raster blocks table stores the pixels of each raster band. ArcSDE tiles the pixels into blocks according to a user-defined dimension. ArcSDE does not have a default dimension; however, applications that store raster data in ArcSDE do. ArcToolbox and ArcCatalog, for example, use default raster block dimensions of 128-by-128 pixels per block. The dimensions of the raster block along with the compression method, if one is specified, determine the storage size of each raster block.

The raster blocks table contains the rasterband_id column, which is a foreign key reference to the raster band table's rasterband_id primary key. ArcSDE joins these tables together on the primary/foreign key reference when accessing the blocks of the raster band.

ArcSDE populates the raster blocks table according to a declining resolution pyramid. The number of levels specified by the application determines the height of the pyramid. ArcToolbox and ArcCatalog calculate the pyramid for you, so there is no need to define the number of levels.

The pyramid begins at the base, or level 0, which contains the original pixels of the image. The pyramid proceeds toward the apex by coalescing four pixels from the previous level into a single pixel at the current level. This process continues until less than four pixels remain or until ArcSDE exhausts the defined number of levels.

The apex of the pyramid is reached when the uppermost level has less than four pixels. The additional levels of the pyramid increase the number of raster block table rows by one third. However, since it is possible for the user to specify the number of levels, the true apex of the pyramid may not be obtained, limiting the number of records added to the raster blocks table.

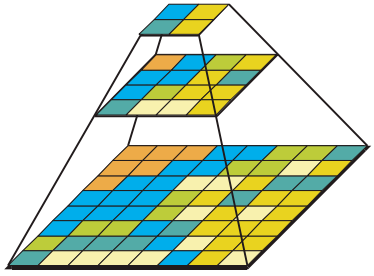


Figure A.1 When you build a pyramid, more rasters are created by progressively downsampling the previous level by a factor of two until the apex is reached. As the application zooms out and the raster cells grow smaller than the resolution threshold, ArcSDE selects a higher level of the pyramid. The purpose of the pyramid is to optimize display performance.

The pyramid allows ArcSDE to provide the application with a constant resolution of pixel data regardless of the rendering window's scale. Data of a large raster transfers quicker to the client when a pyramid exists since ArcSDE transfers fewer cells of reduced resolution.

Raster schema

When you import a raster into an ArcSDE database, ArcSDE adds a raster column to the business table of your choice. You may name the raster column whatever you like, so long as it conforms to DB2's column naming convention. ArcSDE restricts one raster column per business table.

The raster column is a foreign key reference to the raster_id column of the raster table created during the addition of the raster column. Also joined to the raster table's raster_id primary key, the raster band table stores the bands of the image. The raster auxiliary table, joined one-to-one to the raster band table by rasterband_id, stores the metadata of each raster band. The rasterband_id also joins the raster band table to the raster blocks table in a many-to-one relationship. The raster blocks table rows store blocks of pixels, determined by the dimensions of the block.

The sections that follow describe the schema of the tables associated with the storage of raster data. Refer to Figure A.2 for an illustration of these tables and the manner in which they are associated with one another.

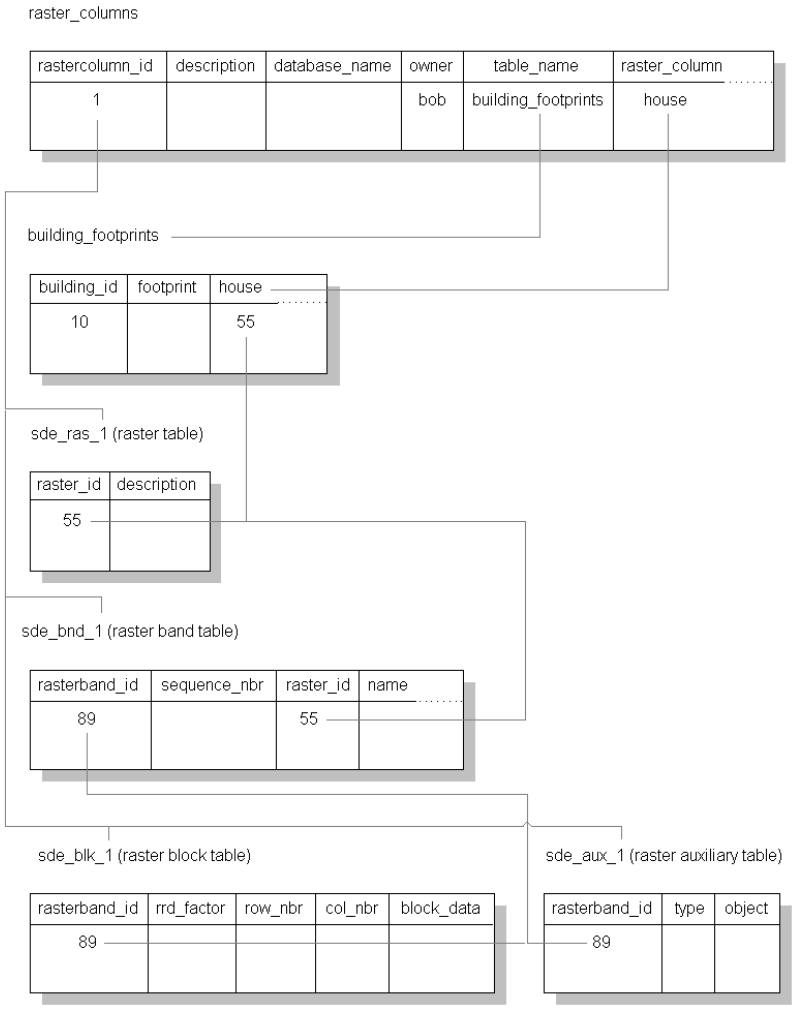


Figure A.2 When ArcSDE adds a raster column to a table, it records that column in the sde user's raster_columns table. The rastercolumn_id table is used in the creation of the table names of the raster, raster band, raster auxiliary, and raster blocks table.

RASTER_COLUMNS table

When you add a raster column to a business table, ArcSDE adds a record to the RASTER_COLUMNS system table maintained in the sde user's schema. ArcSDE also creates four tables to store the raster images and metadata associated with each one.

NAME	DATA TYPE	NULL?
rastercolumn_id	INTEGER	NOT NULL
description	VARCHAR(65)	NULL
database_name	VARCHAR(32)	NULL
owner	VARCHAR(32)	NOT NULL
table_name	VARCHAR(128)	NOT NULL
raster_column	VARCHAR(128)	NOT NULL
cdate	INTEGER	NOT NULL
config_keyword	VARCHAR(32)	NULL
minimum_id	INTEGER	NULL
base_rastercolumn_id	INTEGER	NOT NULL
rastercolumn_mask	INTEGER	NOT NULL
srid	INTEGER	NULL

Raster columns table

- rastercolumn_id (SE_INTEGER_TYPE)—The table's primary key.
- description (SE_STRING_TYPE)—The description of the raster table.
- database_name (SE_STRING_TYPE)—The DB2 database name.
- owner (SE_STRING_TYPE)—The schema of the raster column's business table.
- table_name (SE_STRING_TYPE)—The business table name.
- raster_column (SE_STRING_TYPE)—The raster column name.
- cdate (SE_INTEGER_TYPE)—The date the raster column was added to the business table.
- config_keyword (SE_STRING_TYPE)—The DBTUNE configuration keyword whose storage parameters determine how the tables and indexes of the raster are stored in the DB2 database. For more information on DBTUNE configuration keywords and their storage parameters, review Chapter 3, ‘Configuring DBTUNE storage parameters'.
- minimum_id (SE_INTEGER_TYPE)—Defined during the creation of the raster, it establishes the value of the raster table's raster_id column.
- base_rastercolumn_id (SE_INTEGER_TYPE)—If a view of the business table is created that includes the raster column, an entry is added to the RASTER_COLUMNS table. The raster column entry of the view will have its own rastercolumn_id. The base_rastercolumn_id will be the rastercolumn_id of the business table used to create the view. This base_rastercolumn_id maintains referential integrity to the business table. It ensures that actions performed on the business table raster column are reflected in the view. For example, if the business

table's raster column is dropped, it will also be dropped from the view, essentially removing the view's raster column entry from the RASTER_COLUMNS table.

- rastercolumn_mask (SE_INTEGER_TYPE)—Currently not used, maintained for future use.
- srid (SE_INTEGER_TYPE)—The spatial reference ID (SRID) is a foreign key reference to the DB2GSE.GSE_SPATIAL_REF table. For images that can be georeferenced, the SRID references the coordinate reference system the image was created under.

Business table

In the example that follows, the fictitious BUILD_FOOTPRINTS business table contains the raster column house_image. This is a foreign key reference to the raster table created in the user's schema. In this case the raster table contains a record for each raster of a house. It should be noted that images of houses cannot be georeferenced. Therefore, the SRID column of the RASTER_COLUMN record for this raster is NULL.

NAME	DATA TYPE	NULL?
building_id	INTEGER	NOT NULL
building_footprint	INTEGER	NOT NULL
house_picture	INTEGER	NOT NULL

BUILDING_FOOTPRINTS business table with house image raster column

- building_id (SE_INTEGER_TYPE)—The table's primary key
- building_footprints (SE_INTEGER_TYPE)—A spatial column and foreign key reference to a feature table containing the building footprints
- house_image (SE_INTEGER_TYPE)—A raster column and foreign key reference to a raster table containing the images of the houses located on each building footprint

Raster table (SDE_RAS_<rastercolumn_id>)

The raster table, created as SDE_RAS_<raster_column_id> in the DB2 database, stores a record for each image stored in a raster column. The raster_column_id column is assigned by ArcSDE whenever a raster column is created in the database. A record for each raster column in the database is stored in the ArcSDE RASTER_COLUMNS system table maintained in the sde user's schema.

