15

Migrating Columns to SecureFile LOBs

Oracle recommends that you migrate your existing columns that use the LONG or LONG RAW datatype or BasicFile LOB storage to the SecureFile LOB storage. This chapter covers various techniques to help with this migration.

All forms of LONG data types (LONG, LONG RAW, LONG VARCHAR, LONG VARRAW) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the LONG data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use CLOB and NCLOB data types for large amounts of character data.

> **Note:**
>
> All discussions in this chapter are valid for migrating the LONG datatype to CLOB or NCLOB, and the LONG RAW datatype to BLOB. Most of the text in this chapter talks just about the LONG datatype for brevity.

- Migration Considerations
  This section discusses various factors to be considered while migrating LOB data types or storage.
- Migration Methods
  This section describes various methods you can use to migrate LONG or BasicFile LOB data to SecureFile storage.
- Other Considerations While Migrating LONG Columns to LOBs
  This section describes some more considerations when migrating LONG columns to LOBs.

15.1 Migration Considerations

This section discusses various factors to be considered while migrating LOB data types or storage.

Space requirements

Most migration techniques copy the contents of the table into a new space, and free the old space at the end of the operation. This temporarily doubles the space requirements. If space is limited, then you can perform the BasicFile to SecureFile migration one partition at a time.

Preventing Generation of REDO Data When Migrating

Migrating LONG datatype or BasicFiles LOB columns to SecureFile generates redo data, which can slow down the performance during the migration.

Redo changes for a column being converted to SecureFiles LOB are logged only if the storage characteristics of the LOB column indicate LOGGING. The logging setting (LOGGING or NOLOGGING) for the LOB column is inherited from the tablespace in which the LOB is created.

You can prevent redo space generation during migration to SecureFiles LOB by following the following steps:

1. Specify the NOLOGGING storage parameter for any new SecureFiles LOB columns.
2. Turn LOGGING on when the migration is complete.
3. Make a backup of the tablespaces containing the table and the LOB column.

15.2 Migration Methods

This section describes various methods you can use to migrate LONG or BasicFile LOB data to SecureFile storage.

Topics

- Migrating LOBs with SecureFiles Migration Utility
  This is the recommended method to migrate BasicFile LOB data to SecureFile storage. This utility encapsulates all the functionality offered by Online Redefinition and saves you the time and effort involved in manually running a series of API calls.
- Migrating LOBs with Online Redefinition
  Use Online redefinition to migrate LONG or BasicFile LOB data to SecureFile storage by running several API calls.
- Migrating LOBs with Data Pump
  Oracle Data Pump can either recreate tables as they are in your source database, or recreate LOB columns as SecureFile LOBs.

15.2.1 Migrating LOBs with SecureFiles Migration Utility

This is the recommended method to migrate BasicFile LOB data to SecureFile storage. This utility encapsulates all the functionality offered by Online Redefinition and saves you the time and effort involved in manually running a series of API calls.

Advantages

- No need to take the table or partition offline.
- Perform the migration at the database, schema, table or LOB segment level.
- After migrating the data, you can also use the SecureFiles Migration Utility to compress the SecureFile LOBs.

Disadvantages

- Additional storage equal to the entire table or partition required and all LOB segments must be available.
- Global indexes must be rebuilt.

To migrate BasicFile LOB data to SecureFile storage using the SecureFiles migration utility:

1. Run the following command as is to create a table.

   ```
   create table migration_config (ctime date, data clob , constraint c1
   check(data is json));
   ```

2. Make a single entry in the table to specify the schema, table, and columns that you want to migrate. Enter `first` as the value for `run_type` as this is the first time you are running the script. For others, provide values based on your environment.

