Converting from LONG RAW to BLOB in an ArcSDE® for Oracle® geodatabase

An ESRI® Technical Paper • May 2007

Converting from LONG RAW to BLOB in an ArcSDE for Oracle geodatabase

An ESRI Technical Paper

Contents

Converting from LONG RAW to BLOB in an ArcSDE for Oracle geodatabase

Oracle has announced the deprecation of the LONG RAW data type with the upcoming 11*g* release of their database software. While it is not known what support will be available for the LONG RAW data type when 11*g* is released, it is possible that LONG RAW columns created prior to the 11*g* release will continue to be supported by the Oracle database. However, some ESRI customers have experienced unexplained ORA-03106 errors while using data stored in LONG RAW data types after upgrading to Oracle 10*g*. This document has been written to assist users who want to convert LONG RAW columns to Binary Large Object (BLOB) columns.

Always perform a backup of the database before making any modifications.

Quick-Start Guide

If you already have an understanding of ArcSDE data storage and Oracle BLOB technology, you will not need to digest the contents of this document in detail. Perhaps you just want a limited set of steps that will guide you through the conversion of the LONG RAW columns in your ArcSDE tables to a BLOB data type. Those steps are provided here with references to the detailed information within this document, should you need to review it.

1. First, estimate the storage space that will be required to store the tables and large object (LOB) segments after they have been converted. To obtain an accurate estimate, install and execute the stored procedures of the *estimate_blob_storage* PL/SQL package included with this document. For information on how to install and use the *estimate_blob_storage* package, refer to the section Using *estimate_blob_storage*.

2. Based on the space estimates from the previous step, extend existing tablespaces or create new tablespaces that will store the tables containing the BLOB columns and the LOB segments. Note that the LOB index is typically not used and, therefore, will not store data. However, be aware that it will always be created with the LOB segment and occupy the initial space allocation. For a better understanding of how binary data is stored in the BLOB column, refer to the section Understanding the Oracle BLOB Data Type in this paper.

3. Update the DBTUNE parameters to create and store binary data in BLOB columns rather than LONG RAW columns. The following DBTUNE storage parameters are listed with their BLOB storage values:

```
ATTRIBUTE_BINARY BLOB
GEOMETRY_STORAGE SDELOB
RASTER_STORAGE BLOB

   B_STORAGE "PCTFREE 0 INITRANS 4
              TABLESPACE BIZZTABS
              LOB (DOCUMENT) STORE AS
              (TABLESPACE BIZZ_LOB_SEGMENT
               CACHE PCTVERSION 0)"

   A_STORAGE "PCTFREE 0 INITRANS 4
              TABLESPACE BIZZTABS
              LOB (DOCUMENT) STORE AS
              (TABLESPACE BIZZ_LOB_SEGMENT
               CACHE PCTVERSION 0)"

   F_STORAGE "PCTFREE 0 INITRANS 4
              TABLESPACE VECTOR
              LOB (POINTS) STORE AS
              (TABLESPACE VECTOR_LOB_SEGMENT
               CACHE PCTVERSION 0)"

   BLK_STORAGE "PCTFREE 0 INITRANS 4
                TABLESPACE RASTER
                LOB (BLOCK_DATA) STORE AS
                (TABLESPACE RASTER_LOB_SEGMENT
                 CACHE PCTVERSION 0)"

   AUX_STORAGE “PCTFREE 0 INITRANS 4
                TABLESPACE AUXTABS
                LOB (OBJECT) STORE AS
                (TABLESPACE AUXTABS
                 CACHE PCTVERSION 0)”
```

In the above parameters, you will need to substitute the names of your tablespaces and add the additional Oracle storage parameters that are unique to your installation.

Refer to the Modifying the DBTUNE Parameters to Store BLOB Columns section for a more detailed explanation of these parameters. Also refer to the

ArcGIS® online help topics: The dbtune file and the DBTUNE table, DBTUNE configuration keywords, and DBTUNE configuration parameter names-configuration string pairs. The help can be accessed from the ESRI support site, http://support.esri.com.

4. Convert the LONG RAW columns to BLOB. Install and execute the stored procedures of the *sde_metadata* and *convert_lr_to_blob* PL/SQL packages. Refer to the section Using *convert_lr_to_blob* for more information.

What ArcSDE Data Is Stored in a LONG RAW Column

LONG RAW and BLOB are Oracle data types that store binary data. Binary data ranges from an executable program, a raster image, a compressed geometry string, or even readable text such as an XML document. The forms of binary data stored by ArcSDE are vector, raster, XML, and user defined.

The vector data—otherwise known as geometry or shape data—consists of a compressed string of vertices that are stored in a feature class. The size of the vector data relies on its type. It can be as small as a single point, which will occupy approximately 12 bytes, or exceed many kilobytes such as a multipolygon covering a large area of interest.

An ArcSDE 9.2 feature class may be stored in one of five different storage formats: the SDEBINARY, SDELOB, ST_GEOMETRY, OGCWKB, or SDO_GEOMETRY. The SDEBINARY format stores a LONG RAW data type and has been the traditional default storage type.

Feature classes stored as SDEBINARY have a feature table with a POINTS column defined as LONG RAW.

At ArcSDE 9.2, raster data can be stored as either a LONG RAW, BLOB, or SDO_GEORASTER data type. Prior to ArcSDE 9.2, the default storage type for raster data was LONG RAW. However, with the 9.2 release, the default storage was changed to BLOB.

Whenever raster columns are stored as LONG RAW, the raster blocks table contains a BLOCK_DATA column and a raster auxiliary table contains an OBJECT column that are both defined as LONG RAW.

ArcSDE also stores XML data in a BLOB column; however, since you have never been able to store this data as LONG RAW, you do not need to worry about converting it.

ArcSDE allows you to add your own binary columns to the business tables of table classes, feature classes, and raster catalogs. Prior to ArcSDE 9.2, the user-defined binary columns were stored as LONG RAW. With the release of ArcSDE 9.2, the default was changed to BLOB.

Understanding the Oracle BLOB Data Type

The word BLOB is a database management system (DBMS) industry acronym for binary large object. BLOB columns were implemented several years ago by Oracle Corporation to replace the aging LONG RAW technology for storing binary data.

The LONG RAW column has a very simple structure: it simply exists in row with the data of the other columns. The BLOB column, on the other hand, has a complex structure.

The architecture of the BLOB data type is divided into three basic components: the BLOB column, the LOB segment, and the LOB index. The BLOB column stores the LOB locator (36 bytes) and binary data in row if it is less than 3,965 bytes and in-row storage has not been disabled for the column. (**Note:** Tests by ESRI have shown that allowing storage in row provides the best performance, so you are advised not to disable in-row storage.) If the binary data exceeds 3,964 bytes, the in-row storage space of the BLOB column is not allocated, and the LOB locator references the binary data stored in the LOB segment.

Therefore, a value stored in a BLOB column with in-row storage enabled will always be at least 36 bytes (the space allocated to the LOB locator) and may be as large as 4,000 bytes (the combined space allocated to the LOB locator and the maximum space that can be allocated to binary data stored in row).

The LOB segment is divided into chunks. Chunks must be a multiple of the Oracle data block size. For example, if the data block size is 8K, the LOB segment can be created with a minimum chunk size of 8K. If the length of the data stored within the LOB segment is 5,000 bytes, it will be stored in the LOB segment since it exceeds 3,964 bytes and the chunk size is 8K or 8,192 bytes. In this case, 3,192 bytes of the LOB segment chunk will remain unused. Transferring data from LONG RAW to BLOB can result in more space being required—perhaps as much as a 30 percent increase due to the unused space in the LOB segment. This is unavoidable if your data exceeds the 3,964 byte in-row storage threshold of the BLOB column.