NAME	DATA TYPE	NULL?
raster_id	INTEGER	NOT NULL
raster_flags	INTEGER	NULL
description	VARCHAR(65)	NULL

Raster description table schema (SDE_RAS_<raster_column_id>)

- raster_id (SE_INTEGER_TYPE)—The primary key of the raster table and unique sequential identifier of each image stored in the raster table
- raster_flags (SE_INTEGER_TYPE)—A bitmap set according to the characteristics of a stored image
- description (SE_STRING_TYPE)—A text description of the image (not implemented at ArcSDE 8.1)

Raster band table (SDE_BND_<rastercolumn_id>)

Each image referenced in a raster may be subdivided into one or more raster bands. The raster band table, created as SDE_BND_<rastercolumn_id>, stores the raster bands of each image stored in the raster table. The raster_id column of the raster band table is a foreign key reference to the raster table's raster_id primary key. The rasterband_id column is the raster band table's primary key. Each raster band in the table is uniquely identified by the sequential rasterband_id.

NAME	DATA TYPE	NULL?
rasterband_id	INTEGER	NOT NULL
sequence_nbr	INTEGER	NOT NULL
raster_id	INTEGER	NOT NULL
name	VARCHAR(65)	NULL
band_flags	INTEGER	NOT NULL
band_width	INTEGER	NOT NULL
band_height	INTEGER	NOT NULL
band_types	INTEGER	NOT NULL
block_width	INTEGER	NOT NULL
block_height	INTEGER	NOT NULL
block_origin_x	DOUBLE	NOT NULL
block_origin_y	DOUBLE	NOT NULL
eminx	DOUBLE	NOT NULL
eminy	DOUBLE	NOT NULL
emaxx	DOUBLE	NOT NULL
emaxy	DOUBLE	NOT NULL
cdate	INTEGER	NOT NULL
mdate	INTEGER	NOT NULL

Raster band table schema

- rasterband_id (SE_INTEGER_TYPE)—The primary key of the raster band table that uniquely identifies each raster band.
- sequence_nbr (SE_INTEGER_TYPE)—An optional sequential number that can be combined with the raster_id as a composite key as a second way to uniquely identify the raster band.
- raster_id (SE_INTEGER_TYPE)—The foreign key reference to the raster table's primary key. Uniquely identifies the raster band when combined with the sequence_nbr as a composite key.
- name (SE_STRING_TYPE)—The name of the raster band.
- band_flags (SE_INTEGER_TYPE)—A bitmap set according to the characteristics of the raster band.
- band_width (SE_INTEGER_TYPE)—The pixel width of the band.
- band_height (SE_INTEGER_TYPE)—The pixel height of the band.
- band_types (SE_INTEGER_TYPE)—A bitmap band compression data.
- block_width (SE_INTEGER_TYPE)—The pixel width of the band's tiles.
- block_height (SE_INTEGER_TYPE)—The pixel height of the band's tiles.
- block_origin_x (SE_DOUBLE_TYPE)—The leftmost pixel.
- block_origin_y (SE_DOUBLE_TYPE)—The bottom-most pixel.

If the image has a map extent, the optional eminx, eminy, emaxx, and emaxy will hold the coordinates of the extent.

- eminx (SE_DOUBLE_TYPE)—The band's minimum x-coordinate.
- eminy (SE_DOUBLE_TYPE)—The band's minimum y-coordinate.
- emaxx (SE_DOUBLE_TYPE)—The band's maximum x-coordinate.
- emaxy (SE_DOUBLE_TYPE)—The band's maximum y-coordinate
- cdate (SE_INTEGER_TYPE)—The creation date
- mdate (SE_INTEGER_TYPE)—The last modification date

Raster blocks table (SDE_BLK_<rastercolumn_id>)

Created as SDE_BLK_<rastercolumn_id>, the raster blocks table stores the actual pixel data of the raster images. ArcSDE evenly tiles the bands into blocks of pixels. Tiling the raster band data enables efficient storage and retrieval of the raster data.

The rasterband_id column of the raster block table is a foreign key reference to the raster band table's primary key. A composite unique key is formed by combining the rasterband_id, rrd_factor, row_nbr, and col_nbr columns.

NAME	DATA TYPE	NULL?
rasterband_id	INTEGER	NOT NULL
rrd_factor	INTEGER	NOT NULL
row_nbr	INTEGER	NOT NULL
col_nbr	INTEGER	NOT NULL
block_data	BLOB	NOT NULL

Raster block table schema

- rasterband_id (SE_INTEGER_TYPE)—The foreign key reference to the raster band table's primary key.
- rrd_factor (SE_INTEGER_TYPE)—The reduced resolution dataset factor determines the position of the raster band block within the resolution pyramid. The resolution pyramid begins at 0 for the highest resolution and increases until the raster band's lowest resolution level has been reached.
- row_nbr (SE_INTEGER_TYPE)—The block's row number.
- col_nbr (SE_INTEGER_TYPE)—The block's column number.
- block_data (SE_BLOB_TYPE)—The block's tile of pixel data.

Raster band auxiliary table (SDE_AUX_<rastercolumn_id>)

The raster band auxiliary table, created as SDE_AUX_<rastercolumn_id>, stores optional raster metadata such as the image color map, image statistics, and bitmasks used for image overlay and mosaicking. The rasterband_id column is a foreign key reference to the primary key of the raster band table.

NAME	DATA TYPE	NULL?
rasterband_id	INTEGER	NOT NULL
type	INTEGER	NOT NULL
object	BLOB	NOT NULL

Raster auxiliary table schema

- rasterband_id (SE_INTEGER_TYPE)—The foreign key reference to the raster band table's primary key
- type (SE_INTEGER_TYPE)—Abitmap set according to the characteristics of the data stored in the object column
- object (SE_BLOB_TYPE)—May contain the image color map, image statistics, etc.

APPENDIX B

DB2 Spatial Extender geometry types

ArcSDE for DB2 stores spatial data in the DB2 Spatial Extender data types. Therefore, before spatial data can be stored in a DB2 database, the Spatial Extender must be installed, and the database must be spatially enabled. This document describes the ArcSDE/DB2 Spatial Extender interface and provides a brief overview of the spatial data types and functions available after the database has been spatially enabled with the DB2 Spatial Extender. For more information about the DB2 Spatial Extender, see the IBM *DB2 Spatial Extender User's Guide* and *Reference.*

The DB2 Spatial Extender embeds a GIS into your DB2 database. The DB2 Spatial Extender module implements the Open GIS Consortium, Inc. (OpenGIS®, or OGC), SQL 3 specification of spatial types, columns capable of storing spatial data such as the location of a landmark, a street, or a parcel of land.

The (GIS) of the past was spatially centric and focused on gathering spatial data and attaching nonspatial attribute data to it. The Spatial Extender module integrates spatial and nonspatial data, providing a seamless point of access through the DB2 SQL interface.

In addition to new data types, the DB2 Spatial Extender provides new capabilities such as spatial joins. Application programmers typically join tables by comparing two or more columns to determine whether their values are equal, not equal, greater than, and so on. The DB2 Spatial Extender includes functions capable of comparing the values of spatial columns to determine if they intersect, overlap, and so forth. These two-dimensional functions can join tables based on their spatial relationship and answer

questions such as “Is this school within five miles of a hazardous waste site?” Internally, the DB2 Spatial Extender ST_Overlaps function evaluates this question as, “Does this polygon (the building footprint of a school) overlap this circular polygon (the five-mile radius of a hazardous waste site)?” An application programmer can join a table storing sensitive sites, such as schools, playgrounds, and hospitals, to another table containing the locations of hazardous sites and return a list of sensitive areas at risk.

How the DB2 Spatial Extender works

Once the DB2 Spatial Extender is installed, you can create spatially enabled tables that include spatial columns. Geographic features can be inserted into the spatial columns. The DB2 Spatial Extender converts spatial data into its storage format from one of the following external formats:

- Well-known text (WKT) representation
- Well-known binary (WKB) representation
- Geography Markup Language (GML) representation
- ESRI shape representation

ArcSDE uses the ESRI shape representation.

Accessing the spatially enabled tables through the ArcSDE server can be done by applications using the existing tools offered by the GIS software or by creating applications using the Spatial Database Engine™ (SDE®) C API. An experienced Open Database Connectivity (ODBC) programmer can also make calls to the DB2 Spatial Extender spatial functions. The majority of this document is devoted to discussing and applying these spatial functions.

After spatially enabling and loading data into your database, you can include Spatial Extender functions in your SQL statements, comparing the values of spatial columns, transforming the values into other spatial data, and describing the properties of the data.

Adding records to the DB2GSE.ST_SPATIAL_REFERENCES_SYSTEMS catalog view

The spatial reference system identifies the coordinate transformation matrix for each geometry. Geometry is the term adopted by the Open GIS Consortium to refer to two-dimensional spatial data. All spatial reference systems known to the database are stored in the DB2GSE.ST_SPATIAL_REFERENCE _SYSTEMS catalog view.

NAME	DATA TYPE	NULL?
srs_id	Integer	NOT NULL
srs_name	varchar(128)	NOT NULL
x_offset	Double	NOT NULL
x_scale	Double	NOT NULL
y_offset	Double	NOT NULL
y_scale	Double	NOT NULL
z_offset	Double	NOT NULL
z_scale	Double	NOT NULL
m_offset	Double	NOT NULL
m_scale	Double	NOT NULL
min_x	Double	NOT NULL
max_x	Double	NOT NULL
min_y	Double	NOT NULL
max_y	Double	NOT NULL
min_z	Double	NOT NULL
max_z	Double	NOT NULL
min_m	Double	NOT NULL
max_m	Double	NOT NULL
coordsys_name	varchar(128)	NOT NULL
coordsys_type	varchar(128)	NOT NULL
organization	varchar(128)	NULL
organization_coordsys_id	varchar(128)	NULL
defintion	varchar(2048)	NOT NULL
description	varchar(256)	NULL

DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view schema

The DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view stores a record for each spatial reference in the database.