Example Command

The following example shows a example single entry that specifies the objects that you want to migrate.

```
insert into migration_config values
    (systimestamp,
    '{"schema_name" : ["TEST2"],
    "table_name" : ["TEST1.TAB_DEFERRED_SEGCREATION1",
"TEST1.TAB_NON_LOB1",
    "TEST1.BASIC1A", "TEST1.BASIC3A"],
    "column_name" : ["TEST1.TAB_PARTS1.a",
    "TEST1.BASIC123.a", "TEST1.BASIC125.a"],
    "metadata_schema_name" : "TEMP1",
    "run_type" : "first",
    "directory_path" : "<full path to folder for log files>",
    "compress_storage_rec_threshold" : 5000,
    "trace" : 1}');
```

Where,

- **schema_name**: Mandatory. Specify a comma-separated list of schema names that you want to migrate and compress. If you do not specify a value, {"schema_name" : []}, the entire schema is not migrated. Instead the script checks for finer granularity that may be specified in the table_name or column_name arrays.
- **table_name**: Mandatory. Specify a comma-separated list of tables that you want to migrate and compress. You must enter the name in the following format, <schema_name>.<table_name>, where the schema name prefixes the table name. In the following example, TEST1 is the name of the schema and BASIC1A and BASIC3A are the names of the tables.

  ```
  "table_name" : ["TEST1.BASIC1A", "TEST1.BASIC3A"]
  ```

  If you do not specify a value, "table_name" : [], then the script checks for finer granularity that may be specified in the column_name array.
- **column_name**: Mandatory. Specify a comma-separated list of columns that you want to migrate and compress. You must enter the name in the following format, <schema_name>.<table_name>.<column_name>, where the schema name and table name prefixes the column name.
  If you do not specify a value, "column_name" : [], then all LOB columns belonging to the specified table are migrated.
- **metadata_schema_name**: Mandatory. Enter a unique schema name. This is a temporary schema which the script uses to store metadata tables or reports that are generated during the migration. The schema must have a default ASSM tablespace to store intermediate data.
- **run_type**: Mandatory. The permitted values are first, second, and third corresponding to the three stages in which you execute the script.
- **directory_path**: Mandatory. Enter the complete path to the folder where you want to save the generated log files if trace is 1.
- **compress_storage_rec_threshold**: Optional. The script recommends compressing LOBs that are above the storage threshold that you enter. The default value is 5000 MB.

- **trace**: Optional. Set this to 1 to enable tracing. The log files are saved in the folder path that you specify in directory_path. If you do not enter a value or enter any other value, then the log files are not generated.

Example entry to migrate and compress all tables in specified schemas

The following example entry in the migration_config table would migrate and then compress all tables and columns in the specified schemas, TEST1 and TEST2 when you run the script.

```
insert into migration_config values
    (systimestamp,
    '{"schema_name" : ["TEST1","TEST2"],
    "table_name" : [],
    "segment_name" : [],
    "metadata_schema_name" : "TEMP1",
    "run_type" : "first",
    "directory_path" : "<full_path>",
    "trace" : 1}');
```

Example entry to migrate and compress all schemas, tables, and columns

The following example entry in the migration_config table would migrate and then compress ***all*** schemas, tables, and columns when you run the script.

```
insert into migration_config values (
    systimestamp,
    '{"schema_name" : [],
    "table_name" : [],
    "segment_name" : [],
    "metadata_schema_name" : "TEMP1",
    "run_type" : "first",
    "directory_path" : "<full_path>",
    "trace" : 1}');
```

3. Run the script as SYS user after creating the table and inserting a row with details of the required configuration.

   ```
   SQL> @securefile_migration_script.sql
   ```

   The following three reports are generated and stored as tables in the temporary table space. You had specified the name of the temporary table space in metadata_schema_name in a previous step.

   - sf_migration_table_ddl_report table provides details about the DDL information for all the tables in specified schema for all users.
   - sf_migration_index_ddl_report table provides details about index DDL information for all the tables in specified schema for all users.
   - sf_migration_basicfile_report table lists all the BasicFile LOB segments under all users.

4. Look at the reports and identify if there are any BasicFile LOBs that you do not want to migrate, such as when BasicFile LOB segment is on MSSM tablespace. If you do not want to migrate a BasicFile LOB, change the value of the Allow migrate column in the

sf_migration_basicfile_report table to **N** for the specific LOB that you do not want to migrate. The default value for all LOBs is **Y**.