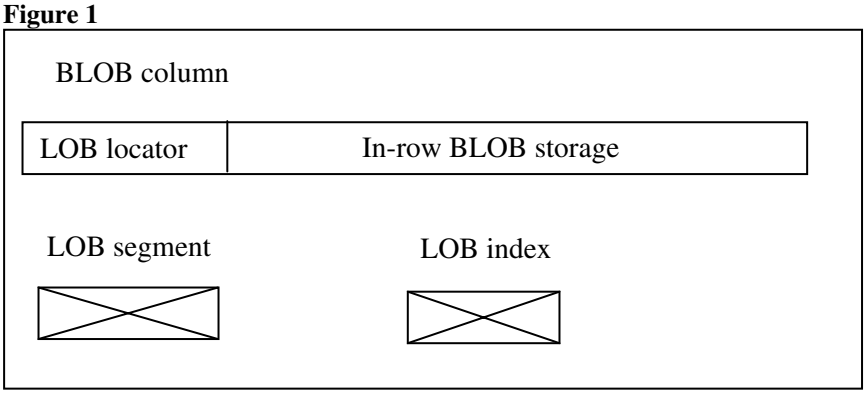
The 8K chunk size experiences the best I/O performance while wasting the least amount of space. The 16K chunk size will waste more space than an 8K chunk size. Therefore, to avoid the loss of space, you are advised to either re-create the database that currently has a 16K data block size with an 8K data block size or, if that is not possible, create LOB segments in tablespaces that have been created with an 8K block size. To do that, you will need to allocate an 8K buffer cache in the Oracle system global area (SGA).

Chunk sizes of 4K and 2K have been found to waste less space, but the increase in I/O cost does not warrant using them.

The LOB index is only used if the number of chunks addressed by the LOB locator exceeds 12; otherwise, the first 12 chunks are addressed by the LOB locator.

In the figures provided below, the three possible storage cases of binary data stored in a BLOB column are presented. In the first case, figure 1 illustrates binary data that is less than the 3,965 byte in-row storage threshold. If in-row storage is not disabled for the BLOB column, the LOB segment and the LOB index are not used. Typically, this will result in a faster fetch of the BLOB data due to the reduced number of I/O operations since Oracle does not need to access the LOB segment or the LOB index.

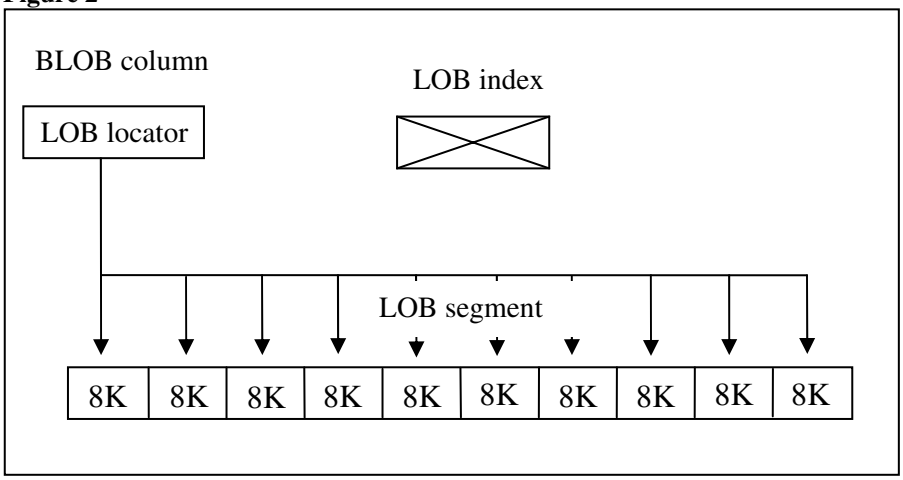
Figure 1



In this case, the binary data is less than 3965 bytes and will fit in row. Since the data is not stored in the LOB segment, neither the LOB segment nor the LOB index are referenced for this row of data.

Figure 2 illustrates the second case, in which the binary data is larger than 3,964 bytes and cannot fit in row. Therefore, the LOB locator references the binary data that is stored in the LOB segment. Since the binary data does not occupy more than 12 chunks in the LOB segment, the LOB locator stores its addresses. In this case, the LOB index is not used.

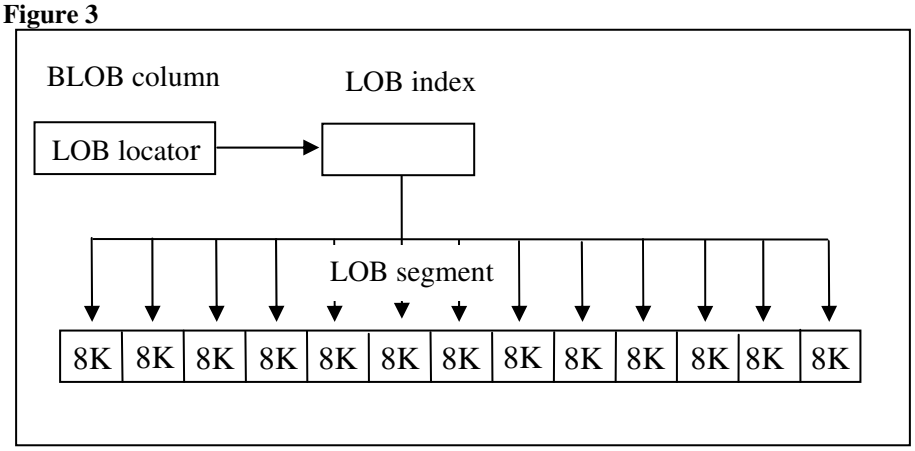
Figure 2



In this case, the binary data exceeds the in-row threshold of 3,964 bytes and must be stored in the LOB segment. However, since it occupies only 10 chunks, it can be referenced directly from the LOB locator, which can address up to 12 chunks.

The final case, figure 3, illustrates that the binary data is so large that the LOB index is required. In this case, the binary data exceeds the in-row storage plus requires more than12 chunks within the LOB segment to store it. For data this large, the LOB locator references the LOB index to obtain the location of the chunks within the LOB segment. This case is extremely rare for vector data and can be avoided for raster data.

Figure 3



In this case, the binary data exceeds the in-row threshold of 3,964 bytes and must be stored in the LOB segment. The binary data is so large that it must occupy 13 chunks; this must be addressed by the LOB index since the LOB locator can only address up to 12 chunks. In this case, the LOB locator will reference the LOB index to obtain the address of the binary data's chunks stored in the LOB segment.

Modifying the DBTUNE Parameters to Store BLOB Columns

The storage parameters of the DBTUNE table control how ArcGIS creates tables and indexes in Oracle. Some of the storage parameters also determine which data type will be used when a table is created.

For more information on how to maintain the DBTUNE table, consult the ArcGIS Desktop Help. Within the help system, you will find topics explaining how to organize the storage parameters into keywords as well as how to edit the values of the storage parameters.

Data Type DBTUNE Storage Parameters

The ArcSDE DBTUNE storage parameters, GEOMETRY_STORAGE, RASTER_STORAGE, and ATTRIBUTE_BINARY, determine which Oracle data type will be used to store ArcSDE data.

Note that the ArcSDE 9.1 RASTER_BINARY_TYPE storage parameter has been replaced at ArcSDE 9.2 with the RASTER_STORAGE parameter. If you currently have ArcSDE 9.1 installed, substitute all references to RASTER_STORAGE with RASTER_BINARY_TYPE. The RASTER_BINARY_TYPE parameter accepts only two values: LONGRAW and BLOB.

The GEOMETRY_STORAGE parameter controls the storage of vector data that is stored in a feature class. The RASTER_STORAGE parameter controls the storage of raster data that is stored in a raster dataset, raster catalog, or raster attribute. Finally, the ATTRIBUTE_BINARY parameter controls the storage of all other binary data that is not vector or raster. (**Note:** XML documents are always stored in a BLOB column, so there is no need to worry about converting them.)

The default value for the GEOMETRY_STORAGE parameter would create a feature table POINTS column of type LONG RAW. With the release of ArcSDE 9.2, the default value of RASTER_STORAGE and ATTRIBUTE_BINARY was changed to create a

column of type BLOB in compliance with Oracle's decision to begin deprecating the LONG RAW data type.

To create BLOB columns, the parameters need to be set as follows within a given DBTUNE keyword:

```
GEOMETRY_STORAGE SDELOB
RASTER_STORAGE   BLOB
ATTRIBUTE_BINARY BLOB
```

ESRI recommends the following LOB storage parameters for vector and raster data.