The datatype for each column is defined below.

- srs_id (INTEGER_TYPE) — Contains the unique ID that identifies each SRID in the database.
- srs_name (SE_STRING_TYPE) — The name of the spatial reference system.
- x_offset (SE_DOUBLE_TYPE) — The x-value offset or the minimum allowable x-ordinate value.
- x_scale (SE_DOUBLE_TYPE)) — Scale factor by which to multiply the figure that results when an offset is subtracted from an x coordinate.
- y_offset (SE_DOUBLE_TYPE) — The y-value offset or the minimum allowable y-ordinate value.
- y_scale (SE_DOUBLE_TYPE)) — Scale factor by which to multiply the figure that results when an offset is subtracted from a y coordinate.
- z_offset (SE_DOUBLE_TYPE) — The z-value offset or the minimum allowable z-ordinate value.
- z_scale (SE_DOUBLE_TYPE)) — Scale factor by which to multiply the figure that results when an offset is subtracted from a z coordinate
- m_offset (SE_DOUBLE_TYPE) — The m-value offset or the minimum allowable M-ordinate value.
- m_scale (SE_DOUBLE_TYPE)) — Scale factor by which to multiply the figure that results when an offset is subtracted from a measure.
- min_x (SE_DOUBLE_TYPE) — minimum possible value for x-coordinates. This value is derived from the values in the x_offset and x_scale columns.
- max_x (SE_DOUBLE_TYPE) — maximum possible value for x-coordinates. This value is derived from the values in the x_offset and x_scale columns.
- min_y (SE_DOUBLE_TYPE) — minimum possible value for y-coordinates. This value is derived from the values in the y_offset and y_scale columns.
- max_y (SE_DOUBLE_TYPE) — maximum possible value for y-coordinates. This value is derived from the values in the y_offset and y_scale columns.
- min_z (SE_DOUBLE_TYPE) — minimum possible value for z-coordinates. This value is derived from the values in the z_offset and z_scale columns.

- max_z (SE_DOUBLE_TYPE) — maximum possible value for z-coordinates. This value is derived from the values in the z_offset and z_scale columns.
- min_m (SE_DOUBLE_TYPE) — minimum possible value for measures. This value is derived from the values in the m_offset and m_scale columns.
- max_m (SE_DOUBLE_TYPE) — maximum possible value for measures. This value is derived from the values in the m_offset and m_scale columns.
- coordsys_name (SE_STRING_TYPE) — The name of the coordinate system on which the spatial reference system is based.
- coordsys_type (SE_STRING_TYPE) — The type of the coordinate system on which the spatial reference system is based.
- organization (SE_STRING_TYPE) —Name of the organization that defines the coordinate system on which the spatial reference is based. Set to NULL if organization is NULL.
- organization_coordsys_id (SE_STRING_TYPE) — Name of the organization that defines the coordinate system on which the spatial reference is based. Set to NULL if organization is NULL.
- Definition (SE_STRING_TYPE) — WKT representation of the defined coordinate system.
- Description (SE_STRING_TYPE) — Description of the spatial reference system.

Internal functions use the parameters of a spatial reference system to translate and scale each floating-point coordinate of the geometry into 32-bit positive integers prior to storage. Upon retrieval, the coordinates are restored to their external floating point format.

The floating-point coordinates are converted to integers by subtracting the falsex and falsey values, which translates to the false origin, scales by multiplying by the xyunits, adds a half unit, and truncates the remainder.

The optional z-coordinates and measures are dealt with similarly, except that they are translated with falsez and falsem and scaled with zunits and munits, respectively.

The spatial reference identifier, the primary key, contains a unique number for each spatial reference system.

The spatial reference system is assigned to geometry during its construction. The spatial reference system must exist in the spatial reference table. All geometries in a column must have the same spatial reference system.

Whenever ArcSDE creates a feature class it searches the ST_SPATIAL_REFERENCE_SYSTEMS catalog view in an attempt to locate a matching spatial reference system. If one is found, the SRID is assigned to the feature class; otherwise, ArcSDE adds a new spatial reference system to the ST_SPATIAL_REFERENCE_SYSTEMS view and assigns it to the feature class.

The ArcSDE administration tools shp2sde columns and cov2sde columns provide an option for you to enter a predefined SRID when you use them to create a new feature class. In this example, the roads coverage is converted to the roads feature class with a SRID of 10. The coordinates of the coverage feature must fit within the extent of the spatial reference system. Each feature found to lie outside the spatial reference system's extent is rejected.

```
cov2sde -o create -l roads, feature -f roads -R 10 -g 100,0,0 -u world -p
world
```

Creating feature classes in a DB2 database

A DB2 spatial table can include one or more spatial columns, although ArcSDE restricts a feature class to a single spatial column. Spatial columns are defined with one of the DB2 Spatial Extender's UDTs. A spatial column can only accept data of the type required by the spatial column. For example, an ST_Polygon column rejects integers, characters, and even other types of nonpolygon geometry.

When ArcSDE creates a DB2 table with a spatial column, it also creates an OBJECTID integer column. The OBJECTID column is required by ArcSDE client applications to keep track of selection sets; more specifically it is used in ArcSDE logfiles.

A record is added to the ST_GEOMETRY_COLUMNS catalog view whenever ArcSDE creates a feature class in a DB2 database. This record is added to the view automatically when a table is created with a column defined with a spatial type.

NAME	DATA TYPE	NULL?
table_schema	varchar(128)	NOT NULL
table_name	varchar(128)	NOT NULL
column_name	varchar(128)	NOT NULL
type_schema	varchar(128)	NOT NULL
type_name	varchar(128)	NOT NULL
srs_name	varchar(128)	NULL
srs_id	integer	NULL

Geometry_columns table schema

The ST_GEOMETRY_COLUMNS catalog view stores a record for each geometry column in the database.

The data type for each column is defined below.

- table_schema (SE_STRING_TYPE) — The owner of the geometry column's table
- table_name (SE_STRING_TYPE) — The geometry column's table name.
- column_name (SE_STRING_TYPE) — The name of the geometry column. The combination of table_schema, table-name, and column_name uniquely identifies the column.
- type_schema (SE_STRING_TYPE) — Schema name to which the declared data type of the spatial column belongs. Obtained from the DB2 catalog.
- srs_name(SE_STRING_TYPE) — Name of the spatial reference system that is associated with the spatial column. If no spatial reference system is associated with the spatial column then SRS_NAME is NULL. A spatial reference system can be associated with a spatial column by using the command "db2gse register_spatial_column" with the appropriate parameters.
- srs_id (SE_INTEGER_TYPE)—Numeric identifier of the spatial reference system that is associated with the spatial column. If no spatial reference system is associated with the column, then SRS_ID is NULL.

Creating a spatial index

Spatial columns contain two-dimensional geographic data, and applications querying those columns require an index strategy that will quickly identify all geometries that lie within a given extent. For this reason DB2 Spatial Extender provides support for the creation of a three-level grid spatial index.

DB2 spatial extender provides a utility, called the *Index ADVISOR* that will analyze the spatial column data and suggest appropriate grid sizes.

For example,

```
Gseidx connect to sde user sde using arcsde
get geometry information
for column sde.valve(shape)
advise grid sizes
analyze 10 percent
```

Please refer to Chapter 11, "Using Indexes and views to access spatial data" in the DB2 Spatial Extender User's Guide and Reference for more details on this utility.

Note also that ArcCatalog and ArcSDE administration tools, sdelayer, shp2sde, and cov2sde, provide support for creating the spatial index.

See Chapter 2 of this book, 'Essential configuring and tuning', for a discussion on selecting the spatial index's grid cell sizes.

Updating statistics

The DB2 optimizer may not use the spatial index unless the statistics on the table are up-to-date. If the spatial index is created after the data has been loaded, the statistics are up-to-date and the optimizer will use the index. However, if the index is created and data is loaded afterwards, the optimizer will not use the spatial index because the statistics will be out of date. To update the statistics use the following SQL.

```
RUNSTATS ON TABLE <table_name> WITH DISTRIBUTION AND DETAILED INDEXES ALL;
```

When updating statistics for ArcSDE feature classes, you should use the tools provided by either ArcCatalog or the update_dbms_stats operation of the ArcSDE administration tool sdetable. For more information on using these tools to update statistics, see Chapter 2, 'Essential configuring and tuning'.

Spatial Extender data types

The *Oxford American Dictionary* defines the noun 'geometry' as "the branch of mathematics dealing with the properties of and relations of lines, angles, surfaces, and solids." On August 11, 1997, the OGC, in its publication of *OpenGIS Features for ODBC (SQL) Implementation Specification*, coined another definition for the noun geometry. The word was selected to define the geometric features that, for the past millennium or more, cartographers have used to map the world. Typically, points represent an object at a single location, linestrings represent a linear characteristic, and polygons represent a spatial extent. An abstract definition of the Open GIS noun geometry might be "a point or aggregate of points symbolizing a feature on the ground". This definition, however, fails to describe the rich set of properties and functionality associated with geometry.

To understand geometry in this context, it is easier to describe it as it has been implemented within the DB2 Spatial Extender as a UDT, like all UDTs in an object relational system geometry, it has a unique set of properties and methods.

Geometry columns as a data type, allow you to define columns that store spatial data. The Geometry data type itself is an abstract noninstantiable superclass, the subclasses of

which are instantiable. An instantiated data type is one that can be defined as a table column and have values of its type inserted into it. A column can be defined as type ST_Geometry, but ST_Geometry values cannot be inserted into it since they cannot be instantiated. Only the subclass values can be inserted into this column because only they can be instantiated. Therefore, the Geometry data type can accept and store any of its subclasses, while its subclass data types can only accept their own values.