5. Create an interim table for the LOBs that you want to migrate. Ensure that the interim table that you create is identical to the original table in all respects except for the property that you intent to change.

   Example Original Table

   For example, let's consider that the existing BasicFile table that you want to migrate has the following properties.

   ```
   CREATE TABLE basic1a
   (
       a CLOB,
       b NUMBER
   );
   ```

   Example Interim Table

   As shown in the following example, the interim table that you create must be identical to the original table, except for the store as securefile in the interim table and the name of the interim table must be unique.

   ```
   CREATE TABLE basic1a_int1
   (
       a CLOB,
       b NUMBER
   ) lob(a) store as securefile;
   ```

6. Update the row that you have inserted to change the run_type to second as shown in the following command.

   ```
   SQL> update migration_config set data = JSON_TRANSFORM(data, SET
   '$.run_type' = 'second');
   ```

7. Run the script as SYS user.

   ```
   SQL> @securefile_migration_script.sql
   ```

   The BasicFile LOB data is migrated to SecureFile storage.

   After the migration is completed successfully, the script generates the following reports and stores it as tables in the temporary table space. You had specified the name of the temporary table space in metadata_schema_name in a previous step.

   - sf_migration_lob_statistics_report table provides details about the LOBs, such as storage, compression ratio, compression recommendation, compression type.
   - sf_migration_lob_compression_report table contains recommendations about which LOBs should be compressed in the COMPRESS_RECOMMENDATION column as Y (yes) or N (no). The recommendation is based on the information available in sf_migration_lob_statistics_report.

8. Look at sf_migration_lob_compression_report and identify if you want to compress the LOBs are per the recommendation. If you do not want to compress the LOBs, skip the next steps.

9. Look at the reports and identify if there are any SecureFile LOBs that you do not want to compress. If you do not want to compress a LOB, change the value of the `compress_recommendation` column in `sf_migration_lob_statistics_report` to **N** for the specific LOB that you do not want to compress. The default value is **Y** for all LOBs.

10. Create an interim table for the LOBs that you want to compress. Ensure that the interim table that you create is identical to the original table in all respects except for the property that you intent to change.

    Example Original Table

    For example, let's consider that the existing BasicFile table that you want to migrate has the following properties.

    ```
    CREATE TABLE basic1a
    (
        a CLOB,
        b NUMBER
    );
    ```

    Example Interim Table

    As shown in the following example, the interim table that you create must be identical to the original table, except for the `LOB(a) STORE AS SECUREFILE seg_basic1a (ENABLE STORAGE IN ROW CACHE LOGGING COMPRESS MEDIUM)` in the interim table and the name of the interim table must be unique.

    ```
    CREATE TABLE comp_basic1a_int1
    (
        a CLOB,
        b NUMBER
    ) LOB(a) STORE AS SECUREFILE seg_basic1a (ENABLE STORAGE IN ROW CACHE
    LOGGING COMPRESS MEDIUM);
    ```

    Where, you can specify `LOW`, `MEDIUM`, and `HIGH` as the options to provide varying degrees of compression.

11. Update the row that you have inserted to change the `run_type` to `third`.

    ```
    SQL> update migration_config set data = JSON_TRANSFORM(data, SET
    '$.run_type' = 'third');
    ```

12. Run the script as `SYS` user.

    ```
    SQL> @securefile_migration_script.sql
    ```

    The `sf_migration_lob_compression_report` report provides details about the updated status of compression.

15.2.2 Migrating LOBs with Online Redefinition

Use Online redefinition to migrate LONG or BasicFile LOB data to SecureFile storage by running several API calls.

Consider using the SecureFiles Migration Utility, which automates this task and saves you the time and effort involved in manually running a series of API calls. See Migrating LOBs with SecureFiles Migration Utility.

While online redefinition for LONG to LOB migration must be performed at the table level, BasicFile to SecureFile migration can be performed at the table or partition level.