- Always enable in-row storage because most geographic information system (GIS) data fits within the 3,964 bytes in-row threshold. Performance is best when data is stored in row.
- Enable cache since ArcSDE data is frequently read.
- Since ArcSDE does not perform updates on BLOB data but instead performs only inserts and deletes, set the PCT_VERSION to 0 as there is no need to maintain older versions of the data within the LOB segment.
- You should not use a chunk size less than 8K. Chunk sizes of 2K and 4K increase the amount of I/O as the Oracle server process must fetch more chunks. You will probably find that an 8K chunk size will waste less space than 16K. If you use a chunk size of 2K or 4K, you will find that it wastes less space, but tests have found that the display time for most raster and vector data increases dramatically over storing in an 8K or 16K chunk size. Since the chunk size must always be a multiple of the data block size, the best data block size to use for storing GIS data in BLOBs is 8K.

The following is an example of how the raster DBTUNE storage parameters have been modified to accommodate a raster blocks table stored as a BLOB data type.

```
RASTER_STORAGE "BLOB"
BLK_STORAGE "PCTFREE 0 INITRANS 4 TABLESPACE RASTER
             LOB (BLOCK_DATA) STORE AS
             (TABLESPACE RASTER_LOB_SEGMENT
              CACHE PCTVERSION 0)"

AUX_STORAGE "PCTFREE 0 INITRANS 4 TABLESPACE RASTER
             LOB (OBJECT) STORE AS
             (TABLESPACE RASTER
              CACHE PCTVERSION 0)"
```

If the raster block pixel data is less than 3,965 bytes, it will be stored within the BLOCK_DATA column in the RASTER tablespace. However, if it exceeds this threshold, it will be stored in the LOB segment in the RASTER_LOB_SEGMENT tablespace. The LOB index is only used if the number of chunks exceeds 12. This is unlikely to happen for ArcSDE data. Consider a LOB segment with a chunk size of 8K. Before the LOB index is used, the ArcSDE binary data will need to exceed 96K.

The following is an example of how the vector DBTUNE storage parameters have been modified to accommodate the feature table stored in a BLOB data type:

```
GEOMETRY_STORAGE "SDELOB"
F_STORAGE "PCTFREE 0 INITRANS 4 TABLESPACE VECTOR
            LOB (POINTS) STORE AS
            (TABLESPACE VECTOR_LOB_SEGMENT
             CACHE PCTVERSION 0)"
```

If the feature's binary data is less than 3,965 bytes, it will be stored within the POINTS column in the VECTOR tablespace. However, if it exceeds this threshold, it will be stored in the LOB segment in the VECTOR_LOB_SEGMENT tablespace.

```
ATTRIBUTE_BINARY "BLOB"
B_STORAGE "PCTFREE 0 INITRANS 4 TABLESPACE BIZZTABS
            LOB (DOCUMENT) STORE AS
            (TABLESPACE BIZZ_LOB_SEGMENT
             CACHE PCTVERSION 0)"
A_STORAGE "PCTFREE 0 INITRANS 4 TABLESPACE BIZZTABS
            LOB (DOCUMENT) STORE AS
            (TABLESPACE BIZZ_LOB_SEGMENT
             CACHE PCTVERSION 0)"
```

If the business table's binary data is less than 3,965 bytes, it will be stored within the business table's BLOB column in the BIZZTABS tablespace. However, if it exceeds this threshold, it will be stored in the LOB segment in the BIZZ_LOB_SEGMENT tablespace. In this example, the BLOB column is called DOCUMENT. If the above B_STORAGE DBTUNE parameter is used to create a table that does not have a DOCUMENT column, the following error will be returned by Oracle:

```
ORA-00904: "DOCUMENT": invalid identifier
```

Therefore, it is not wise to add B_STORAGE or A_STORAGE parameters referencing a specific BLOB column to the DEFAULTS keyword, since the business table must contain these columns. Instead, you create separate DBTUNE keywords and add these storage parameters to the keywords. The keyword that contains the storage parameter is referenced during the creation of the table. It should also be noted that storage parameters of the DEFAULTS keyword are used if they are not included with a specific keyword. Due to this fact, it is not necessary to add a particular storage parameter within a keyword if its configuration string is identical to the same DEFAULTS keywords storage parameter. For instance, if all the storage parameters except B_STORAGE and A_STORAGE of a new keyword called ROADS have the same configuration string as those of the DEFAULTS keyword, you would only need to create the B_STORAGE and A_STORAGE parameters under the ROADS keyword. All other storage parameters would be read from the DEFAULTS keyword since they were not found in the ROADS keyword.

Using the SQL ALTER TABLE Statement

As of Oracle9i, the ALTER TABLE statement can directly convert a LONG RAW column to a BLOB column. This section explains the basic methodology of how you would use the ALTER TABLE statement to convert a LONG RAW column of any of your ArcSDE tables to a BLOB column.

The PL/SQL *convert_lr_to_blob* package included with this document uses the ALTER TABLE command to convert LONG RAW columns to BLOB. The stored procedure package also fetches the storage parameters from the DBTUNE table and updates the ArcSDE metadata. For more information on this package, refer to the section Using *convert_lr_to_blob*.

If you choose to convert ArcSDE tables from LONG RAW to BLOB by using the ALTER TABLE statement directly you should update the ArcSDE metadata by installing and the *sde_metadata* package included with this document. The package is installed by the owner of the geodatabase, which by default is the SDE user. You may grant the execute privilege to other users.

Execute the *update_metadata* stored procedure of this package to update the metadata after you have converted a LONG RAW column to BLOB. This stored procedure will only change the metadata if it detects that it is not synchronized with the current description of the table. The stored procedure requires two arguments: the qualified table name and the type. The type argument has three possible values: 'BUSINESS', 'LAYER', and 'RASTER'. If you converted a binary column of the business table MYTABLE owned by DAN, the stored procedure is executed as follows:

```
SQL> exec
sde_metadata.update_metadata('DAN.MYTABLE','BUSINESS');
```

If you converted the binary column of a feature table for the feature class MYTABLE owned by DAN, the stored procedure would be executed as follows:

```
SQL> exec
sde_metadata.update_metadata('DAN.MYTABLE','LAYER');
```

If you converted the binary column of a raster blocks table for the raster dataset MYTABLE owned by DAN, the stored procedure would be executed as follows:

```
SQL> exec
sde_metadata.update_metadata('DAN.MYTABLE','RASTER');
```

The basic syntax of the ALTER TABLE statement to convert a LONG RAW column to a BLOB column looks like the following statement:

```
ALTER TABLE SDE_BLK_10 MODIFY (BLOCK_DATA BLOB);
```

Using this basic syntax, Oracle will create three new segments within the tablespace in which the SDE_BLK_10 table was created, and it will remove the original segment that contained the LONG RAW column. Also, Oracle will invalidate all indexes on the table

following this type of modification. In this case, the unique index SDE_BLK_10_UK will be rebuilt with the following syntax:

```
ALTER INDEX SDE_BLK_10_UK REBUILD PARALLEL;
```

The above ALTER TABLE syntax is useful if the table is relatively small and you are not concerned about space. However, for large tables, the syntax needs to be more sophisticated, because you need to direct the placement of the three new segments, allowing you to recover the space vacated by the old segment. Therefore, for the purposes of this document, consider all tables containing LONG RAW columns to be large so that you need to recover the space following the conversion.

Let's assume that the following conditions exist:

1. The SDE_BLK_10 table contains the BLOCK_DATA column that is stored as a LONG RAW data type, and it has been created in the RASTER_LONG tablespace.
2. The RASTER_LONG tablespace has 20 data files assigned to it on 4 different logical drives. At the end of the conversion process, you want to be able to drop the RASTER_LONG tablespace, delete the 20 data files assigned to it, and reuse the disk space for something else.
3. Following the conversion, the segments of the SDE_BLK_10 table must be stored in the RASTER_BLOB and the RASTER_LOB_SEGMENT tablespaces.