Throughout the remainder of this document the term geometry or geometries collectively refers to the superclass called Geometry and all of its subclass data types. Whenever it is necessary to specify the geometry superclass directly, it will be referred to as the Geometry superclass or the Geometry data type.

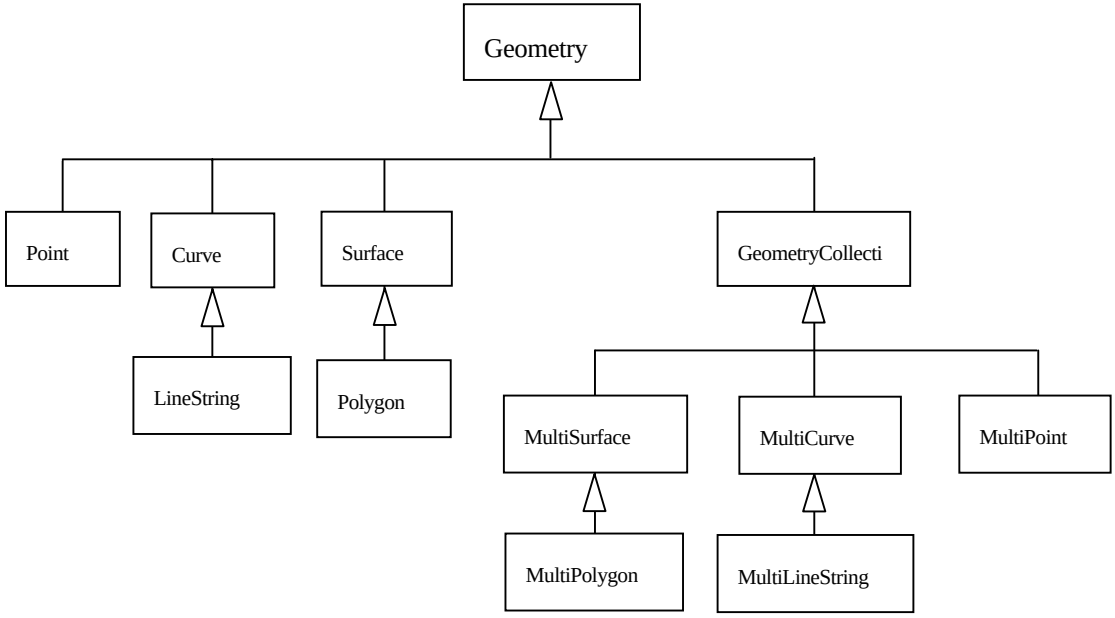


Figure B.1 The hierarchy of the Geometry datatype is divided into the subtypes Point, Curve and Surface simple types and the geometry collections MultiSurface, MultiCurve, and MultiPoint. LineString is the subtype of Curve. Polygon is the subtype of Surface. MultiPolygon is the subtype of MultiSurface. MultiLineString is the subtype of MultiCurve.

Geometry properties

Each subclass inherits the properties of the Geometry superclass but also has properties of its own. Functions that operate on the Geometry data type will accept any of the subclass data types. However, some functions have been defined at the subclass level and will only accept certain subclasses' data types.

Interior, boundary, exterior

All geometries occupy a position in space defined by their interior, boundary, and exterior. The exterior of a geometry is all space not occupied by the geometry. The boundary of a geometry serves as the interface between its interior and exterior. The interior is the space occupied by the geometry. The subclass inherits the interior and exterior properties directly; however, the boundary property differs for each.

The spatial extender ST_Boundary function takes geometry as an input parameter and returns its boundary as a new geometry. The resulting geometry is represented in the spatial reference system of the given geometry.

Simple or nonsimple

Some subclasses of Geometry (LineStrings, MultiPoints, and MultiLineStrings) are either simple or nonsimple. They are simple if they obey all topological rules imposed on the subclass and nonsimple if they "bend" a few. A LineString is simple if it does not intersect its interior. A MultiPoint is simple if none of its elements occupy the same coordinate space. A MultiLineString is simple if none of its element's interiors are intersected by its own interior.

The Spatial Extender ST_IsSimple function or method takes a geometry and returns 1 (TRUE) if the geometry is simple and 0 (FALSE) otherwise.

Empty or not empty

A Geometry is empty if it does not have any points. An empty geometry has a NULL envelope, boundary, interior, and exterior. An empty geometry is always simple and can have z-coordinates or measures. Empty LineStrings and MultiLineStrings have a 0 length. Empty polygons and multipolygons have a 0 area.

The Spatial Extender ST_IsEmpty predicate function takes an ST_Geometry and returns 1 (TRUE) if the ST_Geometry is empty and 0 (FALSE) otherwise.

Number of points

A geometry can have zero or more points. A geometry is considered empty if it has zero points. The point subclass is the only geometry that is restricted to zero or one point; all other subclasses can have zero or more.

Envelope

The envelope of a geometry is the bounding geometry formed by the minimum and maximum (x,y) coordinates. The envelopes of most geometries form a boundary rectangle; however, the envelope of a point is the point since its minimum and maximum coordinates are the same, and the envelope of a horizontal or vertical linestring is a linestring represented by the boundary (the endpoints) of the source linestring.

The Spatial Extender ST_Envelope function takes an ST_Geometry and returns an ST_Geometry that represents the source ST_Geometry's envelope.

Dimension

A geometry can have a dimension of 0, 1, or 2.

The dimensions are

0—Has neither length nor area

1—Has a length

2—Contains area

The point and multipoint subclasses have a dimension of 0. Points represent zero-dimensional features that can be modeled with a single coordinate, while multipoints represent data that must be modeled with a cluster of unconnected coordinates.

The subclasses linestring and multilinestring have a dimension of 1. They store road segments, branching river systems, and any other features that are linear in nature.

Polygon and multipolygon subclasses have a dimension of 2. Forest stands, parcels, water bodies, and other features whose perimeter encloses a definable area can be rendered by either the polygon or multipolygon data type.

Dimension is important not only as a property of the subclass but also plays a part in determining the spatial relationship of two features. The dimension of the resulting feature or features determines whether or not the operation was successful. The dimensions of the features are examined to determine how they should be compared.

The Spatial Extender ST_Dimension function takes a geometry feature and returns its dimension as an integer.

Z-coordinates

Some geometries have an associated altitude or depth. Each of the points that form the geometry of a feature can include an optional z-coordinate that represents an altitude or depth normal to the earth's surface.

The Spatial Extender Is3D predicate function takes a geometry and returns 1 (TRUE) if the function has z-coordinates and 0 (FALSE) otherwise.

Measures

Measures are values assigned to each coordinate. The value represents anything that can be stored as a double-precision number.

The Spatial Extender IsMeasured predicate function takes geometry and returns 1 (TRUE) if it contains measures and 0 (FALSE) otherwise.

Spatial reference system

The spatial reference system identifies the coordinate transformation matrix for each geometry.

The Spatial Extender ST_SRID function takes a geometry and returns its spatial reference identifier as an integer.

Instantiable subclasses

The Geometry data type is not instantiable but instead must store its instantiable subclasses. The subclasses are divided into two categories: the base geometry subclasses and the homogeneous collection subclasses. The base geometries include Point, LineString, and Polygon, while the homogeneous collections include MultiPoint, MultiLineString, and MultiPolygon. As the names imply, the homogeneous collections are collections of base geometries. In addition to sharing base geometry properties, homogeneous collections have some of their own properties as well.

The Spatial Extender ST_GeometryType function takes a geometry and returns the instantiable subclass in the form of a character string. The Spatial Extender ST_NumGeometries function takes a homogeneous collection and returns the number of base geometry elements it contains. The Spatial Extender ST_GeometryN function takes a homogeneous collection and an index and returns the nth base geometry.

ST_Point

An ST_Point is a zero-dimensional geometry that occupies a single location in coordinate space. An ST_Point has a single x,y coordinate value. An ST_Point is always simple and has a NULL boundary. It is used to define features such as oil wells, landmarks, and elevations.

Spatial Extender functions that operate solely on the ST_Point data type include ST_X, ST_Y, ST_Z, and ST_M.

The ST_X function returns a point data type's x coordinate value as a double-precision number.

The ST_Y function returns a point data type's y coordinate value as a double-precision number.

The ST_Z function returns a point data type's z coordinate value as a double-precision number.

The ST_M function returns a point data type's m coordinate value as a double-precision number.

ST_LineString

An ST_LineString is a one-dimensional object stored as a sequence of points defining a linear interpolated path. The ST_LineString is simple if it does not intersect its interior. The endpoints (the boundary) of a closed ST_LineString occupy the same point in space. An ST_LineString is a ring if it is both closed and simple. As well as the other properties inherited from the superclass ST_Geometry, ST_LineStrings have length. ST_LineStrings are often used to define linear features such as roads, rivers, and power lines.

The endpoints normally form the boundary of an ST_LineString unless the ST_LineString is closed, in which case the boundary is NULL. The interior of an ST_LineString is the connected path that lies between the endpoints, unless it is closed, in which case the interior is continuous.

Spatial Extender functions that operate on ST_LineStrings include ST_StartPoint, ST_EndPoint, ST_PointN, ST_Length, ST_NumPoints, ST_IsRing, and ST_IsClosed.

The ST_StartPoint function takes an ST_LineString and returns its first point.

The ST_EndPoint function takes an ST_LineString and returns its last point.