Online Redefintion Advantages

- No need not take the table or partition offline
- Can be done in parallel.
  To set up parallel execution of online redefinition, run:

  ```
  ALTER SESSION FORCE PARALLEL DML;
  ```

Online Redefinition Disadvantages

- Additional storage equal to the entire table or partition required and all LOB segments must be available
- Global indexes must be rebuilt

Example 15-1 Online Redefinition for Migrating Tables from BasicFiles LOB storage to SecureFile LOB storage

```
REM Grant privileges required for online redefinition.
GRANT EXECUTE ON DBMS_REDEFINITION TO pm;
GRANT ALTER ANY TABLE TO pm;
GRANT DROP ANY TABLE TO pm;
GRANT LOCK ANY TABLE TO pm;
GRANT CREATE ANY TABLE TO pm;
GRANT SELECT ANY TABLE TO pm;
REM Privileges required to perform cloning of dependent objects.
GRANT CREATE ANY TRIGGER TO pm;
GRANT CREATE ANY INDEX TO pm;
CONNECT pm/pm

-- This forces the online redefinition to execute in parallel
ALTER SESSION FORCE parallel dml;

DROP TABLE cust;
CREATE TABLE cust(c_id NUMBER PRIMARY KEY,
    c_zip NUMBER,
    c_name VARCHAR(30) DEFAULT NULL,
    c_lob CLOB
);
INSERT INTO cust VALUES(1, 94065, 'hhh', 'ttt');
-- Creating Interim Table
-- There is no requirement to specify constraints because they are
-- copied over from the original table.
CREATE TABLE cust_int(c_id NUMBER NOT NULL,
```

```
    c_zip NUMBER,
    c_name VARCHAR(30) DEFAULT NULL,
    c_lob CLOB
) LOB(c_lob) STORE AS SECUREFILE (NOCACHE FILESYSTEM_LIKE_LOGGING);
DECLARE
    col_mapping VARCHAR2(1000);
BEGIN
-- map all the columns in the interim table to the original table
    col_mapping :=
    'c_id c_id , '||
    'c_zip c_zip , '||
    'c_name c_name, '||
    'c_lob c_lob';
DBMS_REDEFINITION.START_REDEF_TABLE('pm', 'cust', 'cust_int', col_mapping);
END;
/
DECLARE
    error_count pls_integer := 0;
BEGIN
    DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS('pm', 'cust', 'cust_int',
      1, TRUE,TRUE,TRUE,FALSE, error_count);
    DBMS_OUTPUT.PUT_LINE('errors := ' || TO_CHAR(error_count));
END;
/
EXEC DBMS_REDEFINITION.FINISH_REDEF_TABLE('pm', 'cust', 'cust_int');
-- Drop the interim table
DROP TABLE cust_int;
DESC cust;
-- The following insert statement fails. This illustrates
-- that the primary key constraint on the c_id column is
-- preserved after migration.
INSERT INTO cust VALUES(1, 94065, 'hhh', 'ttt');
SELECT * FROM cust;
```

Example 15-2 Online Redefinition for Migrating Tables from the LONG datatype to a SecureFile LOB

The steps for `LONG` to LOB migration are:

- Create an empty interim table. This table holds the migrated data when the redefinition process is done. In the interim table:
  - Define a `CLOB` or `NCLOB` column for each `LONG` column in the original table that you are migrating.
  - Define a `BLOB` column for each `LONG RAW` column in the original table that you are migrating.
- Start the redefinition process. To do so, call `DBMS_REDEFINITION.START_REDEF_TABLE` and pass the column mapping using the `TO_LOB` operator as follows:

```
DBMS_REDEFINITION.START_REDEF_TABLE(
    'schema_name',
    'original_table',
    'interim_table',
    'TO_LOB(long_col_name) lob_col_name',
    'options_flag',
    'orderby_cols');
```

where `long_col_name` is the name of the `LONG` or `LONG RAW` column that you are converting in the original table and `lob_col_name` is the name of the LOB column in the interim table. This LOB column holds the converted data.

- Call the `DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS` procedure as described in the related documentation.
- Call the `DBMS_REDEFINITION.FINISH_REDEF_TABLE` procedure as described in the related documentation.