First, disable logging on the SDE_BLK_10 table. Doing so significantly reduces the amount of redo generated during the conversion process.

```
ALTER TABLE SDE_BLK_10 NOLOGGING;
```

Next, alter the table to convert the BLOCK_DATA column from LONG RAW to BLOB.

```
ALTER TABLE SDE_BLK_10 MODIFY (BLOCK_DATA BLOB)
LOB (BLOCK_DATA) STORE AS
(TABLESPACE RASTER_LOB_SEGMENT NOCACHE NOLOGGING PCTVERSION
0);
```

This statement includes the LOB parameters that determine the LOB segment and LOB index to be stored in the RASTER_LOB_SEGMENT tablespace. Initially, caching and logging are disabled during the conversion to keep redo activity to a minimum. The space reserved for older versions of the LOB in the LOB segment is set to a minimum since ArcSDE never performs an update on this type of LOB data but instead performs a delete and inserts a new one if a change is required. For this reason, PCTVERSION is set to 0.

Following the conversion of the LONG RAW to a LOB column, the SDE_BLK_10 table remains in the RASTER_LONG tablespace. However, the LOB segment and the LOB index of the new LOB column were created in the RASTER_LOB_SEGMENT tablespace. Since one of the conditions is to remove the RASTER_LONG tablespace and recover the space it occupies, the SDE_BLK_10 table must be moved to the

RASTER_BLOB tablespace. The following statement is used to move the SDE_BLK_10 table to the RASTER_BLOB tablespace:

```
ALTER TABLE SDE_BLK_10 MOVE TABLESPACE RASTER_BLOB;
```

Now, assuming that no other segments from other tables and indexes in the database occupy the RASTER_LONG tablespace, it is possible to drop the tablespace and delete the data files assigned to it.

As a final step, caching and logging are enabled for the SDE_BLK_10 table and the index on the table, which was invalidated by the conversion process, is rebuilt.

```
ALTER TABLE SDE_BLK_10
MODIFY LOB (BLOCK_DATA) (CACHE);

ALTER TABLE SDE_BLK_10 LOGGING;

ALTER INDEX SDE_BLK_10_UK REBUILD PARALLEL;
```

Using the SQL TO_LOB Function

Another way to convert a LONG RAW column to a BLOB column is to use the Oracle TO_LOB function. Using the TO_LOB function, you create a new table with a BLOB column and load the contents of the original table containing the LONG RAW column into it. You then drop the original table and rename the new table to the original table name. The indexes and constraints on the table will need to be re-created.

The following is an example of how to convert the raster blocks table using the TO_LOB function. You will create the SDE_BLK_10 table with a LOB column as SDE_BLK_10_NEW. To do this, obtain the CREATE TABLE statement for the table from the DBMS_METADATA.GET_DDL stored procedure. If you are using SQL*Plus, you can use the SPOOL command to send the output of the stored procedure to a SQL script file, which you can modify and execute to create the new LOB version of the table.

```
SPOOL create_block_table.sql
SELECT DBMS_METADATA.GET_DDL('TABLE','SDE_BLK_10')
FROM DUAL;
SPOOL OFF
```

The following CREATE TABLE statement creates the new table with a BLOB column. In reality, the CREATE TABLE statement generated by the GET_DDL stored procedure would also create a unique index constraint on the integer columns and contain a number of storage parameters. You should remove the unique index constraint and modify the storage parameters so that the table, LOB segment, and LOB index end up in the appropriate tablespaces.

```
CREATE TABLE SDE_BLK_10_NEW
                   (RASTERBAND_ID NUMBER(38) NOT NULL,
                    RRD_FACTOR NUMBER(38) NOT NULL,
                    ROW_NBR NUMBER(38) NOT NULL,
                    COL_NBR NUMBER(38) NOT NULL,
                    BLOCK_DATA BLOB)
TABLESPACE RASTER_BLOB LOB (BLOCK_DATA) STORE AS
(TABLESPACE RASTER_LOB_SEGMENT CACHE PCTVERSION 0);
```

Insert the records of the original SDE_BLK_10 table into the new SDE_BLK_10_NEW table using the TO_LOB function.

```
INSERT INTO SDE_BLK_10_NEW
SELECT RASTERBAND_ID, RRD_FACTOR, ROW_NBR, COL_NBR,
       TO_LOB(BLOCK_DATA)
FROM SDE_BLK_10;
```

Now drop the SDE_BLK_10 table and rename the new table to the old name.

```
DROP TABLE SDE_BLK_10;

RENAME SDE_BLK_10_NEW TO SDE_BLK_10;
```

Finally, create the required unique index on the table's integer columns. This is a simplified version of the statement. In reality, you would also include storage parameters compliant with the requirements of your database. These storage parameters could be obtained from the output of the GET_DDL stored procedure used to generate the CREATE TABLE statement.

```
CREATE UNIQUE INDEX SDE_BLK_10_UK ON SDE_BLK_10
(RASTERBAND_ID, RRD_FACTOR, ROW_NBR, COL_NBR)
TABLESPACE RASTER_INDEX;
```

If you choose to convert ArcSDE tables from LONG RAW to BLOB by using the TO_LOB function, you should update the ArcSDE metadata by installing and running the *sde_metadata* package included with this document. The package is installed by the owner of the geodatabase, which, by default, is the SDE user. You may grant the execute privilege to other users.

Execute the *update_metadata* stored procedure of this package to update the metadata after you have converted a LONG RAW column to BLOB. This stored procedure will only change the metadata if it detects that it is out of sync with the current description of the table. The stored procedure requires two arguments: the qualified table name and the type. The type argument has three possible values: ‘BUSINESS’, ‘LAYER’, and ‘RASTER’. If you converted a binary column of the business table MYTABLE owned by DAN, the stored procedure is executed as follows:

```
SQL> exec
sde_metadata.update_metadata('DAN.MYTABLE','BUSINESS');
```

If you converted the binary column of a feature table for the feature class MYTABLE owned by DAN, the stored procedure would be executed as follows:

```
SQL> exec
sde_metadata.update_metadata('DAN.MYTABLE','LAYER');
```

If you converted the binary column of a raster blocks table for the raster dataset MYTABLE owned by DAN, the stored procedure would be executed as follows:

```
SQL> exec
sde_metadata.update_metadata('DAN.MYTABLE','RASTER');
```

Exporting to an ArcGIS File Geodatabase

Perhaps you are not able to allocate the additional space to the Oracle database required to either alter the table or create a new table using the TO_LOB function. If you do have disk space available on another system, at ArcGIS 9.2 you can export the raster dataset, raster catalog, or feature class to a file geodatabase. The only requirement is that the resulting file geodatabase must fit onto contiguous disk space. In other words, a file geodatabase cannot span multiple logical disk drives. If this is not possible, you should consider the next section, which discusses the sdeexport and sdeimport ArcSDE command line tools. These tools allow you to break the export file into logical volumes that can fit on multiple disk drives.

To determine how large the file geodatabase will be if you export a large raster dataset, you can use the sderaster ArcSDE command tool with the list operation and the –storage option. The total size displayed by this command is very close to the amount of space required when the raster dataset is exported to the file geodatabase. Consult the ArcSDE Administration Command Reference provided with ArcSDE for information on the sderaster command.

Once the ArcGIS object has been exported to the file geodatabase, you can modify the DBTUNE parameters to create a BLOB object instead of a LONG RAW, then delete the ArcGIS object and import it from the file geodatabase. Keep in mind that users may access the object from the file geodatabase during the import operation.

Using sdeexport and sdeimport

If you do not have ArcGIS 9.2 installed, you will not be able to export to a file geodatabase since the file geodatabase became available at this release. Therefore, your next option is to use the sdeexport and sdeimport commands. ArcSDE command line tools are not geodatabase aware and require that, upon importing the export file, you must register the raster dataset, raster catalog, or feature class with the geodatabase.

Once the object has been exported, it can be deleted. The DBTUNE parameters will need to be modified to create and store BLOB columns. Then the sdeimport command can be used to move the data back into the database. The data will be unavailable until the sdeimport command is complete.

Using *estimate_blob_storage*

The *estimate_blob_storage* package estimates the storage space that will be needed when a LONG RAW column is converted to a BLOB column. The package was designed to allow you to estimate the space needed for all the candidate tables within a schema for individual tables.

To install the package, you will need to be able to create a temporary table in your default tablespace that can store a maximum 10,000 rows of BLOB data. The *estimate_space* stored procedure of this package will convert up to the first 10,000 rows of a LONG RAW column into a temporary table with a BLOB column. Around 150 MB of storage space is required for 10,000 rows of a raster blocks table that stores 16-bit raster data.