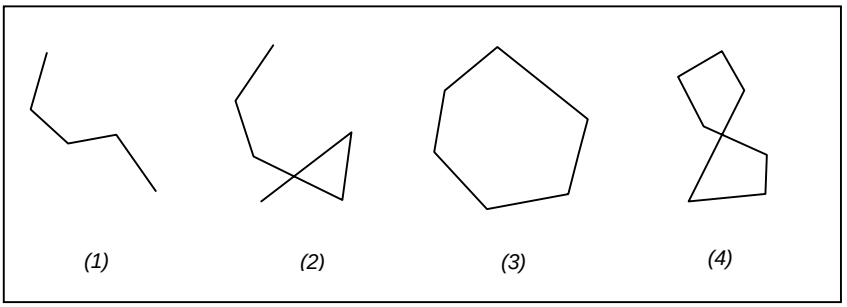
The ST_PointN function takes an ST_LineString and an index to an nth point and returns that point.

The ST_Length function takes an ST_LineString and returns its length as a double-precision number.

The ST_NumPoints function takes an ST_LineString and returns the number of points in its sequence as an integer.

The ST_IsRing predicate function takes an ST_LineString and returns 1 (TRUE) if the ST_LineString is a ring and 0 (FALSE) otherwise.

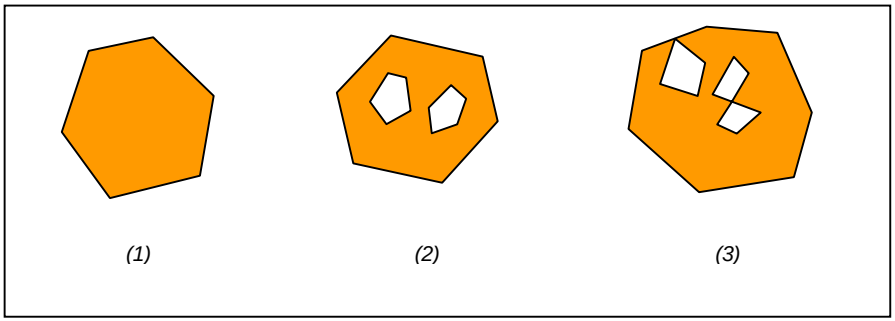
The ST_IsClosed predicate function takes an ST_LineString and returns 1 (TRUE) if the ST_LineString is closed and 0 (FALSE) otherwise.



Examples of ST_LineString objects: (1) a simple nonclosed ST_LineString, (2) a nonsimple nonclosed ST_LineString, (3) a closed simple ST_LineString and therefore is a ring, and (4) a closed nonsimple ST_LineString and is not a ring.

ST_Polygon

An ST_Polygon is a two-dimensional surface stored as a sequence of points defining its exterior bounding ring and 0 or more interior rings. ST_Polygon, by definition, is always simple. Most often ST_Polygon defines parcels of land, water bodies, and other features having spatial extent.



Examples of ST_Polygon objects: (1) an ST_Polygon whose boundary is defined by an exterior ring; (2) an ST_Polygon whose boundary is defined by an exterior ring and two interior rings, and the area inside the interior rings is part of the ST_Polygon's exterior; and (3) a legal ST_Polygon because the rings intersect at a single tangent point.

The exterior and any interior rings define the boundary of an ST_Polygon, and the space enclosed between the rings defines the ST_Polygon's interior. The rings of an ST_Polygon can intersect at a tangent point but never cross. In addition to the other properties inherited from the superclass ST_Geometry, ST_Polygon has area.

Spatial Extender functions that operate on ST_Polygon include ST_Area, ST_ExteriorRing, ST_NumInteriorRing, ST_InteriorRingN, ST_Centroid, and ST_PointOnSurface.

The ST_Area function takes an ST_Polygon and returns its area as a double-precision number.

The ST_ExteriorRing function takes an ST_Polygon and returns its exterior ring as an ST_LineString.

The ST_NumInteriorRing takes an ST_Polygon and returns the number of interior rings that it contains.

The ST_InteriorRingN function takes an ST_Polygon and an index and returns the nth interior ring as an ST_LineString.

The ST_Centroid function takes an ST_Polygon and returns an ST_Point that is the center of the ST_Polygon's envelope.

The ST_PointOnSurface function takes an ST_Polygon and returns an ST_Point that is guaranteed to be on the surface of the ST_Polygon.

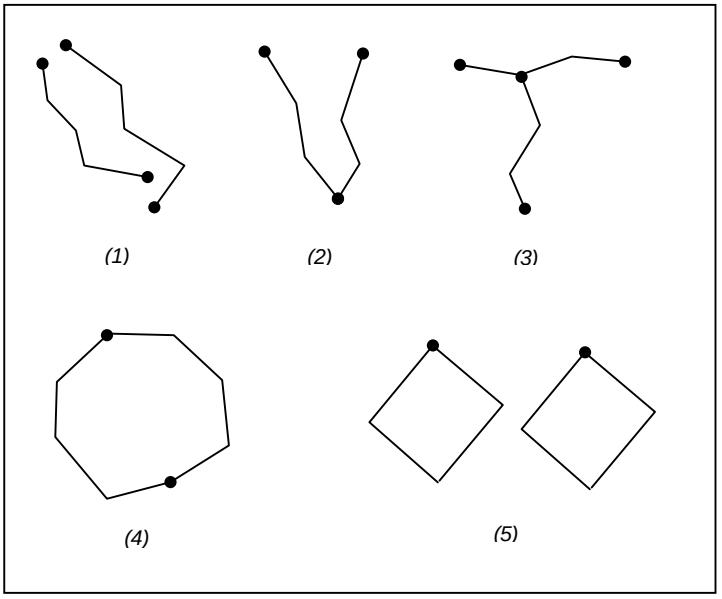
ST_MultiPoint

An ST_MultiPoint is a collection of ST_Points and, just like its elements, it has a dimension of 0. An ST_MultiPoint is simple if none of its elements occupy the same coordinate space. The boundary of an ST_MultiPoint is NULL. ST_MultiPoints define aerial broadcast patterns and incidents of a disease outbreak.

ST_MultiLineString

An ST_MultiLineString is an collection of ST_LineStrings. ST_MultiLineStrings are simple if they only intersect at the endpoints of the ST_LineString elements. ST_MultiLineStrings are nonsimple if the interiors of the ST_LineString elements intersect.

The boundary of an ST_MultiLineString is the nonintersected endpoints of the ST_LineString elements. The ST_MultiLineString is closed if all its ST_LineString elements are closed. The boundary of an ST_MultiLineString is NULL if all the endpoints of all the elements are intersected. In addition to the other properties inherited from the superclass ST_Geometry, ST_MultiLineStrings have length. ST_MultiLineStrings are used to define streams or road networks.



Examples of ST_MultiLineStrings: (1) a simple ST_MultiLineString whose boundary is the four endpoints of its two ST_LineString elements; (2) a simple ST_MultiLineString because only the endpoints of the ST_LineString elements intersect. The boundary is two nonintersected endpoints; (3) a nonsimple ST_MultiLineString because the interior of one of its ST_LineString elements is

intersected. The boundary of this ST_MultiLineString is the three nonintersected endpoints; (4) a simple nonclosed ST_MultiLineString. It is not closed because its element ST_LineStrings are not closed. It is simple because none of the interiors of any of the element ST_LineStrings intersect; (5) a simple closed ST_MultiLineString. It is closed because all its elements are closed. It is simple because none of its elements intersect at the interiors.

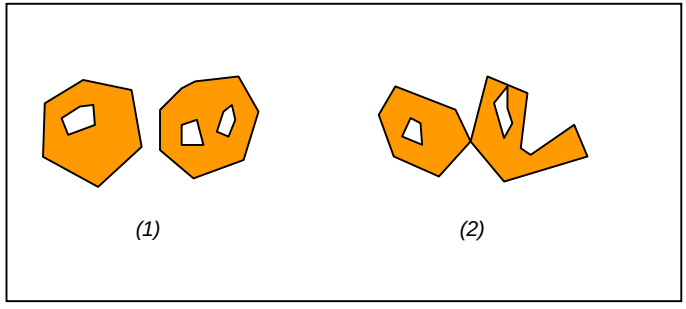
Spatial Extender functions that operate on ST_MultiLineStrings include ST_Length and ST_IsClosed.

The ST_Length function takes an ST_MultiLineString and returns the cumulative length of all its ST_LineString elements as a double-precision number.

The ST_IsClosed predicate function takes an ST_MultiLineString and returns 1 (TRUE) if the ST_MultiLineString is closed and 0 (FALSE) otherwise.

ST_MultiPolygon

The boundary of an ST_MultiPolygon is the cumulative length of its elements' exterior and interior rings. The interior of an ST_MultiPolygon is defined as the cumulative interiors of its element ST_Polygons. The boundary of an ST_MultiPolygon's elements can only intersect at a tangent point. In addition to the other properties inherited from the superclass ST_Geometry, ST_MultiPolygons have area. ST_MultiPolygons define features such as a forest stratum or a noncontiguous parcel of land such as a Pacific island chain.



Examples of ST_MultiPolygon: (1) an ST_MultiPolygon with two ST_Polygon elements. The boundary is defined by the two exterior rings and the three interior rings; and (2) an ST_MultiPolygon with two ST_Polygon elements. The boundary is defined by the two exterior rings and the two interior rings. The two ST_Polygon elements intersect at a tangent point.

Spatial Extender functions that operate on ST_MultiPolygons include ST_Area, ST_Centroid, and ST_PointOnSurface.

The ST_Area function takes an ST_MultiPolygon and returns the cumulative ST_Area of its ST_Polygon elements as a double-precision number.

The ST_Centroid function takes an ST_MultiPolygon and returns an ST_Point that is the center of an ST_MultiPolygon's envelope.

The ST_PointOnSurface function takes an ST_MultiPolygon and returns an ST_Point that is guaranteed to be normal to the surface of one of its ST_Polygon elements.

APPENDIX C

Storing locators

A locator is an object that you can use to convert textual descriptions of locations into geographic features. The most common locator is an address locator, which you can use to geocode addresses. For additional documentation on creating and using locators in ArcGIS, see *Geocoding in ArcGIS* in the ArcGIS documentation set.