The following example demonstrates online redefinition for `LONG` to LOB migration.

```
REM Grant privileges required for online redefinition.
GRANT execute ON DBMS_REDEFINITION TO pm;
GRANT ALTER ANY TABLE TO pm;
GRANT DROP ANY TABLE TO pm;
GRANT LOCK ANY TABLE TO pm;
GRANT CREATE ANY TABLE TO pm;
GRANT SELECT ANY TABLE TO pm;

REM Privileges required to perform cloning of dependent objects.
GRANT CREATE ANY TRIGGER TO pm;
GRANT CREATE ANY INDEX TO pm;

CONNECT pm/pm

-- This forces the online redefinition to execute in parallel
ALTER SESSION FORCE parallel dml;

DROP TABLE cust;
CREATE TABLE cust(c_id   NUMBER PRIMARY KEY,
                  c_zip  NUMBER,
                  c_name VARCHAR(30) DEFAULT NULL,
                  c_long LONG
                  );
INSERT INTO cust VALUES(1, 94065, 'hhh', 'ttt');

-- Creating Interim Table
-- There is no requirement to specify constraints because they are
-- copied over from the original table.
CREATE TABLE cust_int(c_id  NUMBER NOT NULL,
                  c_zip  NUMBER,
                  c_name VARCHAR(30) DEFAULT NULL,
                  c_long CLOB
                  );

DECLARE
 col_mapping VARCHAR2(1000);
BEGIN
--  map all the columns in the interim table to the original table
 col_mapping :=
               'c_id             c_id  , '||
               'c_zip            c_zip , '||
               'c_name           c_name, '||
               'to_lob(c_long)   c_long';

DBMS_REDEFINITION.START_REDEF_TABLE('pm', 'cust', 'cust_int', col_mapping);
END;
/

DECLARE
 error_count PLS_INTEGER := 0;
BEGIN
  DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS('pm', 'cust', 'cust_int',
                                          1, true, true, true, false,
                                          error_count);

  DBMS_OUTPUT.PUT_LINE('errors := ' || to_char(error_count));
END;
/

EXEC  DBMS_REDEFINITION.FINISH_REDEF_TABLE('pm', 'cust', 'cust_int');

-- Drop the interim table
DROP TABLE cust_int;

DESC cust;

-- The following insert statement fails. This illustrates
-- that the primary key constraint on the c_id column is
-- preserved after migration.

INSERT INTO cust VALUES(1, 94065, 'hhh', 'ttt');

SELECT * FROM cust;
```

15.2.3 Migrating LOBs with Data Pump

Oracle Data Pump can either recreate tables as they are in your source database, or recreate LOB columns as SecureFile LOBs.

When Oracle Data Pump recreates tables, by default it recreates them as they existed in the source database. Therefore, if a LOB column was a BasicFiles LOB in the source database, Oracle Data Pump attempts to recreate it as a BasicFile LOB in the imported database. However, you can force creation of LOBs as SecureFile LOBs in the recreated tables by using a `TRANSFORM` parameter for the command line, or by using a `LOB_STORAGE` parameter for the `DBMS_DATAPUMP` and `DBMS_METADATA` packages.

Example:

```
impdp system/manager directory=dpump_dir schemas=lobuser dumpfile=lobuser.dmp
      transform=lob_storage:securefile
```

Note:

The transform name is not valid in transportable import.

> **See Also:**
>
> TRANSFORM for using TRANSFORM parameter to convert to SecureFile LOBs

You can use the keyword `HIDDEN` to distinguish a default inline LOB size from a user-specified one.

Example:

```
CREATE TABLE <tab> (…) LOB (L1) STORE AS … [ENABLE STORAGE IN ROW [4000|8000]
        HIDDEN];
```

Restrictions on Migrating LOBs with Data Pump

You can't use SecureFile LOBs in non-ASSM tablespace. If the source database contains LOB columns in a tablespace that does not support ASSM, then you'll see an error message when you use Oracle Data Dump to recreate the tables using the securefile clause for LOB columns.