The following privileges must be granted to the user that will execute the stored procedures of this package.

CREATE SESSION
CREATE TABLE
SELECT ANY TABLE
UNLIMITED TABLESPACE

If it is deemed undesirable to grant UNLIMITED TABLESPACE to the user, you must at least grant access to the tablespaces the existing tables are stored in. The package queries the USER_TABLESPACES view to obtain the block size of the tablespaces. This block size is then used to determine the space required for the lob segment space for the BLOB column should the binary data be large enough to be stored out of line.

The user which installs and owns the *estimate_blob_storage* package must be granted the CREATE PROCEDURE. Typically this is the user that executes the package but it does not need to be.

```
sqlplus system/<password>
```

GRANT CREATE SESSION TO <user>;
GRANT CREATE TABLE TO <user>;
GRANT SELECT ANY TABLE TO <user>;
GRANT UNLIMITED TABLESPACE TO <user>;
GRANT CREATE PROCEDURE TO <user>;

This package can be installed under any schema using the following syntax:

```
sqlplus <user>/<password>

@estimate_blob_storage.sps

@estimate_blob_storage.spb sde
```

Using the Oracle SQL*Plus utility, a connection is made by the owner of the schema under which the tools are to be installed. The package specification is created by running

the *estimate_blob_storage.sps* script, followed by the creation of the package body by running the *estimate_blob_storage.spb* script. Notice that an argument is required for the package body script. It is the name of the geodatabase under which the tables you want to convert are referenced. Support for multiple geodatabases has been included in the ArcSDE 9.2 release. If you have created another geodatabase other than the sde master geodatabase and added tables that store data in LONG RAW columns, you will need to create the package with the name of that geodatabase argument to estimate storage for those tables.

Perhaps in addition to the master sde geodatabase you have created a geodatabase under the joe schema. When installing the package body, you would substitute "joe" for sde in the above example:

```
@estimate_blob_storage.spb joe
```

The installation of the package body will use the argument to create stored procedures that reference the correct qualified ArcSDE metadata table. For instance, if the tables to be estimated for BLOB storage are referenced from the joe geodatabase, the JOE.TABLE_REGISTRY must be queried rather than the SDE.TABLE_REGISTRY.

After installing the package, you must grant privileges to the user who will execute the stored procedures. Note that this is not necessary if the user that installed the packages will also be executing them. To grant privileges, use the following syntax:

```
grant execute on estimate_blob_storage to alexis;
```

In this case, privileges have been granted to user alexis. For this user to execute a stored procedure of the package, the following syntax must be used:

```
exec gina.estimate_blob_storage.estimate();
```

In this case, the package was installed under the user gina schema but executed by the user alexis. If the package was installed under the user alexis schema and executed by alexis, a qualifying schema name would not have been necessary and could be executed as follows:

```
exec estimate_blob_storage.estimate();
```

The stored procedures display the results of estimates with the *put_line* stored procedure of the Oracle-provided *DBMS_OUTPUT* package. Therefore, to see the results of the *estimate_blob_storage* stored procedure, you will need to enable server output with the following syntax:

```
set SERVEROUTPUT on
```

Note: The output of the DBMS_OUTPUT package is written to a buffer that is not displayed to the output device until execution is returned to the parent process. Therefore, the output of the estimate_blob_storage package stored procedures will not be displayed

until they have finished execution. Therefore, be careful not to assume that the process is hung if it is not reporting output.

The SQL*PLUS SPOOL command is also useful for capturing the output of the package to a file for later examination. Writing the output to a file allows you to use your file editor to search through the output.

The hierarchy of the stored procedures within the package permits you to estimate the space for either the registered tables of an entire geodatabase or specific data objects within a geodatabase. The stored procedure references of the package that follow are described in descending order, where the highest in the hierarchy, *estimate*, is described first. This stored procedure will fetch the list of registered ArcSDE tables from the TABLE_REGISTRY and convert each one by passing it to the *estimate_business_table*.

The *estimate_business_table* stored procedure determines if the business table contains a LONG RAW column and estimates the space required to convert the business table with a call to *estimate_space*. It also determines if the business table has a dependent adds table and, if it does, estimates the space required to convert it with another call to *estimate_space*. The *estimate_layer* and *estimate_raster* stored procedure are called if the business table has a feature column or a raster column.

The output of the *estimate* stored procedure looks like this:

```
SQL> set SERVEROUTPUT ON
SQL> exec estimate_blob_storage.estimate();
Processing business table RASTER.COUNTIES and its
dependents.
*
The converted feature table RASTER.F253 will need at least
1.11 MB.
The associated lob segment will need at least 0 bytes.
*
Processing business table RASTER.LAKES and its dependents.
*
The converted raster block table RASTER.SDE_BLK_237 will
need at least 692.69 KB.
The associated lob segment will need at least 255.13 MB.
*
*
The converted raster auxiliary table RASTER.SDE_AUX_237
will need at least 6.24 KB.
The associated lob segment will need at least 0 bytes.
*
*
The converted feature table RASTER.F252 will need at least
191 bytes.
The associated lob segment will need at least 0 bytes.
*
```

```
Processing business table RASTER.ROUTE_DATA and its
dependents.
*
The table RASTER.ROUTE_DATA is empty.
*
*
The table RASTER.A472 is empty.
*

PL/SQL procedure successfully completed.

SQL>
```

For each business table the *estimate_business_table* stored procedure examines, it displays the message "Processing business table <name> and its dependents." If neither the business table nor any of its dependents contain a LONG RAW column, this is the only message that you will see regarding this table. No further processing will be performed.

However, if LONG RAW columns are found, you will see other messages. For the RASTER.COUNTIES table the message is:

```
Processing business table RASTER.COUNTIES and its
dependents.
*
The converted feature table RASTER.F253 will need at least
1.11 MB.
The associated lob_segment will need at least 0 bytes.
```

The RASTER.COUNTIES table has a dependent feature table, RASTER.F253, that will need at least 1.11 MB after the LONG RAW column has been converted to BLOB. The LOB segment for the BLOB column of this feature table is estimated to store no chunks. That is because the *estimate_space* stored procedure, which actually does all the work, did not find any features that exceeded the 3,964 in-row threshold when it sampled the first 5,000 to 10,000 rows of the feature table.

You should also consider the message for the RASTER.LAKES business table:

```
Processing business table RASTER.LAKES and its dependents.
*
The converted raster block table RASTER.SDE_BLK_237 will
need at least 692.69 KB.
The associated lob segment will need at least 255.13 MB.
```

The RASTER.LAKES table has a dependent raster blocks table, RASTER.SDE_BLK_237, that will need at least 692.69 KB for the raster blocks table but will require at least 255.13 MB in the LOB segment. So much space is required in the LOB_SEGMENT table and so little is required for the raster blocks table because vast

majority, if not all, of the blocks exceed the 3,964 in-row threshold and must be stored in the LOB segment.

Consider also the message for the RASTER.ROUTE_DATA table:

```
Processing business table RASTER.ROUTE_DATA and its
dependents.
*
The table RASTER.ROUTE_DATA is empty.
```

In this case, a LONG RAW column was found in the ROUTE_DATA table, but since the table is empty, the *estimate_space* stored procedure is not able to provide an estimate.

The estimate *Stored Procedure*

The *estimate* stored procedure estimates the space that will be required when the LONG RAW columns of all registered business tables are converted to BLOB.

The syntax is as follows:

```
PROCEDURE estimate (v_owner IN name_t DEFAULT NULL,
                    verbose IN verbose_t DEFAULT FALSE);
```

where

- v_owner is the owner of the tables that are to be estimated. If you do not specify an owner, the *estimate* stored procedure will estimate the space required to convert all registered tables and their dependents. If you do specify an owner, only the owner's registered tables and their dependents are converted. The default value is NULL.
- verbose is a Boolean that, when set to TRUE, displays the elapsed time of the table conversions or error messages. The default value is FALSE.

The *estimate* stored procedure calls the *estimate_business_table* stored procedure for each business table referenced in the TABLE_REGISTRY that is owned by the specified owner if one was passed in as an argument. If an owner is not specified, *estimate_business_table* is called for all tables listed in the TABLE_REGISTRY.