ArcSDE stores locator definitions in the SDE_locators table. Three main types of locators can be stored in an ArcSDE database:

- Locator styles are used as templates on which to base new locators.
- Locators define the inputs, outputs, the logic, and one or more reference datasets that are used to find locations. Locators are usually created by adding some properties to a locator style that specify which reference datasets and which columns in those reference datasets to use to find locations. Using ArcCatalog to create a locator based on a locator style is the easiest way to create a new locator.
- Attached locators are copies of locators that are used to create a geocoded feature class. When you create a geocoded feature class by geocoding a table of addresses using an address locator, ArcSDE stores a copy of the locator that was used to create the geocoded feature class. ArcSDE uses this attached locator when you rematch addresses in the geocoded feature class.

Each locator style, locator, and attached locator has a number of properties that define the locator. ArcSDE stores each property of a locator as a record in the SDE_metadata table.

Address locators use a set of geocoding rules that define how addresses are parsed, standardized, and matched to the reference data used by the address locator. ArcSDE

stores geocoding rules in the GCDRULES table. Each row in the GCDRULES table corresponds to a single file in a set of geocoding rules. For information on geocoding rule files, see the *Geocoding Rule Base Developer Guide* in the ArcGIS documentation set.

Many address locators require a geocoding index table for each reference data table. Geocoding index tables are tables used by a locator to quickly search for records in the corresponding reference datasets that may be matches for an address. The XID column in a geocoding index table is a foreign key to the OBJECTID column in the corresponding reference dataset. When you create a new address locator that requires a geocoding index table for a reference dataset, ArcSDE creates the geocoding index table if it does not already exist.

When a locator is instantiated, ArcSDE reads the locator record from the SDE_locators table, and all of the corresponding locator properties from the SDE_metadata table. Some of the locator properties specify which set of geocoding rules to use, which are read from the GCDRULES table. Other locator properties specify which feature classes or tables in the ArcSDE database are used as reference datasets, and which geocoding index tables, if any, correspond to these reference datasets.

When you use a locator to geocode an address, the locator uses the specified geocoding rules to parse the given address into its components. If the locator uses geocoding index tables to index the reference data, the locator properties specify which of these address components to use to search for matches in the geocoding index table(s), and which transformations (usually the Soundex function) to apply to the address components when searching for records in the geocoding index table. ArcSDE searches for records matching the geocoding index query in the geocoding index table. The resulting set of records from the geocoding index table is joined to the corresponding reference data table to generate a set of candidates for the address. ArcSDE uses the locator's properties to determine which columns in the reference data feature class or table correspond to address components used by the locator, and uses the geocoding rules to assign a score to each candidate.

Locator schema

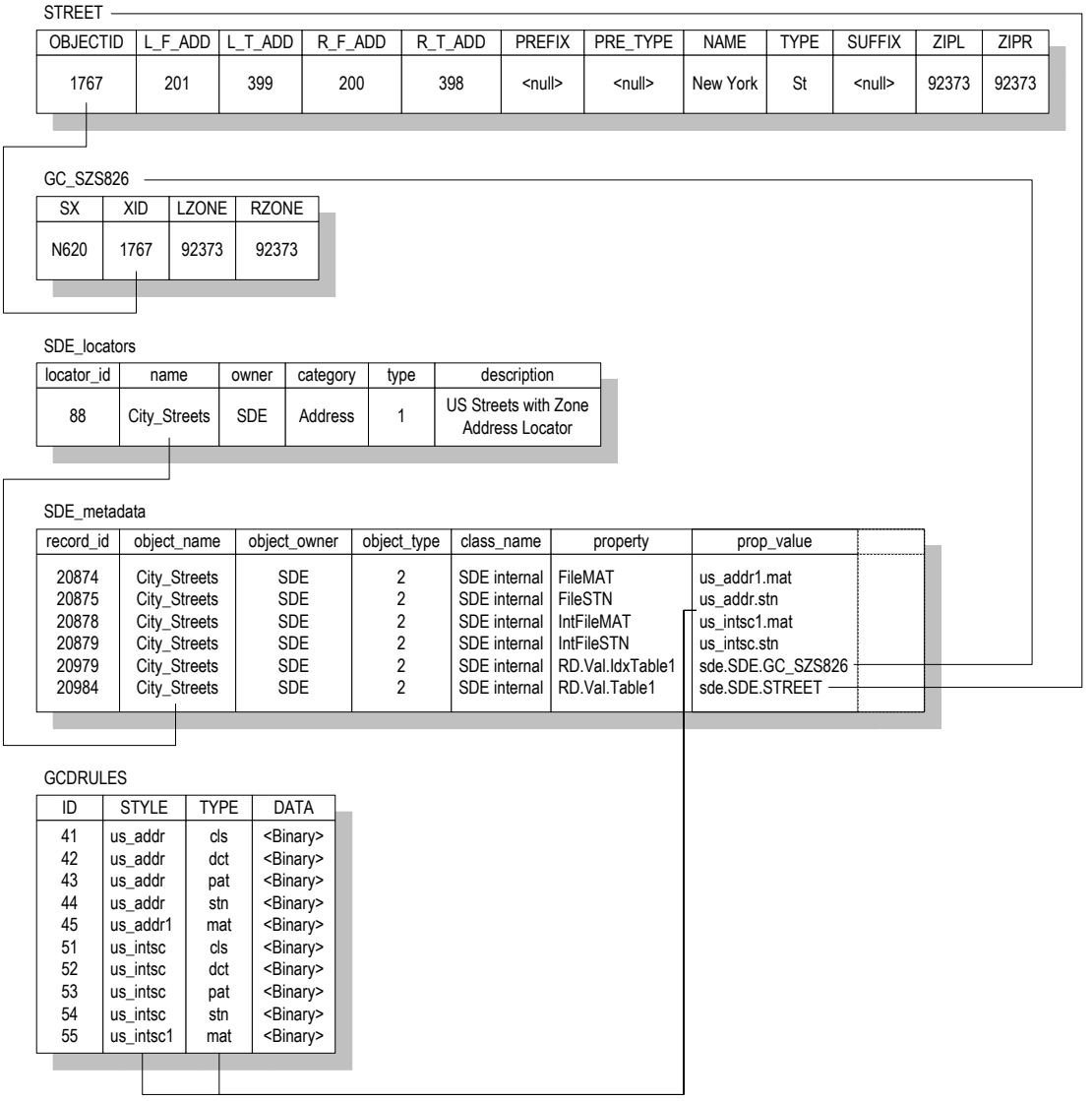
When you create a locator in an ArcSDE database, ArcSDE adds a record to the SDE_locators table that defines the locator. ArcSDE also adds a record to the SDE_metadata table for each property of the locator. The object_name column in the SDE_metadata table is a foreign key to the Name column in the SDE_locators table that ArcSDE uses to associate locators with their properties.

Each locator has associated FileMAT and FileSTN properties in the SDE_metadata table that define which geocoding rules the locator uses. The values of these properties are in

the format *style.type*, and define which geocoding rule files, stored in the GCDRULES table, the locator uses to match addresses. The locator uses the value of these properties in the SDE_metadata table to query the GCDRULES table on the STYLE and TYPE columns to retrieve the correct set of geocoding rules. Locators that support intersection geocoding have associated IntFileMAT and IntFileSTN properties that define the geocoding rules to use for intersection geocoding.

When you create an address locator, ArcSDE may create one or more geocoding index tables for the reference datasets used by the locator, depending upon the locator style on which the address locator is based. Geocoding index table names are prefixed with "GC_", and include characters identifying the type of geocoding index table, and the Geodatabase object class ID of the table or feature class that it indexes. The XID column in a geocoding index table is a foreign key to the OBJECTID column in the table or feature class that the geocoding index table indexes.

In the example that follows, an ArcSDE database contains a STREET feature class that represents street centerlines for a particular geographic area, such as a city. In addition to the geometry for the street centerlines, the STREET feature class contains attributes for the address ranges that can be found along the street, and the components of the street name. The ArcSDE table schema required to store a locator to allow address geocoding on this feature class is described here.



STREET

OBJECTID	L_F_ADD	L_T_ADD	R_F_ADD	R_T_ADD	PREFIX	PRE_TYPE	NAME	TYPE	SUFFIX	ZIPL	ZIPR
1767	201	399	200	398	<null>	<null>	New York	St	<null>	92373	92373

GC_SZS826

SX	XID	LZONE	RZONE
N620	1767	92373	92373

SDE_locators

locator_id	name	owner	category	type	description
88	City_Streets	SDE	Address	1	US Streets with Zone Address Locator

SDE_metadata

record_id	object_name	object_owner	object_type	class_name	property	prop_value
20874	City_Streets	SDE	2	SDE internal	FileMAT	us_addr1.mat
20875	City_Streets	SDE	2	SDE internal	FileSTN	us_addr.stn
20878	City_Streets	SDE	2	SDE internal	IntFileMAT	us_intsc1.mat
20879	City_Streets	SDE	2	SDE internal	IntFileSTN	us_intsc.stn
20979	City_Streets	SDE	2	SDE internal	RD.Val.IdxTable1	sde.SDE.GC_SZS826
20984	City_Streets	SDE	2	SDE internal	RD.Val.Table1	sde.SDE.STREET

GCDRULES

ID	STYLE	TYPE	DATA
41	us_addr	cls	<Binary>
42	us_addr	dct	<Binary>
43	us_addr	pat	<Binary>
44	us_addr	stn	<Binary>
45	us_addr1	mat	<Binary>
51	us_intsc	cls	<Binary>
52	us_intsc	dct	<Binary>
53	us_intsc	pat	<Binary>
54	us_intsc	stn	<Binary>
55	us_intsc1	mat	<Binary>

Business table

In this example, the STREET feature class represents street centerlines within a particular geographic area, and contains attributes that allow address locators to geocode addresses using this feature class. By default, ArcSDE stores geometry for feature

classes in a separate feature table in the ArcSDE compressed binary format, which is described in Appendix A.