To import non-ASSM tables with LOB columns, run another import for these tables without using `TRANSFORM=LOB_STORAGE:SECUREFILE`.

Example:

```
impdp system/manager directory=dpump_dir schemas=lobuser dumpfile=lobuser.dmp
```

15.3 Other Considerations While Migrating LONG Columns to LOBs

This section describes some more considerations when migrating LONG columns to LOBs.

- Migrating Applications from LONGs to LOBs
  Most APIs that work with `LONG` data types in the PL/SQL, JDBC and OCI environments are enhanced to also work with LOB data types.
- Alternate Methods for LOB Migration
  Online Redefinition is the preferred way for migrating `LONG` data types to LOBs. However, if keeping the application online during the migration is not your primary concern, then you can also use one of the following ways to migrate `LONG` data to LOBs.

15.3.1 Migrating Applications from LONGs to LOBs

Most APIs that work with `LONG` data types in the PL/SQL, JDBC and OCI environments are enhanced to also work with LOB data types.

These APIs are collectively referred to as the data interface for LOBs. Among other things, the data interface provides the following benefits:

- Changes needed are minimal in PL/SQL, JDBC and OCI applications that use tables with columns converted from `LONG` to LOB data types.
- You can work with LOB data types in your application without having to deal with LOB locators.

See Also:

- Data Interface for LOBs for details on JDBC and OCI APIs included in the data interface.
- SQL Semantics and LOBs for details on SQL syntax supported for LOB data types.
- PL/SQL Semantics for LOBs for details on PL/SQL syntax supported for LOB data types.

Note:

You can use various techniques to do either of the following:

- Convert columns of type `LONG` to either `CLOB` or `NCLOB` columns
- Convert columns of type `LONG RAW` to `BLOB` type columns

Unless otherwise noted, discussions in this chapter regarding LONG to LOB conversions apply to both of these data type conversions.

However, there are differences between `LONG` and LOB data types that may impact your application migration plans or require you to modify your application.

Identify Application Rewrite Using utldtree.sql

When you migrate your table from `LONG` to LOB column types, certain parts of your PL/SQL application may require rewriting. You can use the utility, `rdbms/admin/utldtree.sql`, to determine which parts.

The `utldtree.sql` utility enables you to recursively see all objects that are dependent on a given object. For example, you can see all objects which depend on a table with a `LONG` column. You can only see objects for which you have permission.

Instructions on how to use `utldtree.sql` are documented in the file itself. Also, `utldtree.sql` is only needed for PL/SQL. For SQL and OCI, you have no requirement to change your applications.

SQL Differences

- Indexes: LONG and LOB data types only support domain and functional indexes.
  - Any domain index on a `LONG` column must be dropped before converting the `LONG` column to LOB column. This index may be manually recreated after the migration.
  - Any function-based index on a `LONG` column is unusable during the conversion process and must be rebuilt after converting. Application code that uses function-based indexing should work without modification after the rebuild.
    To rebuild an index after converting, use the following steps:
    1. Select the index from your original table as follows:

       ```
       SELECT index_name FROM user_indexes WHERE table_name='LONG_TAB';
       ```

> **Note:**
>
> The table name must be capitalized in this query.

2. For each selected index, use the command:

   ```
   ALTER INDEX <index> REBUILD
   ```

- Constraints: The only constraint allowed on LONG columns are NULL and NOT NULL. All constraints of the LONG columns are maintained for the new LOB columns. To alter the constraints for these columns, or alter any other columns or properties of this table, you have to do so in a subsequent ALTER TABLE statement.
- Default Values: If you do not specify a default value, then the default value for the LONG column becomes the default value of the LOB column.
- Triggers: Most of the existing triggers on your table are still usable. However, you cannot have LOB columns in the UPDATE OF list of an AFTER UPDATE OF trigger. For example, the following create trigger statement is not valid:

  ```
  CREATE TABLE t(lobcol CLOB);
  CREATE TRIGGER trig AFTER UPDATE OF lobcol ON t ...;
  ```

  LONG columns are allowed in such triggers. So, you must drop the AFTER UPDATE OF triggers on any LONG columns before migrating to LOBs.
- Clustered tables: LOB columns are not allowed in clustered tables, whereas LONGs are allowed. If a table is a part of a cluster, then any LONG or LONG RAW column cannot be changed to a LOB column.