The estimate_business_table *Stored Procedure*

The *estimate_business_table* stored procedure estimates the space that will be required when the LONG RAW column of a registered business table and any associated feature class or raster dataset is converted to BLOB.

The syntax is as follows:

```
PROCEDURE estimate_business_table(
          qtabname IN name_t,
          verbose IN verbose_t DEFAULT FALSE);
```

where

- qtabname is the qualified name of the business table. This is a required argument.

- verbose is a Boolean that, when set to TRUE, displays the elapsed time of the table conversions or error messages. The default value is FALSE.

The qualified table name is one of <schema>.<table>. If the schema is omitted from the qualified table name, the *estimate_business_table* stored procedure assumes that the business table and all its dependents are owned by the currently connected user.

The *estimate_business_table* stored procedure calls the *estimate_table, estimate_layer,* and *estimate_raster* stored procedures. The *estimate_business_table* stored procedure will estimate the space required for the following:

- The table
- The BLOB's LOB segment after the business table's LONG RAW column is converted to BLOB
- The LONG RAW column of any of the table's feature class, raster table, and adds table dependents

Execute the *estimate_business_table* stored procedure if you want to estimate all the tables containing LONG RAW columns associated with a given business table.

The estimate_layer *Stored Procedure*

The *estimate_layer* stored procedure estimates the space that will be required when the LONG RAW column of a feature class is converted to BLOB.

The syntax is as follows:

```
PROCEDURE estimate_layer (qtabname IN name_t,
                          verbose IN verbose_t DEFAULT FALSE);
```

where

- qtabname is the qualified name of the business table and the feature table associated with it. This is a required argument.

- verbose is a Boolean that, when set to TRUE, displays elapsed time of the table conversions or error messages. The default value is FALSE.

The qualified table name is one of <schema>.<table>. If the schema is omitted from the qualified table name, the *estimate_layer* stored procedure assumes that the feature table is owned by the currently connected user.

The *estimate_layer* stored procedure calls the *estimate_space* stored procedure with the name of the feature table, the DBTUNE configuration keyword, and the F_STORAGE storage parameter.

Execute the *estimate_layer* stored procedure if you want to estimate the space required by a feature table and the LOB segment after converting its LONG RAW column to BLOB.

The estimate_raster *Stored Procedure*

The *estimate_raster* stored procedure estimates the space that will be required when the LONG RAW columns of a raster dataset is converted to BLOB.

The syntax is as follows:

```
PROCEDURE estimate_raster (qtabname IN name_t,
                           verbose IN verbose_t DEFAULT FALSE);
```

where

- qtabname is the qualified name of the business table to which the raster blocks table and raster auxiliary table belong. This is a required argument.
- verbose is a Boolean that, when set to TRUE, displays the elapsed time of the table conversions or error messages. The default value is FALSE.

The qualified table name is one of <schema>.<table>. If the schema is omitted from the qualified table name, the *estimate_raster* stored procedure assumes that the raster tables are owned by the currently connected user.

The *estimate_raster* stored procedure calls the *estimate_space* stored procedure once with the name of the raster blocks table and again with the name of the raster auxiliary table.

Execute the *estimate_raster* stored procedure if you want to estimate the storage space required to convert the LONG RAW column of a raster block table and raster auxiliary table to BLOB.

Using *convert_lr_to_blob*

The *convert_lr_to_blob* package converts LONG RAW columns to BLOB columns using the Oracle ALTER TABLE statement. The package was designed to allow you to convert all the candidate tables within a schema or convert individual tables.

The first part of the conversion requires the installation of the *sde_metadata* package by the owner of the geodatabase. This package contains the *update_metadata* stored procedure. This stored procedure will update the geodatabase metadata after a column has been converted from LONG RAW to BLOB.

The sde user is the owner of the master geodatabase and the one you are likely to be using. However, if you are converting the columns under a geodatabase (multiple geodatabases were introduced at 9.2) other than the sde master, you will need to install the *sde_metadata* package under that user.

This is an example of installing the package under the sde user. Note that you would connect as the sde user using SQL*Plus in this case, and run the sde_metadata.sps script to install the package specification. When you run the sde_metadata.spb script to install the package body, you include the sde owner name as an argument. The argument is used to create the qualified table names of the geodatabase metadata tables that are used by many of the stored procedures within the package.

```
sqlplus sde/<password>

@sde_metadata.sps

@sde_metadata.spb sde
```

After you install this package, you must grant execute privileges to the user who will execute the stored procedure. In this example, user alexis will execute this stored procedure.

```
grant execute on sde_metadata to alexis;
```

The following privileges must be granted to the user that will execute the stored procedures of the *convert_lr_to_blob* package.

CREATE SESSION
ALTER ANY TABLE
ALTER ANY INDEX
SELECT ANY TABLE
UNLIMITED TABLESPACE

If it is deemed undesirable to grant UNLIMITED TABLESPACE to the user, you must at least grant access to the tablespaces the existing tables are stored in as well as any tablespaces that tables will be moved to or lob segments will be created in.

The user which installs and owns the convert_lr_to_blob package must be granted the CREATE PROCEDURE. Typically this is the user that executes the package but it does not need to be.

```
sqlplus system/<password>

GRANT CREATE SESSION TO <user>;
GRANT ALTER ANY TABLE TO <user>;
GRANT ALTER ANY INDEX TO <user>;
GRANT SELECT ANY TABLE TO <user>;
GRANT UNLIMITED TABLESPACE TO <user>;
GRANT CREATE PROCEDURE TO <user>;
```

The second part of the installation requires you to install the *convert_lr_to_blob* package, which can be installed by any user. In this example, user joe is connecting using SQL*Plus and running the convert_lr_to_blob.sps script to install the package specification. When you run the convert_lr_to_blob.spb script to install the package body, you include as an argument the geodatabase owner name. The argument is used to create the qualified table names of the geodatabase metadata tables that are used by many of the stored procedures within the package.

```
sqlplus joe/<password>
```

```
@convert_lr_to_blob.sps

@convert_lr_to_blob.spb sde
```

After you have installed the package, you must grant access to the users that are going to execute it. In this example, the user that will execute the stored procedures is alexis, so joe (who owns the stored procedures) would grant execute privileges to alexis. You must connect as the joe user and grant execute privileges to alexis on the packages as follows:

```
sqlplus joe/<password>

grant execute on convert_lr_to_blob to alexis;
```

In this case, privileges have been granted to user alexis. For alexis to execute a stored procedure of the package, the following syntax must be used:

```
exec joe.convert_lr_to_blob.convert();
```

In this case, the package was installed under the user joe schema but executed by the user alexis. If the package was installed under the user alexis schema and executed by alexis, a qualifying schema name would not have been necessary and could be executed as follows:

```
exec convert_lr_to_blob.convert();
```

The stored procedures display the results of conversions with the *put_line* stored procedure of the Oracle-provided *DBMS_OUTPUT* package. Therefore, to see the results of the *convert_lr_to_blob* stored procedure, you will need to enable server output with the following syntax:

```
set SERVEROUTPUT on
```

Note: The output of the DBMS_OUTPUT package is written to a buffer that is not displayed to the output device until execution is returned to the parent process. Therefore, the output of the convert_lr_to_blob package stored procedures will not be displayed until they have finished execution. Therefore, be careful not to assume that the process is hung if it is not reporting output.

The SQL*PLUS SPOOL command is also useful for capturing the output of the package to a file for later examination. Writing the output to a file allows you to use your file editor to search through the output.

The hierarchy of the stored procedures within the package permits you to convert either the registered tables of an entire geodatabase or specific data objects within a geodatabase. The stored procedure references of the package that follow are described in descending order, where the highest in the hierarchy, *convert*, is described first. This stored procedure will fetch the list of registered ArcSDE tables from the

TABLE_REGISTRY and convert each one by passing it to the *convert_business_table* stored procedure.

The *convert_business_table* stored procedure determines if the business table contains a LONG RAW column and converts it with a call to *convert_table*. It also determines if the business table has a dependent adds table and, if it does, converts it with another call to *convert_table*. The *convert_layer* and *convert_raster* stored procedures are called if the business table has a feature column or a raster column.