NAME	DATA TYPE	NULL?
OBJECTID	INT(4)	NOT NULL
L_F_ADD	INT(4)	NULL
L_T_ADD	INT(4)	NULL
R_F_ADD	INT(4)	NULL
R_T_ADD	INT(4)	NULL
PREFIX	VARCHAR(2)	NULL
PRE_TYPE	VARCHAR(5)	NULL
NAME	VARCHAR(30)	NULL
TYPE	VARCHAR(5)	NULL
SUFFIX	VARCHAR(2)	NULL
ZIPL	VARCHAR(5)	NULL
ZIPR	VARCHAR(5)	NULL
Shape	INT(4)	NULL

STREET business table

- OBJECTID (SE_INTEGER_TYPE) – the table's primary key
- L_F_ADD (SE_INTEGER_TYPE) – the address at the start node on the left side of the street feature
- L_T_ADD (SE_INTEGER_TYPE) – the address at the end node on the left side of the street feature
- R_F_ADD (SE_INTEGER_TYPE) – the address at the start node on the right side of the street feature
- R_T_ADD (SE_INTEGER_TYPE) – the address at the end node on the right side of the feature
- PREFIX (SE_STRING_TYPE) – the prefix direction component of the street's name
- PRE_TYPE (SE_STRING_TYPE) – the prefix type component of the street's name
- NAME (SE_STRING_TYPE) – the base component of the street's name
- TYPE (SE_STRING_TYPE) – the suffix type component of the street's name

- SUFFIX (SE_STRING_TYPE) – the suffix direction component of the street's name
- ZIPL (SE_STRING_TYPE) – the ZIP code on the left side of the street feature
- ZIPR (SE_STRING_TYPE) – the ZIP code on the right side of the street feature
- Shape (SE_INTEGER_TYPE) – a foreign key to the feature table containing the geometry for the feature class

Geocoding index table (GC_SZS<objectclass_id>)

When you create a locator that uses an ArcSDE feature class as reference data, the locator style on which the locator is based may specify that a geocoding index table is used when performing geocoding queries against the feature class. The locator style defines the format of the name of the geocoding index table, as well as the contents. In this example, a locator based on the "US Streets with Zone" locator style was created on the STREETS feature class. Geocoding index tables created by locators based on this style contain a Soundex value for the street name, as well as attributes for the zones on each side of the street feature.

The size of the delta tables also depends on how often records are removed. These tables shrink only when the states preceding the level 0 version are compressed. This occurs only after a version branching directly off the root of the version tree completes and is removed from the system. The compression of states that follows will cause the changes of the states between the level 0 version and the next version following the one removed to be written to the business table and deleted from the delta tables.

NAME	DATA TYPE	NULL?
SX	VARCHAR(4)	NULL
XID	INT(4)	NULL
LZONE	VARCHAR(5)	NULL
RZONE	VARCHAR(4)	NULL

Geocoding index table

- SX (SE_STRING_TYPE) – the Soundex value for the street name
- XID (SE_INTEGER_TYPE) – a foreign key to the OBJECTID column in the business table
- LZONE (SE_STRING_TYPE) – the zone on the left side of the street feature

- RZONE (SE_STRING_TYPE) – the zone on the right side of the street feature

SDE_locators table

When you add a locator to an ArcSDE database, ArcSDE adds a row to the SDE_locators table. Each row in the SDE_locators table defines a locator or locator style.

NAME	DATA TYPE	NULL?
locator_id	INT(4)	NOT NULL
Name	VARCHAR(32)	NOT NULL
Owner	VARCHAR(32)	NOT NULL
Category	VARCHAR(32)	NOT NULL
Type	INT(4)	NOT NULL
Description	VARCHAR(64)	NULL

SDE locators table

- locator_id (SE_INTEGER_TYPE) – the table's primary key
- name (SE_STRING_TYPE) – the name of the locator
- owner (SE_STRING_TYPE) – the name of the ArcSDE user that owns the locator
- category (SE_STRING_TYPE) – the category of the locator; address locators have a category value of "Address"
- type (SE_INTEGER_TYPE) – the type of locator; values in this column are represented as follows:
- 0 – define locator styles
- 1 – define locators (i.e., locators that can be used to find locations)
- 2 – define attached locators (i.e., locators that are attached to a geocoded feature class, and are a copy of the locator and the geocoding options that were used to create the geocoded feature class)
- description (SE_STRING_TYPE) – the description of the locator

SDE_metadata table

When you add a locator to an ArcSDE database, ArcSDE adds a row to the SDE_metadata table for each property of the locator. Each row in the SDE_metadata table defines a single property for a locator. The object_name column is a foreign key to the name column in the SDE_locators table that ArcSDE uses to associate a locator with its properties.

NAME	DATA TYPE	NULL?
record_id	INT(4)	NOT NULL
object_database	VARCHAR(32)	NULL
object_name	VARCHAR(160)	NULL
object_owner	VARCHAR(32)	NOT NULL
object_type	INT(4)	NOT NULL
class_name	VARCHAR(32)	NULL
property	VARCHAR(32)	NULL
prop_value	VARCHAR(255)	NULL
description	VARCHAR(65)	NULL
creation_date	DATETIME(8)	NOT NULL

SDE metadata table

- record_id (SE_INTEGER_TYPE) – the table's primary key
- object_database (SE_STRING_TYPE) – the ArcSDE database in which the described object is storedS; not used for locator properties
- object_name (SE_STRING_TYPE) – the name of the locator to which the property belongs
- object_owner (SE_STRING_TYPE) – the name of the ArcSDE user that owns the record
- object_type (SE_INTEGER_TYPE) – always a value of 2 for locator properties
- class_name (SE_STRING_TYPE) – always a value of "SDE_internal" for locator properties
- property (SE_STRING_TYPE) – the name of the locator property
- prop_value (SE_STRING_TYPE) – the value of the locator property
- description (SE_STRING_TYPE) – not used for locator properties

- creation_date (SE_DATE_TYPE) – the date and time at which the locator property was created

GCDRULES table

The GCDRULES table stores the geocoding rules that are used by address locators to match addresses. Each record in the GCDRULES table corresponds to a geocoding rule file. For descriptions of each of the geocoding rule files and their contents, see the *Geocoding Rule Base Developer Guide* in the ArcGIS documentation set.

NAME	DATA TYPE	NULL?
ID	INT(4)	NOT NULL
STYLE	VARCHAR(32)	NULL
TYPE	VARCHAR(3)	NULL
DATA	image	NULL

Geocoding rules table

- ID (SE_INTEGER_TYPE) – the table's primary key
- STYLE (SE_STRING_TYPE) – the name of the geocoding rule set
- TYPE (SE_STRING_TYPE) – the type of geocoding rule file
- DATA (SE_BLOB_TYPE) – the contents of the geocoding rule file

APPENDIX D

Making a direct connection

Direct connect is another configuration option for ArcSDE and all the ArcSDE concepts and prerequisites also apply to direct connect. The main difference between the ArcSDE application server and direct connect is where the ArcSDE processing takes place. This purpose of this appendix is to provide administrators information on how to setup and configure direct connect configurations for the database as well as client machines. If using the application server exclusively, you do not need this appendix.

What files do you need?

There are two sets of ESRI-supplied files required for direct connect:

1. Direct connect drivers. These are dynamically linked libraries in the bin or lib directory (depending on your operating system) of your client application that provide the functionality to connect and use spatial data in a DBMS. There are drivers for the following databases:

- IBM® DB2
- IBM Informix®
- Microsoft® SQL Server™
- Oracle® 8*i* and 9*i*

These drivers are automatically installed for ArcGIS (the whole product suite), ArcView GIS 3.x Database Access, ArcIMS®, ArcInfo workstation and MapObjects 2. If you are using a non-ESRI custom application built from the ArcSDE C API, you may need to

install the direct connect drivers from the ArcSDE Developer Kit CD-ROM located in the ArcSDE media kit. Check with the supplier of your non-ESRI custom application.

2. Database setup files. These are files needed by an administrator to setup and configure a DBMS for direct connect and include files like sdesetup<dbms>. The setup is exactly the same as it is for the ArcSDE application server. These setup files are located on the platform CD-ROM of choice in the ArcSDE media kit. To get them, you must install ArcSDE for your database. You do not have to create an application server; you only need the files on disk so you can use them against your database.

DBMS considerations are as follows:

- **Oracle*8i*™, Oracle*9i*™**

To facilitate network communication to an Oracle database, each client machine where direct connect is used must have Oracle Net installed.

- **Microsoft SQL Server 7, Microsoft SQL Server 2000**

SQL Server requires Microsoft Data Access Components (MDAC).

If you intend to use ArcCatalog 9.0 or ArcView GIS 3.3 with Database Access 2.1f, MDAC version 2.6(SP1) or greater is required. If using ArcIMS 9.0 or ArcGIS 9.0 to direct connect, you must have MDAC 2.6 or higher.

- **DB2**

Each client machine must be configured for remote database access. Use the DB2 Configuration Assistant on the database host to connect to a remote database.

- **Informix**

Each client machine where direct connect will be used must have the Informix Client SDK 2.8 or the Informix I-connect 2.8 application installed. The client machine must also have the SetNet32 application installed, which comes with both the Informix Client SDK 2.8 and the Informix I-connect 2.8 applications.

How to get your database setup files

You will need to get your database setup files from one of the CD-ROM's in the ArcSDE media kit. The ArcSDE media kit has CD-ROM's by platform with the exception of the ArcSDE Developer Kit CD-ROM. To get your database setup files, you will need to install the software for the ArcSDE application server for your database/platform. For example, if you are using IBM DB2 on a Sun Solaris server, you

will select the Sun Solaris CD-ROM from the ArcSDE media kit and install the DB2 version of ArcSDE on your Sun Solaris server. Please be sure to follow the post installation configuration instructions in the database specific install guide but ignore any instructions about creating the application server. You don't need to do that. Install guides are html files on each CD-ROM. Please read them carefully.