Empty LOBs Compared to NULL and Zero Length LONGs

A LOB column can hold an *empty* LOB. An empty LOB is a LOB locator that is fully initialized, but not populated with data. Because LONG data types do not use locators, the *empty* concept does not apply to LONG data types.

Both LOB column values and LONG column values, inserted with an initial value of NULL or an empty string literal, have a NULL value. Therefore, application code that uses NULL or zero-length values in a LONG column functions exactly the same after you convert the column to a LOB type column.

In contrast, a LOB initialized to empty has a non-NULL value as illustrated in the following example:

```
CREATE TABLE long_tab(id NUMBER, long_col LONG);
CREATE TABLE lob_tab(id NUMBER, lob_col CLOB);

REM     A zero length string inserts a NULL into the LONG column:
INSERT INTO long_tab values(1, '');

REM     A zero length string inserts a NULL into the LOB column:
INSERT INTO lob_tab values(1, '');

REM     Inserting an empty LOB inserts a non-NULL value:
INSERT INTO lob_tab values(1, empty_clob());
```

```
DROP TABLE long_tab;
DROP TABLE lob_tab;
```

Overloading with Anchored Types

For applications using anchored types, some overloaded variables resolve to different targets during the conversion to LOBs. For example, given the procedure `p` overloaded with specifications 1 and 2:

```
procedure p(l long) is ...;       -- (specification 1)
procedure p(c clob) is ...;       -- (specification 2)
```

and the procedure call:

```
declare
     var  longtab.longcol%type;
   BEGIN
     ...
   p(var);
     ...
END;
```

Prior to migrating from `LONG` to LOB columns, this call would resolve to specification 1. Once `longtab` is migrated to LOB columns this call resolves to specification 2. Note that this would also be true if the parameter type in specification 1 were a `CHAR`, `VARCHAR2`, `RAW`, `LONG RAW`.

If you have migrated you tables from `LONG` columns to LOB columns, then you must manually examine your applications and determine whether overloaded procedures must be changed.

Some applications that included overloaded procedures with LOB arguments before migrating may still break. This includes applications that do not use `LONG` anchored types. For example, given the following specifications (1 and 2) and procedure call for procedure `p`:

```
procedure p(n number) is ...;       -- (1)
procedure p(c clob) is ...;         -- (2)

p('123');                 -- procedure call
```

Before migrating, the only conversion allowed was `CHAR` to `NUMBER`, so specification 1 would be chosen. After migrating, both conversions are allowed, so the call is ambiguous and raises an overloading error.

Some Implicit Conversions Are Not Supported for LOB Data Types

PL/SQL permits implicit conversion from `NUMBER`, `DATE`, `ROW_ID`, `BINARY_INTEGER`, and `PLS_INTEGER` data types to a `LONG`; however, implicit conversion from these data types to a LOB is not allowed.

If your application uses these implicit conversions, then you have to explicitly convert these types using the `TO_CHAR` operator for character data or the `TO_RAW` operator for binary data. For example, if your application has an assignment operation such as:

```
number_var := long_var;  -- The RHS is a LOB variable after converting.
```

then you must modify your code as follows:

```
number_var := TO_CHAR(long_var);
-- Assuming that long_var is of type CLOB after conversion
```

The following conversions are not supported for LOB types:

- BLOB to VARCHAR2, CHAR, or LONG
- CLOB to RAW or LONG RAW

This applies to all operations where implicit conversion takes place. For example if you have a SELECT statement in your application as follows:

```
SELECT long_raw_column INTO my_varchar2 VARIABLE FROM my_table
```

and long_raw_column is a BLOB after converting your table, then the SELECT statement produces an error. To make this conversion work, you must use the TO_RAW operator to explicitly convert the BLOB to a RAW as follows:

```
SELECT TO_RAW(long_raw_column) INTO my_varchar2 VARIABLE FROM my_table
```

The same holds for selecting a CLOB into a RAW variable, or for assignments of CLOB to RAW and BLOB to VARCHAR2.