The normal output of the *convert* stored procedure appears as follows:

```
SQL> set SERVEROUTPUT ON
SQL> SPOOL myoutput.txt
SQL> exec convert_lr_to_blob.convert();
Processing table QUAKES
Processing table WORLD

PL/SQL procedure successfully completed.

SQL>SPOOL OFF
SQL>
```

You must enable the SQL*Plus environment SERVEROUTPUT to display PL/SQL output; otherwise, SQL*Plus will only report whether the PL/SQL stored procedure was successful or not:

```
SQL> exec convert_lr_to_blob.convert();

PL/SQL procedure successfully completed.

SQL>
```

You can obtain a detailed output of the processing of any of the *convert_lr_to_blob* stored procedures by setting the verbose argument to TRUE (its default value is FALSE). The following output is an example of the *convert* stored procedure with the verbose argument set to TRUE.

```
SQL> set SERVEROUTPUT ON
SQL> SPOOL myoutput.txt
SQL> exec convert_lr_to_blob.convert(verbose=>TRUE);
Processing table QUAKES
Converting business table RASTER.QUAKES LONG RAW column to
BLOB.
start time: 2007-03-07 02:03:40
Converting table RASTER.QUAKES
FYI: QUAKES does not contain a LONG RAW column.
finish time: 2007-03-07 02:03:40
Elapsed time: 0 seconds.
```

```
Converting feature table RASTER.F312 points column to BLOB.
start time: 2007-03-07 02:03:40
Converting table RASTER.F312
finish time: 2007-03-07 02:03:40
Elapsed time: 0 seconds.
Processing table WORLD
Converting business table RASTER.WORLD LONG RAW column to
BLOB.
start time: 2007-03-07 02:03:40
Converting table RASTER.WORLD
FYI: WORLD does not contain a LONG RAW column.
finish time: 2007-03-07 02:03:40
Elapsed time: 0 seconds.
Converting raster block table RASTER.SDE_BLK_290 block_data
column to BLOB.
start time: 2007-03-07 02:03:40
Converting table RASTER.SDE_BLK_290
finish time: 2007-03-07 02:03:03
Elapsed time: 23 seconds.
Converting raster auxillary table RASTER.SDE_AUX_290 object
column to BLOB
start time: 2007-03-07 02:03:03
Converting table RASTER.SDE_AUX_290
finish time: 2007-03-07 02:03:03
Elapsed time: 0 seconds.
Converting feature table RASTER.F311 points column to BLOB.
start time: 2007-03-07 02:03:03
Converting table RASTER.F311
finish time: 2007-03-07 02:03:03
Elapsed time: 0 seconds.

PL/SQL procedure successfully completed.

SQL> SPOOL OFF
SQL>
```

Within this output, there are a number of points of interest such as the FYI statements. These are information messages telling you that the procedure attempted to do something but was unable to. For instance, the above output contains the following statement:

```
FYI: QUAKES does not contain a LONG RAW column.
```

In this case, the procedure did not find a LONG RAW column on the RASTER.QUAKES table. However, when the RASTER.F312 is converted, the FYI message does not appear. This indicates that a LONG RAW column was found and converted.

Notice also that the elapsed time to convert the feature table, RASTER.F312, was 0. The small size of the F312 table accounts for why it was converted at subsecond speed. Not until the *convert* stored procedure encounters the much larger raster blocks table, RASTER.SDE_BLK_290, does the *convert* stored procedure register an elapsed time of 23 seconds.

The convert *Stored* Procedure

The *convert* stored procedure converts the LONG RAW column of all registered business tables and any dependent adds tables, feature tables, raster blocks tables, and raster auxiliary tables. A business table is registered if it is referenced in the TABLE_REGISTRY of the geodatabase.

The syntax is as follows:

```
PROCEDURE convert (v_owner IN name_t DEFAULT NULL,
                   kword IN name_t DEFAULT 'DEFAULTS',
                   verbose IN verbose_t DEFAULT FALSE);
```

where

- v_owner is the owner of the tables that are to be converted. If you do not specify an owner, the *convert* stored procedure will convert all registered tables and their dependents. If you do specify an owner, only the owner's registered tables and their dependents are converted. The default value is NULL.
- kword is the configuration keyword. The default value is DEFAULTS.
- verbose is a Boolean that, when set to TRUE, displays the elapsed time of the table conversions or error messages. The default value is FALSE.

The *convert* stored procedure calls the *convert_business_table* stored procedure for each business table referenced in the TABLE_REGISTRY.

Execute the *convert* stored procedure if you want to convert all the tables containing LONG RAW columns in your ArcSDE geodatabase. Before you call this stored procedure, verify that your tablespaces contain enough space.

The convert_business_table *Stored Procedure*

The *convert_business_table* stored procedure converts the LONG RAW columns of the business table, adds table, feature table, raster blocks table, and raster auxiliary table. The syntax is as follows:

```
PROCEDURE convert_business_table(
          qtabname IN name_t,
          kword IN name_t DEFAULT 'DEFAULTS',
          verbose IN verbose_t DEFAULT FALSE);
```

where

- qtabname is the qualified name of the business table. This is a required argument.
- kword is the configuration keyword. The default is DEFAULTS.

- verbose is a Boolean that, when set to TRUE, displays the elapsed time of the table conversions or error messages. The default value is FALSE.

The qualified table name is one of <schema>.<table>. If the schema is omitted from the qualified table name, the *convert_business_table* stored procedure assumes that the business table and all its dependents are owned by the currently connected user.

The *convert_business_table* stored procedure calls the *convert_table, convert_layer,* and *convert_raster* stored procedures. The *convert_business_table* stored procedure will convert the LONG RAW column of the business table to BLOB as well as the LONG RAW column of any of its feature class, raster table, and adds table dependents.

The B_STORAGE parameter is passed to the *convert_table* stored procedure if the business table contains a LONG RAW column. The A_STORAGE parameter is passed to the *convert_table* stored procedure if the business table's adds table contains a LONG RAW column.

Execute the *convert_business_table* stored procedure if you want to convert all the tables containing LONG RAW columns associated with a given business table. Before you call this stored procedure, verify that your tablespaces contain enough space.

The convert_layer *Stored Procedure*

The *convert_layer* stored procedure will convert a feature table's POINTS column from LONG RAW to BLOB. The syntax is as follows:

```
PROCEDURE convert_layer (qtabname IN name_t,
                         kword IN name_t DEFAULT 'DEFAULTS',
                         verbose IN verbose_t DEFAULT FALSE);
```

where

- qtabname is the qualified name of the business table with which the feature table is associated. This is a required argument.
- kword is the DBTUNE configuration keyword from which the F_STORAGE parameter will be read. The default value is DEFAULTS.
- verbose is a Boolean that, when set to TRUE, displays elapsed time of the table conversion or error messages. The default value is FALSE.

The qualified table name is one of <schema>.<table>. If the schema is omitted from the qualified table name, the *convert_layer* stored procedure assumes that the feature table is owned by the currently connected user.

The *convert_layer* stored procedure calls the *convert_table* stored procedure with the name of the feature table, the DBTUNE configuration keyword, and the F_STORAGE parameter.

Execute the *convert_layer* stored procedure if you want to convert the feature tables containing LONG RAW columns in your ArcSDE geodatabase. Before you call this stored procedure, verify that the LOB segment and table segment tablespaces contain enough space.

The convert_raster *Stored Procedure*

The *convert_raster* stored procedure converts a raster blocks table's BLOCK_DATA column and the raster auxiliary table's OBJECT column from LONG RAW to BLOB. The syntax is as follows:

```
PROCEDURE convert_raster (qtabname IN name_t,
                          kword IN name_t DEFAULT 'DEFAULTS',
                          verbose IN verbose_t DEFAULT FALSE);
```

where

- qtabname is the qualified name of the business table to which the raster blocks table and raster auxiliary table belong. This is a required argument.
- kword is the configuration keyword. The default value is DEFAULTS.
- verbose is a Boolean that, when set to TRUE, displays the elapsed time of the table conversions or error messages. The default value is FALSE.

The qualified table name is one of <schema>.<table>. If the schema is omitted from the qualified table name, the *convert_raster* stored procedure assumes that the raster tables are owned by the currently connected user.

The *convert_raster* stored procedure calls the *convert_table* stored procedure once with the name of the raster blocks table and again with the name of the raster auxiliary table. The configuration keyword and the BLK_STORAGE parameter are passed with the raster block table name. The configuration keyword and the AUX_STORAGE parameter are passed with the raster auxiliary table name.