Why do I need to install the ArcSDE application server software?

Installation of the ArcSDE application server is to get the database setup and administration files only. If you are a direct connect only site, you do not need to start an ArcSDE application server. All you need to do is install the ArcSDE files to disk and then follow the post installation configuration instructions. The administration files that get installed (eg: sdesetup<dbms>, sdeconfig, sdedbtune, sdelayer) are useful for managing your connection parameters, dbtune table and manual registration/unregistration of 3rd party layers. Please see the *Managing ArcSDE Services* book and the *ArcSDE Configuration and Tuning Guides* for more information.

If you use both the application server and direct connect at your site, you already have or soon will have ArcSDE setup and administration files installed anyway. It is important to note that once your database is configured for use with the ArcSDE application server, it is also ready for direct connect usage.

Environment variables

For each client machine, there are environment variables you must set. If necessary, ask your Windows or UNIX system administrator to find out how to set environment variables on your systems.

The SDEHOME environment variable

You must set the SDEHOME variable to tell the client application:.

- Where the direct connect driver files are stored. For ESRI client applications, the direct connect files are located in the same directory where the client application's other dynamicly linked library files get installed. For Windows applications, this is normally in the bin directory of your client applications install location. For UNIX and Linux systems, these will normally be in the lib directory.

To set this environment variable, you must specify the full file path for it. For example,

Unix: setenv SDEHOME /unix1/arcgis/

Windows: use Windows utilities to set a variable to something like this

```
Variable:   Value:
SDEHOME     C:\Program Files\ArcGIS\
```

The direct connect process will "look" for the appropriate driver in the bin or lib directories of the path specifiied.

You do not have to set the SDEHOME environment variable if the following are true:

- Your users are using ESRI client applications built with the ArcSDE 9.0 C API (a list of these applications is in Chapter 1, 'Introducing direct connect')

- Your users are not using UNIX

Unix or Linux systems

1. Include $SDEHOME/lib in the library environment variable for your platform.

   If your database is an Oracle database, include $ORACLE_HOME/lib as well.

   For example:

   setenv LD_LIBRARY_PATH $SDEHOME/
   lib:$ORACLE_HOME/lib:/usr/ openwin/lib:/usr/lib

2. Add the bin directory to the system path:and

   An example follows for the SDEHOME variable.

   setenv PATH $JAVA_HOME/bin:$SDEHOME/
   bin:$AEJHOME/bin:/usr/sbin:/usr/bin:/usr/local/ bin:
/etc:/usr/ucb:/usr/dt/bin:/usr/bin/X11

3. If ArcIMS is your client application and Oracle is the database, append $ORACLE_HOME/lib to the LD_LIBRARY_PATH variable in the aimsappsrvr and aimsmonitor scripts, located in the $AIMSHOME/Xenv directory.

   For example, where your LD_LIBRARY_PATH variable now reads:

   LD_LIBRARY_PATH-$AIMSHOME/lib:$AIMSHOME/bin;
   export LD_LIBRARY_PATH

   It should now be:

LD_LIBRARY_PATH-$AIMSHOME/lib:$AIMSHOME/
bin:$ORACLE_HOME/lib; export LD_LIBRARY_PATH

The ETC directory

If an etc directory exists for the client application, it must be located in the directory you specified for SDEHOME. If it isn't located there, you must create it there. This etc directory is where the log file of error messages will be stored by default.

The dbinit.sde file

This file is located in the etc directory of your SDEHOME. This file can be u sed to set environment variables for direct connect use. It may be more convenient to set environment variables for direct connect here than via system tools.

See Chapter 3 in *Managing ArcSDE Application Servers* for more information on the dbinit.sde file.

Client/database compatibility

Direct connect drivers are only compatable with a same-vintage database configured for ArcSDE. For example, you cannot direct connect from ArcMap 9.0 to a a database that is still at an 8.3 configuration. You would have to run the 9.0 setup configuration on that 8.3 database to be able to use direct connect from the ArcMap 9.0 client.

Registration and authorization

ArcSDE application servers and all direct connect configurations must be registered before use. The end result of the registration process is an authorization file that is used to enable the software for use. Please note that if you are an existing ArcSDE user, your ArcSDE 8.x keycode will not work with 9.0. To register in the United States, go http://service.esri.com. If you are not in the United States, please call your local distributor to register your software. If the Internet is not an option, you can contact ESRI Customer Service or your local distributor to register and receive your 9.0 authorization file.

Setting up clients for DB2 direct connect

Set up the database

You must set up and configure each database that users will be direct connecting to. Use standard DB2 tools , ArcSDE tools and documentation to

1. install the application server software

2. perform the post installation configuration (application server start up is not required for direct connect)

When your database is configured and authorized for ArcSDE, you are ready to set up your client machines.

Setting up the client machines

When you set up the client machines, you perform the following steps in order on the client machine:

1. If your host database must connect to a remote database, you must use the DB2 Configuration Assistant on the database host to connect to a remote database. The Configuration Assistant comes with DB2 and lets you configure and maintain the database objects that you or your applications will be using.

2. Set environment variables.

3. Create a local user account.

4. Test the connection.

Use the DB2 Configuration Assistant

Note: steps are provided here as a convenience but do not supercede or otherwise replace DB2 documentation. Please refer to DB2's documentation for all information on this topic.

Use the DB2 Configuration Assistant to configure the client to connect directly to a DB2 instance. The Configuration Assistant comes with DB2 and lets you configure and maintain the database objects that you or your applications will be using. It is available as part of the DB2 Administration Client and DB2 Application Development Client.

Each DB2 database that will be accessed must be configured at your DB2 client before you can work with it. You must configure your DB2 clients so they can work with the available objects. From the Configuration Assistant, you can work with existing database objects, add new ones, bind applications, set database manager configuration parameters, and import and export configuration information.

To open the Configuration Assistant in Windows, click Start, point to Programs, click IBM DB2, click Set-up Tools, then click Configuration Assistant. The Configuration Assistant opens. In UNIX, open the Configuration Assistant with the db2ca command.

After the Main panel appears choose Selected in the drop down Menu and select Add Database Using Wizard.

Select how you want to set up a connection. In this menu, you indicate how you will be adding the database that you want to connect to. Each method involves a slightly different set of wizard pages.

Use a profile invokes: Select a database from a profile. Specify an alias for the database. Register this database as a data source.

Search the network invokes: Select a database from the network search result. Specify an alias for the database. Register this database as a data source. Manually configure a connection to a database invokes:

Specify catalog options. (Only appears if the Lightweight Directory Access Protocol (LDAP) is enabled.) Select a communications protocol. Specify communication parameters. (A page tailored to the protocol specified on the previous page.) Specify information for a database on this system. (Only appears if the database is local.) Specify information for a database on a remote system. (Only appears if the database is remote.)

Following is an example choosing the network option. Select Search the network option.

Now choose the Add System button to select a database from the Network.

You can use Discover to retrieve information known about the TCP/IP system and populate the window as shown in the following screen. In the Node name field, specify the cataloged system where the database is located. The node name you choose must not already exist in the node directory or the admin node directory.

In the System name field, specify the physical machine, server system, or workstation where the target database is located. The system name on the server system is defined by the DB2SYSTEM DAS configuration parameter. If the system is not listed, you can issue the db2 get admin cfg command on the server to retrieve its value.

If your network supports TCP/IP, then you can use discovery to help complete the remaining fields on this window. After you select a System name, clicking Discover opens the Discovery Search window where you can select the instance node that you want to add.

Specify an alias for the database.

The next step would be to register the database as an ODBC data source and press Finish.

You should test the database connection that you created. In the Configuration Assistant advanced view: Click on the Databases tab. The Databases page opens. Select the database that you want to work with. From the Selected menu, click Test Connection. The Test Connection notebook opens. Select the type of connections that you want to test. In the User ID field, type a user ID that can connect to the database. Type the password for the User ID in the Password field. If you do not specify a user ID and password, the system password will be used for the connection. Click Test Connection. The Results page opens, displaying the results of the connection test. Optional: Click clear to erase the results.

If the connection is successful you will get the following message.

This process creates an entry in the db2cli.ini file that will look like this. This file resides under the %DB2PATH% dir.

```
[SDEQUART]
DBALIAS=SDEQUART
```

Set environment variables

You must set the SDEHOME and SDE_DATABASE environment variables. Set SDEHOME to point to the directory the client applications .dynamically linked library files are stored.

If your client application is remote (is not running on the same host as the DB2 server), edit the client machine's SDE_DATABASE variable in the dbinit.sde file so that it points to the remote database.

If your client application is local, set the client machine's SDE_DATABASE variable with system tools (do not use the dbinit.sde file to set this) to the name of the DB2 database on the local machine that you want to connect the client to.

Create a local user account

In order for direct connect to work successfully with ArcGIS, a local user with the same user name as specified for the connection to the server must be present on the client machine. This user does not require any special permissions.

To create a local user, please refer to Microsoft's Windows documentation.

For UNIX platforms, please refer to the platform specific commands/procedures (ie. useradd)

Connection syntax

There is a particular syntax to use when connecting with direct connect. For the Service (or instance) value,

```
sde:db2
```

For the Database name, use the alias name specified when setting up the Configuration Assistant. You may also specify the database name in the Service (or instance) value,

```
sde:db2:<db alias name>
```

If the client application is local (running on the same host as the DB2 server), do not specify a value for Server. If the client application is remote, specify a Server value of `remote`.

Test the connection from the client application

Test the connection from the client application you set up to use direct connect.

Index