15.3.2 Alternate Methods for LOB Migration

Online Redefinition is the preferred way for migrating LONG data types to LOBs. However, if keeping the application online during the migration is not your primary concern, then you can also use one of the following ways to migrate LONG data to LOBs.

> **See Also:**
>
> Migration Considerations

Using ALTER TABLE to Convert LONG Columns to LOB Columns

You can use the ALTER TABLE statement in SQL to convert a LONG column to a LOB column.

To do so, use the following syntax:

```
ALTER TABLE [<schema>.]<table_name>
   MODIFY ( <long_column_name> { CLOB | BLOB | NCLOB }
  [DEFAULT <default_value>]) [LOB_storage_clause];
```

For example, if you had a table that was created as follows:

```
CREATE TABLE Long_tab (id NUMBER, long_col LONG);
```

then you can change the column long_col in table Long_tab to data type CLOB using following ALTER TABLE statement:

```
ALTER TABLE Long_tab MODIFY ( long_col CLOB );
```

> **Note:**
>
> The ALTER TABLE statement copies the contents of the table into a new space, and frees the old space at the end of the operation. This temporarily doubles the space requirements.

Note that when using the ALTER TABLE statement to convert a LONG column to a LOB column, only the following options are allowed:

- DEFAULT option, which enables you to specify a default value for the LOB column.
- The *LOB_storage_clause*, which enables you to specify the LOB storage characteristics for the converted column. This clause can be specified in the MODIFY clause.

Other ALTER TABLE options are not allowed when converting a LONG column to a LOB type column.

Copying a LONG to a LOB Column Using the TO_LOB Operator

You can use the CREATE TABLE AS SELECT statement or the INSERT AS SELECT statement with the TO_LOB operator to copy data from a LONG column to a CLOB or NCLOB column, or from a LONG RAW column to a BLOB column. For example, if you have a table with a LONG column that was created as follows:

```
CREATE TABLE Long_tab (id NUMBER, long_col LONG);
```

then you can do the following to copy the column to a LOB column:

```
CREATE TABLE Lob_tab (id NUMBER, clob_col CLOB);
INSERT INTO Lob_tab SELECT id, TO_LOB(long_col) FROM long_tab;
COMMIT;
```

If the INSERT statement returns an error because of lack of undo space, then you can incrementally migrate LONG data to the LOB column using the WHERE clause. After you ensure that the data is accurately copied, you can drop the original table and create a view or synonym for the new table using one of the following sequences:

```
DROP TABLE Long_tab;
CREATE VIEW Long_tab (id, long_col) AS SELECT * from Lob_tab;
```

or

```
DROP TABLE Long_tab;
CREATE SYNONYM Long_tab FOR Lob_tab;
```

This series of operations is equivalent to changing the data type of the column Long_col of table Long_tab from LONG to CLOB. With this technique, you have to re-create any constraints, triggers, grants, and indexes on the new table.

Use of the TO_LOB operator is subject to the following limitations:

- You can use TO_LOB to copy data to a LOB column, but not to a LOB attribute of an object type.
- You cannot use TO_LOB with a remote table. For example, the following statements do not work:

  ```
  INSERT INTO tb1@dblink (lob_col) SELECT TO_LOB(long_col) FROM tb2;
  INSERT INTO tb1 (lob_col) SELECT TO_LOB(long_col) FROM tb2@dblink;
  CREATE TABLE tb1 AS SELECT TO_LOB(long_col) FROM tb2@dblink;
  ```

- You cannot use the TO_LOB operator in the CREATE TABLE AS SELECT statement to convert a LONG or LONG RAW column to a LOB column when creating an index organized table.

  To work around this limitation, create the index organized table, and then do an INSERT AS SELECT of the LONG or LONG RAW column using the TO_LOB operator.
- You cannot use TO_LOB inside any PL/SQL block.