Execute the *convert_raster* stored procedure if you want to convert the raster block table and raster auxiliary table containing LONG RAW columns in your ArcSDE geodatabase. Before you call this stored procedure, verify that your LOB segment and table segment tablespaces contain enough space.

Conversion Scenarios

The following provides a number of possible ways to convert the binary columns of a geodatabase from LONG RAW to BLOB. You should select the one that best fits your situation or modify one to meet your needs.

Scenario 1—Convert the entire geodatabase at one time.

This scenario will work if the geodatabase has the following properties:

- Adequate space exists to complete the conversion.
- The geodatabase does not contain any massive raster datasets or raster catalogs that might require special handling.

- The conversion can be completed during off hours.

For this scenario, the procedure is straightforward.

1. Perform a full DBMS backup of the geodatabase to ensure that it can be recovered in the event of a catastrophic failure during the conversion process.
2. Install the *estimate_blob_storage* package and run the *estimate* stored procedure to obtain the converted table and LOB segment sizes.
3. Create new tablespaces or extend your existing tablespaces to hold the converted tables and LOB segments.
4. Modify your DBTUNE table. Add a new keyword with the appropriate LOB storage parameters assigned to the correct tablespaces.
5. Install and run the *convert_lr_to_blob* and *sde_metadata* packages. Run the *convert* stored procedure to convert. This stored procedure will cycle through all tables registered to the geodatabase, converting their LONG RAW columns to BLOB.
6. Follow up by checking for any tables that may not have been converted. Possible reasons for this include tables were not registered to the geodatabase or the *convert* stored procedure encountered a problem during the conversion such as a lack of storage space.

Scenario 2—Some tables are very large and will need to be handled separately.

This scenario works best for geodatabases that have a few very large tables that need to be converted before the remainder of the geodatabase is converted. In this case, the geodatabase will have the following properties:

- Space is at a premium and must be recovered following the conversion of the large tables.
- The geodatabase contains a few large tables that require special handling.
- The conversion of each large table can be completed during off hours. However, the conversion may need to be completed over the course of several days.

For this scenario, the procedure requires special handling of a number of large tables before the rest of the tables in the geodatabase can be converted.

1. Perform a full DBMS backup of the geodatabase to ensure that it can be recovered in the event of a catastrophic failure during the conversion process.
2. Install the *estimate_blob_storage* package and run the *estimate_business_table* stored procedure to obtain the converted table and LOB segment sizes for the large tables. This stored procedure will estimate the space required for the business table and any associated adds, feature, and raster blocks tables.
3. Create new tablespaces or extend your existing tablespaces to hold the converted tables and LOB segments of the large tables. It is possible that you will not be able to acquire enough space for all the large tables. If this is

the case, you may need to convert the tables one at a time, recovering the space after the old table is dropped and using that space for conversion of the next table.

4. Modify your DBTUNE table. Add a new keyword with the appropriate LOB storage parameters assigned to the correct tablespaces in which the large tables will be stored.
5. Install and run the *convert_lr_to_blob* and *sde_metadata* packages. Run the *convert_layer* stored procedure if you are converting a feature table. Run the *convert_raster* stored procedure if you are converting a raster blocks table. Run the *convert_business* table to convert a business table and any associated adds, feature, or raster blocks tables.
6. Once the large tables have been converted, use the *estimate* stored procedure of the *estimate_blob_storage* package to obtain an estimate for the remainder of the registered tables that contain LONG RAW columns.
7. If necessary, create new tablespaces or extend existing ones to hold the converted tables and LOB segments of the remaining tables.
8. Use the *convert* stored procedure of the *convert_lr_to_blob* package to convert the LONG RAW columns of the remaining tables to BLOB.
9. Follow up by checking for any tables that may not have been converted. The possible reasons for this include tables were not registered to the geodatabase or the *convert* stored procedure encountered a problem during the conversion such as a lack of storage space.

Scenario 3—The geodatabase is massive. It either contains several large raster datasets or hundreds to thousands of vector layers, some of which may be of significant size.

This scenario will require special handling of all data, as it will require a significant amount of time to convert the data. In some cases, the conversion of individual tables might exceed the amount of time available during off hours. Geodatabases that fall into this scenario have the following properties:

- Space may be at a premium, requiring the database administrator (DBA) to develop a strategy to recover space as the conversion process proceeds.
- The geodatabase contains massive raster datasets or a very large number of vector layers of significant size.
- The conversion of some tables may require more time than the off-hour limit permits.

Tables that are too large to be converted within the off-hour time limit can be converted during the day.

The rows of these tables should not be edited while they are being converted. However, queries to the table will continue normally right up to the point that the ALTER TABLE statement substitutes the new segment containing the BLOB column and drops the old segment containing the LONG RAW column. At this point, the indexes become invalidated. While the *convert_table* stored procedure rebuilds the indexes, applications

accessing these tables appear to halt. Therefore, warn application users to expect interruptions to their workflow during the reconstruction of the indexes.

The procedure is the same as scenario 2 with the exception that the large tables will have to be converted during normal daytime activities.

An alternate procedure is to follow the procedure laid out in scenario 4, which is to copy the geodatabase and perform the conversion on the copy.

Scenario 4—The geodatabase is a data source for a Web service that must remain up all the time.

In this scenario, it will be challenging to convert the tables without interrupting the requests to the server portal unless the tables can be duplicated in another geodatabase. Therefore, you will need to back up the geodatabase, recover it to a new location, and convert the copy of the geodatabase. Following the conversion, a substitution occurs, replacing the converted geodatabase with the original unconverted one. Changes made to the original geodatabase during the conversion process will be lost unless they are transferred to the copy.

Under this scenario, the geodatabase has the following properties:

- Resources must be available to duplicate the geodatabase.
- The geodatabase can contain larger raster datasets or many vector layers of significant size.
- Service to the geodatabase cannot be interrupted.

1. Perform a full DBMS backup of the geodatabase to recover it to a new location.
2. Recover the geodatabase to a separate Oracle instance. From this point on, all work is performed on the recovered copy of the geodatabase. Edits to tables in the original geodatabase should be minimized because these changes will need to be duplicated on the copy. Otherwise, the changes will be lost when the copy of the geodatabase is substituted for the original.
3. Install the *estimate_blob_storage* package and run the *estimate_business_table* stored procedure to obtain the converted table and LOB segment sizes for the large tables. This stored procedure will estimate the space required for the business table and any associated adds, feature, and raster blocks tables.
4. Create new tablespaces or extend your existing tablespaces to hold the converted tables and LOB segments of the large tables. It is possible that you will not be able to acquire enough space for all the large tables. If this is the case, you may need to convert the tables one at a time, recovering the space after the old table is dropped and using that space for conversion of the next table.
5. Modify your DBTUNE table. Add a new keyword with the appropriate LOB storage parameters assigned to the correct tablespaces in which the large tables will be stored.

6. Install and run the *convert_lr_to_blob* and *sde_metadata* packages. Run the *convert_layer* stored procedure if you are converting a feature table. Run the *convert_raster* stored procedure if you are converting a raster blocks table. Run *convert_business_table* to convert a business table and any associated adds, feature, or raster blocks tables.
7. Once the large tables have been converted, use the *estimate* stored procedure of the *estimate_blob_storage* package to obtain an estimate for the remainder of the registered tables that contain LONG RAW columns.
8. If necessary, create new tablespaces or extend existing ones to hold the converted tables and LOB segments of the remaining tables.
9. Use the *convert* stored procedure of the *convert_lr_to_blob* package to convert the LONG RAW columns of the remaining tables to BLOB.
10. Follow up by checking for any tables that may not have been converted. The possible reasons for this include tables were not registered to the geodatabase or the *convert* stored procedure encountered a problem during the conversion such as a lack of storage space.
11. Substitute the converted copy of the geodatabase for the original one.
    i. Perform a full backup on the converted geodatabase.
    ii. Point the Web service to the converted geodatabase.
    iii. If the original Oracle instance must be used, recover the converted geodatabase to the original Oracle instance.
    iv. Point the Web service to the original Oracle instance that now has a converted geodatabase.